UNIVERSITY OF SURREY

Meta-Compilation for C++

A dissertation for the degree of Doctor of Philosophy

Edward D. Willink

January 4, 2000

DEFENDED POSITION

- Meta-programming is essential for significant applications.
- Meta-programming is best achieved in C++ by elevating existing language constructs to the meta-level.
- The FOG solution for meta-programming is logical, systematic and adequate for typical applications.
- Compilation problems of both FOG and plain C++ are resolvable by introduction of a clear partitioning between syntactic and semantic analyses.

RELEVANT PUBLICATIONS BY THE AUTHOR

1 Edward D. Willink, Alexander V. Shafarenko and Vyacheslav B. Muchnick. *Use of F-code as a Very High Level Intermediate Language for DSP.* Proceedings of the Third International Euro-Par Conference, Euro-Par'97, 820-823, August 1997. In Euro-Par'97, Parallel Processing, Christian Lengauer, Martin Griebl and Sergei Gorlatch (Eds.), LNCS 1300, Springer-Verlag, August 1997.
http://www.ee.surrey.ac.uk/Personal/E.Willink/europar97.{gz,pdf,ps,zip}

2 Edward D. Willink. *Low level Object Oriented DSP Development in C++*. Proceedings of the 8th International Conference on Signal Processing Applications and Technology, ICSPAT'97, 894-898, San Diego, California, September 1997.
http://www.ee.surrey.ac.uk/Personal/E.Willink/icspat97.{gz,pdf,ps,zip}

3 Edward D. Willink and Vyacheslav B. Muchnick. *Preprocessing C++: Substitution and Composition.* Proceedings of the Eastern European Conference on Technology of Object Oriented Languages and Systems, ISBN 954-90484-1-1, 122-135, Blagoevgrad, Bulgaria, June 1999.
http://www.computing.surrey.ac.uk/research/dsrg/fog/FogToolsEE1.{gz,pdf,ps,zip}

4 Edward D. Willink and Vyacheslav B. Muchnick. *Preprocessing C++: Meta-Class Aspects.* Proceedings of the Eastern European Conference on Technology of Object Oriented Languages and Systems, ISBN 954-90484-1-1, 136-149, Blagoevgrad, Bulgaria, June 1999.
http://www.computing.surrey.ac.uk/research/dsrg/fog/FogToolsEE2.{gz,pdf,ps,zip}

5 Edward D. Willink and Vyacheslav B. Muchnick. *Weaving a Way Past the C++ One Definition Rule.* Position paper for the *Aspect-Oriented Programming Workshop at ECOOP'99*, June 1999.
http://www.computing.surrey.ac.uk/research/dsrg/fog/AopEcoop99.{gz,pdf,ps,zip}

6 Edward D. Willink and Vyacheslav B. Muchnick. *An Object-Oriented Preprocessor fit for C++.* Submitted to IEE Proceedings-Software, August 1999.

ABSTRACT

Software Engineering progresses as improvements are made in languages and methodologies. Significant advances have been made through the use of Object-Oriented Programming; exploiting the effective support available in C++. Further evolution of OOP involving the use of design patterns and aspects requires additional language support.

Increased flexibility in the declaration of objects is proposed in the form of the FOG (Flexible Object Generator) language, which is a superset of C++ implemented by a translator to C++. FOG generalises C++ syntax and supports compile-time meta-programming and reflection.

The syntax generalisations provide the freedom for programmers to organise code to suit programming concerns and eliminate the need for duplication between interfaces and implementations. Further generalisations define composition policies for repeated declarations so that classes, arrays, enumerations and functions may all be extended. These composition policies support the weaving necessary for re-useable implementation of design patterns and for Aspect Oriented Programming.

A declarative form of meta-programming is supported by derivation rules, which specify how a declaration specified in a base class is to be reinterpreted in derived classes. Automated generation of derived functionality is important for a number of design patterns.

More general meta-programming is provided by elevating most run-time concepts to the meta-level, allowing conditional and iterated manipulation of declarations at compile-time. Compile-time execution enables subsequent run-time code to be optimised to suit application requirements.

The use of meta-variables and meta-functions, a well-disciplined lexical context for meta-programming and meta-level execution provide a complete replacement for the traditional C preprocessor functionality, satisfying Stroustrup’s goal of making Cpp redundant. The new functionality is integrated with the language, fits within an Object-Oriented framework and provides adequate support for modern Software Engineering practices.

The C++ grammar is known to pose a significant parsing challenge and to require context dependent type and template knowledge. This creates considerable difficulties when meta-programming occurs in unresolved contexts. A new approach to parsing C++ has therefore been developed that defers the use of type and template information. This approach leads to a simpler grammar implementation. An extended form of regular expression is presented and used to predict known ambiguities and then show that this simpler grammar covers the C++ grammar.

ACKNOWLEDGEMENTS

My various academic supervisors gave invaluable support. Chris Jesshope got me started and prompted me to broaden my programming outlook. Sasha Shafarenko never allowed me to become complacent. Slava Muchnick explained some of the intricacies of data parallelism and then encouraged me as the research direction changed. Slava read drafts enthusiastically, challenging my English from a uniquely Russian perspective.

My industrial supervisor, Edward Stansfield, ensured everything ran smoothly.

Brian Hillam offered encouragement and a suitably objective critique of numerous drafts. Lui Melechi provided reassurance that the direction was sensible.

And finally, of course, I am extremely grateful to my employer, Racal Research, for supplying the motivation, time and resources, and to Peter Blair for presenting the opportunity and then allowing the research to take its course.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

1 Introduction

C++ [C++98] is a popular, very widely used and successful industrial strength language with support for Object Orientation. The popularity of C++ is in part attributable to a very high degree of compatibility with C [C89]. Portability and run-time efficiency are some of C’s and consequently C++’s attractions to programmers.

Efficiency is achieved in C by providing programming constructs that are relatively low level. Efficiency is preserved in C++ by using a restrictive form of Object Model that enables C++ to resolve at compile-time what many other Object-Oriented languages resolve at run-time.

Programmers as well as compiler writers seek to trade compile-time for run-time activity. A programmer may improve run-time efficiency by identifying better algorithms, selecting a more efficient compiler, or structuring code to exploit the good, or avoid the poor, characteristics of a compiler. In order to improve compile-time, there appear to be few approaches, although different coding styles and appropriate management of include file dependencies and compilation unit sizes can show surprising benefits.

The inadequacies of C support for compile-time programming were recognised from the outset by the provision of Cpp (the C preprocessor). C++ has introduced more powerful language constructs but provides no assistance for programmers who need to capture predictable programming structure that does not correspond to language constructs.

Programming style, patterns and aspects

Dramatic improvements in programming time (and maintainability and reliability) may be achieved when an automatic code generator such as lex or yacc is applicable, or when an application generator such as a GUI builder is suitable. For many more mundane programming applications, the structure of the code is in some way predictable, but not of sufficient complexity to justify the development of a custom code generator. For these applications, the programmer is forced into workarounds, exploiting whatever tools are available. These workarounds often require indirect or repeated expression of the programming intent, introducing:

- maintenance problems through lack of clarity
- inefficiencies through the need for repeated editing
- errors through inconsistent repeated editing.

Cpp was for a long time the main tool available to C and C++ programmers. Prior to the introduction of templates, it was standard practice to use some very large preprocessor macros to define generic classes for containers. Templates now provide a powerful solution to problems that can be characterised by the requirement to define a family of types or functions. For many other problems, Cpp remains the only alternative. Lexical pasting using the preprocessor is inelegant and not without its problems, but it is less error prone than manual approaches.

Compatibility with C required C++ to preserve Cpp, although its limitations as a programming tool have long been recognised (and are summarised in Section 2.4). While C++ introduces a number of new constructs that eliminate some traditional uses of Cpp, other uses remained. The power and complexity of Object Orientation and the increasing use of simple patterns [Gamma95], clichés [Gil97] or idioms [Coplien92] considerably increases the need for programmers to program at compile-time and as a result Cpp is perhaps more, rather than less, important to C++ than to C.

Cpp should be replaced rather than eliminated.

The Object Oriented Programming community has recognised that groups of objects with a shared behaviour can be found within apparently dissimilar applications. These shared behaviours are classified by patterns. There is no precise definition of what constitutes a pattern, but it is generally agreed that a pattern is a solution to a recurring problem in a context. Recurrence is an important discriminant between generic patterns and candidate patterns, which may be just special purpose tricks. A pattern is used in a specific context which imposes constraints or forces that influence the way in which the pattern solves the problem The description of a pattern provides a range of solutions and a discussion of how the differing forces from the application context may influence the usage. The generality of the pattern concept allows the pattern form to be very widely applied: from project management through to idiomatic coding. This generality is a little frustrating since the literature lacks focus.

Analysis, design and coding patterns are of significance to this work. Coding patterns tend to be simple program idioms that implement standard coding practices that are not directly supported by the target language. Design patterns capture the collaborations between implemented objects. Analysis patterns similarly capture collaborations, but at a more abstract level reflecting the higher level analysis perspective. The extent to which patterns used during analysis, design or coding appear in the implementation is a rather contentious issue. Automated implementation of patterns would seem attractive but Misconception 4 of [Vlissides98] strongly cautions against this.

From the purist perspective, compromising between the generally conflicting forces unique to each application requires careful selection between a wide range of possible solutions. It is inappropriate and impossible to offer a cookbook solution to a pattern, partly because such a solution cannot offer sufficient flexibility. In many applications, more than one pattern is employed and where the patterns overlap, the solution must be adapted to share rather than duplicate the overlapping functionality.

From the practical perspective, an inferior set of proven cookbook solutions is often preferable to optimal handcrafted solutions. Programmers tend to implement solutions they are familiar with rather than those that could be more optimal. If patterns have been used as part of the analysis and design phases, it seems appropriate for these patterns to find some form of expression in the code. Otherwise, if the patterns are not expressed at all, the patterns are lost [Soukup94] and subsequent code maintenance is hampered by greater barriers to comprehension. When patterns are expressed only in the form of comments, compliance with design principles and constraints is informal; no enforcement occurs during implementation or subsequent maintenance.

The challenge is to provide cookbook solutions with sufficient parameterisation to satisfy the purists, while offering adequate efficiency and utility for practical requirements. Enthusiastic use of templates, as practised in Generative Programming [Czarnecki97] can result in highly configurable functions and to some extent types and components. However much of the required parameterisation involves appropriate selection and configuration of declarations in ways not amenable to template programming. Program configuration in C++ should occur at compile-time, but the facilities of the C preprocessor are inadequate for the task and a generally inappropriate foundation for this new programming paradigm.

Compile-time programming is necessary to configure declarations.

A problem with implementing patterns is that a pattern tends to involve more than one class, and so use of a pattern requires code to be added to more than one class. A similar but larger scale problem arises in Aspect Oriented Programming [Mens97], in which an aspect is a programming concern (such as error recovery) that cuts across more than one class. AO programs organise source code according to the programming concerns and then use a weaver to combine the disparate contributions into a complete program.

Patterns and AOP require weaving.

The C++ One Definition Rule (§3.2) mandates a single definition of each declaration, with the result that with a few exceptions (forward references, externs, typedefs, and namespaces) C++ source code must be organised to satisfy the constraint of complete declarations. It is not possible to interleave class declarations, which seems like a good property, until it is appreciated that this prevents weaving in support of AOP or even support for a multi-class pattern solution.

The One Definition Rule must be circumvented.

Meta-programming

Prior to C++, Object Orientation, as then exemplified by Smalltalk, was perceived to be inherently inefficient because of the run-time costs associated with message dispatch. C++ introduced a more restrictive Object Model that enabled most of the run-time costs to be replaced by compile-time computation. As a result Object Orientation in C++ is efficient and widely used. (More efficient implementation approaches developed to make Smalltalk costs more acceptable are now being exploited by Java).

C++ requires that the layout of objects be frozen at compile-time, and that the type of the recipient of any message is known. The layout constraint implies a single contiguous memory allocation for each object, simplifying memory management and providing member variable access by a simple indexing operation. The messaging constraint enables static and some dynamic methods to be implemented as simple function calls. The remaining dynamic methods require a virtual function that is implemented by a single indirection from a known index into a relatively small dispatch table. These are pragmatic constraints on the Object Model. Elimination of run-time object flexibility removed the need for run-time code to manipulate object structure, and for run-time objects to describe it. The meta-classes that are essential for languages such as Smalltalk were not necessary for C++ and so they are not part of the C++ language.

It has been found that some degree of self-awareness is useful to an Object Oriented program. This may involve

- a knowledge of class names for diagnostic purposes
- availability of inheritance information as Run-Time Type Information to validate dynamic casts
- object layout information to support marshalling for communication
- object layout information for persistent storage in data bases
- full class descriptions for browsers or debuggers

The first two of these needs have been addressed as C++ has evolved from ARM [Ellis90] to ANSI standard [C++98]. Applications requiring more substantial information must resort to special purpose pre- or post-processing. Reflection supports this extra processing directly as part of the compilation process.

Reflection is useful for simple applications.

Reflection is almost essential for sophisticated applications.

When a Smalltalk or CLOS program reflects upon itself, this necessarily happens at run-time, since this is when object structure is defined. Support for reflection is relatively easily provided by formalising the interface to the underlying run-time language support.

In C++, objects are defined at compile-time, and so an opportunity exists for a program to reflect upon itself at compile-time as well as, or instead of, at run-time.

If the purpose of that reflection is just to extract some information or perform some checking in a one-off fashion, it is clearly preferable for such code to execute at compile-time. This is very much in the C++ spirit of maximising run-time performance by resolving as much program structure as possible at compile-time. C++ only optimises those constructs that form part of C++. Reflection supports optimisation of user defined concepts.

If reflection is to happen continuously, then it must occur at run-time. The C++ philosophy dictates that unwanted language functionality should not impose run-time costs, so provision of run-time reflection must be cost free, when unused. The amount of run-time reflection may vary between applications, with the majority not using it at all. Some may wish to just browse data structures describing declarations. Very sophisticated applications may seek to reify[1] the different stages of message dispatch to validate argument lists or call-from access rights. The required support for run-time reflection can be achieved by using compile-time reflection to create data structures and modify code to collaborate with a run-time support environment. The degree of support can be tailored to match the requirements.

It is not entirely clear that compile-time modification of executable code is necessary for many practical applications, and so we consider only perusal and modification of declarations. Modification of code is possible provided the intended target for modification is encapsulated within a template or inline function, which can then be modified as a declaration.

1.1 Goals

The work described in this thesis grew out of a recognition that the programming effort for C++ programs increased disproportionately with the size of the program. This is not particularly unexpected. Large programs have been a consistent problem for software developers. Solutions have been found through improved languages and programming methodologies, with C++ and Object Orientation making their contributions. The concepts of patterns and Aspect Orientation offer further improvements to programming methodologies but highlight constraints imposed by C++. We therefore assess and resolve these limitations.

Declarative Redundancy

C++ exhibits significant lexical redundancy.

There is duplication between many interface and implementation declarations. This requires a duplication of editing effort and provides a limited opportunity for inconsistencies to introduce errors.

In a deep inheritance hierarchy, the same virtual function may have many implementations, again requiring repeated editing effort. This effort is most noticeable when it is necessary to change the function signature: two additional edits may be needed for each derived class to realise what is a single conceptual change. More seriously, an inconsistency between declarations within the hierarchy is not necessarily an error, although a helpful compiler may choose to flag the hazard.

Lexical redundancy should be eliminated.

Algorithmic Redundancy

Implementation of many idioms imposes a well-defined protocol that must be observed by derived classes. C++ provides no mechanism for implementation of this protocol, although in some cases use of a pure virtual function may diagnose

1. To make a thing, typically by creating an object to represent an abstract concept.

a non-implementation. Correct observance of the protocol requires implementors of derived classes to add the code manually, sometimes making use of preprocessor macros. This practice is at best tedious. It makes derivation from third party libraries unnecessarily difficult because application writers have to supply extra library support code.

Predictable code should be provided automatically.

Organisational Restrictions

Implementation of more interesting idioms and patterns requires code injection into multiple classes. This is not supported in C++. A particular solution to a pattern requires an interface and an implementation preprocessor macro for each collaborator. These macros are typically invoked from the interface and implementation files of each class. As a result a pattern involving 3 classes may require 6 macros. The 6 macro invocations that instantiate a single concept are dispersed throughout the source code.

A concept should be instantiated by a single invocation.

For some applications, it is appropriate to partition the source code according to the data structures: algorithmic code is then naturally assigned to classes. For other applications, in particular Aspect Oriented applications, the algorithmic perspective may be more important, and so all the code for one algorithm or aspect should be kept in one source module, while that for another should be in another module. In this situation the aspect cuts across the class structure. C++ requires sequential declaration of complete classes. It is not possible for partial declarations to be interleaved.

Interleaved declarations should be allowed.

FOG

These problems require revision of C++, and so in order to avoid development of a new language, or a new compiler for a modified language, revisions were implemented within a translator from extended C++ to standard C++.

The experimental translator is called Flexible Object Generator (FOG) and the extended language is FOG. The translator

- revises C++ (upward compatibly)
- rearranges source code
- synthesizes declarations
- interprets meta-programs

The most significant revision is the relaxation of the One Definition Rule to support composition of multiple declarations. This is a major semantic enhancement, but it is almost invisible syntactically. This revision opens the door to weaving and pattern implementation.

The facilities of the C preprocessor are essential for practical programming, but integrate very poorly with C or C++. Many of the FOG extensions provide replacements for Cpp functionality, thus meta-functions replace function-like macros, tree-literals invite a replacement avoiding the accidental substitutions characteristic of Cpp, and meta-statements support conditional compilation. Consistent generalisation of each of these concepts results in a compile-time environment in which meta-programs can be interpreted.

1.2 Organisation

A brief overview of FOG is provided in this chapter, so that the full exposition of the FOG grammar can provide examples unconstrained by avoiding forward referencing. The simpler semantic issues are discussed in association with the grammar in Chapter 2, with the exception of the catalogue of built-in meta-functions which may be found in Appendix E. More substantial discussions of the semantics of substitution, composition and meta-classes are deferred until Chapter 3. The changes are described one at a time in Chapter 2 interspersed with discussion. The modified grammar is therefore repeated in Appendix A for ease of comparison with Annex A of [C++98].

A context-free grammar is important for flexible meta-programming, and it is well-known that the C++ grammar is not context-free. This would appear to preclude context-free meta-programming. Chapter 4 examines existing parsing approaches, and introduces a new approach that supports context-free syntactical analysis of C++. The validity of the new approach is shown by using an extended form of regular expression to analyze the C++ grammar and deduce the ambiguities. A working yacc-able implementation of the C++ grammar alone is presented in Appendix B, with the full FOG version in Appendix C.

Operation of FOG as a translator to C++ involves a number of practical concerns regarding file locations, partitioning of declarations into files and generation of appropriate include file dependencies. Syntax extensions to resolve these issues and a description of the algorithms required for code emission are described in Chapter 5.

A number of small examples and a few slightly larger ones are presented in Chapter 6.

Some related work has already been mentioned as part of the motivation for this work. Chapter 7 contains a more extensive review and comparison. The achievements of FOG are summarised in Chapter 8, followed by a glossary of significant terms and acronyms and a list of all references.

The FOG command line is described in Appendix D.

Detailed descriptions of

- discarded syntax
- parsing ambiguity resolution
- C++ ambiguities introduced by the superset grammar approach
- FOG ambiguities introduced by the superset grammar approach

are presented in Appendix F.

1.3 Conventions

Before we provide an overview of the FOG functionality, we must define the typographical conventions used throughout the rest of the text.

The FOG grammar is a superset of C++ and so it is necessary to make occasional reference to the C++ grammar as defined by [C++98] and then summarised in its Annex A, which is very similar to Appendix A of [Stroustrup97]. In order to save the reader having to keep a copy to hand, relevant sections are included in the text and in Appendix A. Specific paragraphs are referenced as §11.4, or more commonly as §11.4-5 where the 5 denotes the numbered paragraph within the section numbered 11.4 in [C++98].

The BNF-like (Backus Naur Form) language of the C++ standard is used in this document. Terminals (such as `static`) are distinguished by the use of a typewriter font. Non-terminals (such as *parameter-declaration-clause*) are in an italicised seriphed font. A production (rule) comprises a non-terminal on its left-hand side followed by a colon followed by the right-hand side. Productions that share a common left-hand side are grouped together with one right-hand side per line. Optional elements are denoted by an $_{opt}$ suffix. Comments may be supplied following a //. Thus

base-specifier:
  `::`$_{opt}$ *nested-name-specifier*$_{opt}$ *class-name* *// defaults to* `private`
  `virtual` *access-specifier*$_{opt}$ `::`$_{opt}$ *nested-name-specifier*$_{opt}$ *class-name*
  access-specifier `virtual`$_{opt}$ `::`$_{opt}$ *nested-name-specifier*$_{opt}$ *class-name*

comprises the 3 rules that define the syntax of a *base-specifier*. (The standard is a little lax in its formatting of these productions, neglecting to use typewriter font for the `virtual` keyword or the `::` punctuation.)

In order to ease comparison of similar FOG and C++ grammar, the two are combined with a ~~strike-through~~ to denote C++ constructs removed in FOG, and an underline to denote FOG constructs added to C++. Thus the C++ rules

primary-expression:
  literal
  `this`
  `::` *identifier*
  `::` *operator-function-id*
  `::` *qualified-id*
  `(` *expression* `)`
  id-expression

and the replacement FOG rules

primary-expression:
  literal
  `this`
  `(` *expression* `)`
  declarator-id

may be shown in combination as

primary-expression:
  literal
  `this`
  ~~`::` *identifier*~~
  ~~`::` *operator-function-id*~~
  ~~`::` *qualified-id*~~
  `(` *expression* `)`
  ~~*id-expression*~~
  declarator-id

When a strike-through or underline is applied to a shared left-hand side rather than a rule, the strike-through or underline applies to all rules, but is omitted in the interests of readability.

~~*using-declaration*~~*:*
  `using` `typename`$_{opt}$ `::`$_{opt}$ *nested-name-specifier unqualified-id* `;`
  `using` `::` *unqualified-id* `;`

Application of a strike-through to the left-hand side (definition) non-terminal implies application of a strike-through to all references of the non-terminal as well.

C++ grammar productions are generalised in FOG, but are not given different meanings. There is therefore no ambiguity in referring to the grammar production for a declaration as *declaration*. Italics in normal text denote a non-terminal.

When it is necessary to show examples of grammar implementation rather than specification, typewriter font is used throughout. Non-parametric terminals are spelled out in single or double quotes (`'*'` or `"class"`). Parametric terminals are shown in mixed-case (`StringLiteral`). Non-terminals are shown in lower case (`base_specifier`).

```
base_specifier:
      "::".opt nested_name_specifier.opt class_name
  |   "virtual" access_specifier.opt "::".opt nested_name_specifier.opt class_name
  |   access_specifier "virtual".opt "::".opt nested_name_specifier.opt class_name
```

Multi-character terminals such as `"::"` or `"virtual"` are a non-standard extension and are not supported by yacc or bison. They are used in the main text for clarity. They are not used in the grammars in Appendix B or Appendix C where an upper-case lexical token such as `SCOPE` or `VIRTUAL` is used.

1.4 FOG Extensions

The FOG grammar is a superset of the C++ grammar, with extensions to support compile-time programming. FOG is a translator from extended C++ to standard C++. The extensions are sufficient to render the C preprocessor redundant, although Cpp continues to be supported for compatibility.

Compile-time programming is often referred to as meta-programming, and data structures describing data are often referred to as meta-data. In particular, classes are described by meta-classes. We therefore refer to FOG as a meta-compiler, and generally use the meta- prefix to refer to a conventional run-time C++ concept elevated to compile-time, or more precisely the new meta-compile-time that precedes conventional compile-time, although it is often convenient to loosely refer to meta-compile-time and compile-time together as just compile-time.

Thus meta-classes describe classes, and meta-inherit from meta-bases: classes which are in turn the meta-classes of the corresponding base classes. A meta-class has meta-members which are meta-functions and meta-variables with static and non-static variants. Statements that are interpreted at meta-compile-time are meta-statements. Declarations for use at meta-compile-time are meta-declarations. The type system available for use at meta-compile-time comprises meta-types.

Meta-compilation involves a two stage translation from FOG source text declarations to C++ declarations suitable for emission. Source declarations are first converted into potential declarations (or specifiers) at which point the eventual scope into which the declaration contributes may be undetermined. Once the scope is determined, the potential declaration is installed as an actual declaration of the determined scope.

Meta-programs run within the compiler, and so it is more accurate to say that the meta-program is interpreted rather than executed. Of course a sophisticated meta-compiler could compile the meta-program and then (meta-)execute it. Meta-programming may operate on potential or actual declarations.

1.4.1 Substitution

FOG introduces a new syntax-based substitution mechanism for meta-variables and meta-functions to support algorithmic contributions to declarations. Substitution is invited rather than imposed as is the case for the C preprocessor.

Early substitution occurs when the source text is first analysed to create potential declarations. Early substitution is invited by the $ operator.

Late substitution occurs when the potential declarations are resolved to actual declarations. Late substitution is invited by the @ operator.

Each substitution operator may be used as

$*tree-expression*	@*tree-expression*
${*tree-expression*}	@{*tree-expression*}

Evaluation of the *tree-expression* yields a replacement in the form of an Abstract Syntax Tree (AST) that is parsed independently of the surrounding tokens. A limited number of expression operators are supported and some useful built-in meta-functions and meta-variables are defined. A *tree-expression* is defined to be the longest possible sequence of tokens. The braced form of invocation must be used when this disambiguation conflicts with the programming intent.

```
typedef @Super Super;              // See Section 6.1.1
$std::error("Illegal condition");
bool isVirtual = $bases[0].is_virtual();
```

```
        class ${ClassName}Null : public $ClassName {};
```

The syntax of $ and @ expressions is defined in Section 2.1.1.6, and the semantics in Section 3.2.

The built-in meta-variables such as Super are defined in Appendix E.3.

The built-in std meta-namespace is defined in Appendix E.2.1.

The built-in meta-functions (such as bases) of the built-in meta-types are defined in Appendix E.4.

Concatenation is described next in overview and then defined in Section 2.1.1.

1.4.2 Concatenation

FOG generalises the ANSI C concatenation of adjacent string-literals to support concatenation of sequences of *character-literal*s, identifiers (including reserved words), *number-literal*s, *string-literal*s and *tree-literal*s without intervening whitespace.

This provides a direct replacement for the rather indirect behaviour of the Cpp ## operator. Defining the concatenation to have the same category as the first element in the sequence enables the Cpp # operator to be realised by concatenation with an empty string-literal.

```
    const char *class_name() const { return ""@Scope; }
```

The above example uses an empty string to cast the built-in meta-variable Scope from an identifier to a string-literal suitable for use as part of a simple Run-Time Type Information (RTTI) system. This example is elaborated upon in Section 6.3.3.

This is a potentially quite dramatic change to the lexical conventions. The surprisingly small incompatibilities that result are discussed in Section 2.3.4.

1.4.3 Composition

FOG removes the major restraint imposed by the C++ One Definition Rule (§3.2), and allows declarations to be organised to suit the programmer, rather than the compiler. It is not necessary for classes to be defined in one place. Contributions to classes, enumerations, arrays or functions may occur in many places. FOG gathers all the contributions together to form the composite whole that is then passed on as legal C++.

A very deliberate corollary of this is the elimination of the need for distinct interface and implementation declarations. FOG generalises C++ syntax to support either an interface or implementation style of declaration. Programmers may therefore use an implementation style of declaration for parts of interfaces

```
    public typedef size_t Class::SizeType;
```

or provide complete implementations in interfaces:

```
    class Class
    {
        protected virtual void f(int x = 0) = 0 { std::cout << x; }
    public:
        static double y = 0;
    };
```

This apparently trivial change has the two major consequences

- interface/implementation redundancy is eliminated
- classes can be extended

The semantics of composition is described in Section 3.4.

1.4.4 Meta-scopes

C++ namespaces and classes have name-spaces in which declarations may be encapsulated for use by members of the namespace or class. C++ functions and variables have only a single occurrence and so reuse of a function or variable-specific context does not arise. Multiple occurrences of the same function or variable declaration are composed in FOG and so a shared context is desirable and provided for use at meta-compile-time by a meta-object scope.

```
int _variable = 5
:{                                  // Start of meta-object scope
    /* meta-object-statements */
};                                  // End of meta-object scope
```

`:{` and `}` follow a variable or function declarator to delimit the scope. Within the scope shared declarations and other new annotation syntax may be provided.

1.4.5 Derivation Rules

There are many idioms that require entirely predictable code to be provided by derived classes in order to comply with a protocol defined by a base class. In C++, a declaration applies to the scope for which it is specified. In FOG, this scope is referred to as the root scope for that declaration. An optional derivation rule specifies how that declaration may be automatically redefined in the inheritance tree of scopes that derive from the root scope. Refining the earlier example

```
auto declaration ClassName()
{
    public virtual !inline const char *class_name() const
    :{
        derived(true) { return ""@Scope; }
    };
};
```

The `:{ ... }` construct provides a meta-scope for a function in which meta-declarations can be provided and function bodies annotated. The `derived(true)` annotation causes the function body and consequently the function declaration to be applied to all classes in the inheritance tree, for which the predicate argument evaluates true. In this case, this is at the root scope and all derived scopes.

1.4.6 Meta-Programming

FOG supports static meta-programming, that is the execution of user supplied code at meta-compile-time. This code may analyze and modify declarations in order to ensure compliance with some constraint.

This may just involve monitoring declarations to ensure that some policy is observed. For instance a meta-program could verify the presence of a virtual destructor under all or selected conditions.

More powerfully, a meta-program may be used to generate code automatically to support run-time access to compile-time information. The generated code can be optimised to suit user requirements:

- a very modest form comparable to RTTI for simple debugging
- a more extensive set of tables or functions to support persistence or marshalling
- a very substantial set of tables to support full run-time meta-programming supported by a corresponding run-time executive

The facilities provided by FOG are very much focused on meta-programming as a manipulation of program declarations. This contrasts with but complements the manipulation of types, functions and expressions using template meta-programming.

1.5 FOG versions

It was originally thought that the goals of FOG could be achieved by relatively simple processing at a lexical level. This proved to be naive. Useful manipulation of C++ declarations requires accurate parsing of those declarations and therefore a fully fledged C++ parser is needed. C++ is context dependent, but flexible meta-programming requires source declarations to be interpreted before their context is known. A new approach is therefore introduced so that FOG can perform context-free syntactical analysis of C++. The evolution to the current implementation is described in Chapter 4.

Superset Implementation

The description of the FOG meta-compiler in this thesis applies mainly to the current incomplete implementation that uses a superset grammar to support context-free parsing of FOG (and C++) source. The lexical and syntactical analyses of this meta-compiler are complete. The semantic analysis and subsequent stages are incomplete, but sufficient to demonstrate the soundness of the approach using simple examples.

In some areas, the examination and resolution of design decisions goes some way beyond the implementation.

Multi-pass Implementation

The previous implementation resolved syntactical ambiguities using multiple passes. Ambiguities in meta-constructs were avoided by using a syntax-driven and consequently context-dependent method. This implementation was more complete, but suffered from deficiencies inherent in the syntax-driven method.

FOG and associated publications are available from the FOG home page:

http://www.computing.surrey.ac.uk/research/dsrg/fog

2 FOG Grammar

Section 1.4 provided a brief overview of the FOG extensions to C+ involving

- substitution
- concatenation
- composition
- meta-scopes
- derivation rules
- meta-programming

In this chapter the language extensions are presented in detail and the semantics specific to each extension is discussed. More general semantic issues are discussed in Chapter 3. Description of the necessary but rather peripheral extensions required to support multiple intermediate files between the FOG translator and the C++ compiler is deferred till Chapter 5.

The changes are presented one at a time in this chapter. The summary of the changed grammar in Appendix A shows all changes combined and is structured to ease comparison with Annex A of [C++98].

2.1 Grammar Extensions

Most of the FOG extensions contribute extra grammar, however the substitution and concatenation functionality is white-space sensitive and so must be performed earlier. This occurs during phase 6 of the C++ translation process.

2.1.1 Substitution, Concatenation and Tokenization

The C++ standard defines (§2.1-6) translation phase 6 as:

> Adjacent ordinary string literals are concatenated. Adjacent wide string literals are concatenated.

and (§2.1-7) phase 7 as:

> White-space characters separating tokens are no longer significant. Each preprocessing token is converted to a token. The resulting tokens are syntactically and semantically analyzed and translated.

FOG generalises phase 6 processing to support more extensive concatenation and also the recognition of substitution operators. The parsed result of a substitution or non-trivial concatenation is referred to as a *tree-literal*, since it comprises a pre-parsed AST.

FOG generalises phase 7 to support tokenization of a *tree-literal* as an *identifier*, and to defer treatment of non-reserved words as *identifier*s.

The enhanced behaviour of FOG is most easily explained by first elaborating the simple C++ descriptions into grammars before showing the revised grammars.

2.1.1.1 C++ Phase 6 Concatenation Grammar

The C++ Phase 6 translations can be expressed as

string-literal$_{cat}$:
    string-literal$_{pp}$
    string-literal$_{cat}$ *whitespace*$_{opt}$ *string-literal*$_{pp}$

‘anything-else’$_{cat}$:
    ‘anything-else’$_{pp}$

$_{pp}$ denotes the preprocessor token input to phase 6 from phase 5 and $_{cat}$ the concatenated output production passed from phase 6 to phase 7.

Single quotes as in ‘*anything-else*’ surround a production whose meaning is obvious though difficult to express compactly.

2.1.1.2 C++ Phase 7 Tokenization Grammar

Phase 7 tokenization can be expressed by the following ‘grammar’

‘discard’: — *// Token is discarded*
    whitespace

‘punctuation’: — *// e.g.* `,` *or* `+=` *or* `...`
    ‘punctuation’$_{cat}$

‘reserved-word’: — *// e.g.* `true` *or* `unsigned` *or* `if`
    identifier$_{cat}$ — *// If identifier*$_{cat}$ *is a reserved word*

character-literal:
    character-literal$_{cat}$

floating-literal:
    number-literal$_{cat}$ — *// If number-literal*$_{cat}$ *is floating point*

integer-literal:
    number-literal$_{cat}$ — *// If number-literal*$_{cat}$ *is fixed point*

string-literal:
    string-literal$_{cat}$

identifier:
    identifier$_{cat}$ — *// If identifier*$_{cat}$ *is anything else*

2.1.1.3 FOG Phase 6 Concatenation Grammar

FOG replaces phase 6 by a white-space sensitive grammar to augment the C++

- concatenation of adjacent string-literals

by

- concatenation of unseparated character-literals
- concatenation of unseparated identifiers
- concatenation of unseparated number-literals
- concatenation of unseparated tree-literals

As in C++, the distinction between ordinary and wide-string literals can be ignored, since their adjacency leads to undefined behaviour (§2.13.4-3). Behaviour is only defined for a sequence of same-width *string-literal*s or *character-literal*s.

text-literal$_{pp}$*:*
    character-literal$_{pp}$
    identifier$_{pp}$ — *// Including all reserved words*
    number-literal$_{pp}$
    string-literal$_{pp}$
    tree-literal$_{pp}$ — *// a* $ *or* @ *expression*

character-literal$_{cat}$*:*
    character-literal$_{pp}$
    character-literal$_{cat}$ *text-literal*$_{pp}$

identifier$_{cat}$*:*
    identifier$_{pp}$
    identifier$_{cat}$ *text-literal*$_{pp}$

number-literal$_{cat}$*:*
    number-literal$_{pp}$
    number-literal$_{cat}$ *text-literal*$_{pp}$

$\textit{string-literal}_{cat}$:
  $\textit{string-literal}_{pp}$
  $\underline{\textit{string-literal}_{cat}\ \textit{text-literal}_{pp}}$
  $\textit{string-literal}_{cat}\ \textit{whitespace}_{opt}\ \textit{string-literal}_{pp}$

$\underline{\textit{tree-literal}_{cat}}$:
  $\textit{tree-literal}_{pp}$
  $\textit{tree-literal}_{cat}\ \textit{text-literal}_{pp}$

$\textit{'anything-else'}_{cat}$:
  $\textit{'anything-else'}_{pp}$

Translation of source tokens involves three significant textual representations[1]:

- the original source code spelling

Any escape sequences and digraphs in this representation are replaced during translation phase 5 to give

- an internal textual (multi-)byte sequence

that comprises straightforward binary encoding of each character. This representation is in turn converted during the emission phase to

- a representation suitable for output

which may require regeneration of escape sequences. This regeneration propagates the original source spelling when the output corresponds directly (no concatenation) to a source token.

Concatenation operates by concatenation of internal sequences without regard to the *character-literal*, *identifier*, *number-literal* or *string-literal* categorisation.

The textual byte sequence of a numeric value is the source spelling, if the value originated from source. Otherwise the textual sequence is generated from the numeric value using a numeric to ASCII conversion. The default formatting is specified only to require precision not less than `long` for an *integer-literal* or `float` for a *floating-literal*. Specific built-in meta-functions may be used for more precisely controlled formatting.

The textual byte sequence of a *tree-literal* comprises the byte sequence of the main unqualified name in the *tree-literal*, such as the *class-name* of a *class-specifier*:

```
auto class_specifier c = class Scope::Class {};
auto identifier i = $c;      // i = Class
```

Preservation of unchanged format and conversion between *identifier* and *string-literal* perspectives provides most of the functionality of Cpp # operator stringizing. Generalisation to *character-literal*s and *number-literal*s provides consistency rather than significant new functionality.

It is surprising that this fundamental lexical extension causes only very minor incompatibilities, mandating protective spaces where only a perverse coding style would omit them:

- around the *string-literal* in a *linkage-specification*: `extern "C"` *declaration*
- around alternative tokens: and, and_eq, `bitand`, `bitor`, ..., `xor`, `xor_eq`
- following *expression* prefixes: `return`, `sizeof`, `throw`

This is described in Section 2.3.4.

There is no incompatibility regarding the `L` prefix for wide-characters or wide-strings since the `L"` and `"` are stripped during phase 3. Phase 6 processes the contents using some form of parallel rather than prefix annotation to signify a wide-string.

1. There is a momentary fourth true source code spelling before trigraphs are replaced in translation phase 1.

2.1.1.4 FOG Phase 7 Tokenization Grammar

The concatenated preprocessor tokens are tokenized by the following ‘grammar’

‘discard’:	*// Token is discarded*
whitespace	
‘reserved-word’:	
identifier$_{cat}$	*// If identifier*$_{cat}$ *is a reserved word*
‘punctuation’:	
‘punctuation’$_{pp}$	
character-literal:	
character-literal$_{cat}$	
floating-literal:	
number-literal$_{cat}$	*// If number-literal*$_{cat}$ *is floating point*
integer-literal:	
number-literal$_{cat}$	*// If number-literal*$_{cat}$ *is fixed point*
string-literal:	
string-literal$_{cat}$	
meta-type-name:	*// If identifier*$_{cat}$ *is a meta-type name*
identifier$_{cat}$	*// (and not a reserved word)*
‘non-reserved-word’:	
identifier$_{cat}$	*// If identifier*$_{cat}$ *is a non-reserved word*
~~identifier~~:	
other-identifier:	
identifier$_{cat}$	*// If identifier*$_{cat}$ *is anything else*
tree-literal:	
tree-literal$_{cat}$	

Identifiers are categorised as one of *reserved-word*, *meta-type-name* (name of a built-in meta-type), *non-reserved-word* (word used in a FOG syntax extension) or *other-identifier*. The latter three and *tree-literal* are combined in the main grammar for uniform treatment where an *identifier* is expected.

identifier:
  other-identifier
  meta-type-name
  non-reserved-word
  tree-literal

The results of concatenation in phase 6 may not yield consistent tokens. For instance:

```
auto identifier plus = '+';
x ${plus}${plus};
```

could be perceived as

```
x ++;
```

If the tokenization grammar recognises that the identifier ++ can be tokenized as the corresponding punctuation token, then the example comprises an *expression-statement* that increments x.

The multi-pass grammar FOG implementation took this approach, reclassifying the byte sequences masquerading as identifiers as reserved-words, punctuation, *number-literal*s or *identifier*s. However *tree-literal*s are not necessarily resolvable during phase 6 without invoking a very tight coupling with syntactic and semantic analysis.

Retokenization of identifiers is no longer performed and so the only change in FOG phase 7 processing is the propagation without interpretation of *tree-literal*s to the syntactic analysis, where they are treated, again without interpretation, as *identifier*s.

The processing is therefore context-free and the example is interpreted as

- *identifier* with value `x`
- *identifier* with value `++`
- the punctuation `;`

This is a syntactically valid definition of a global variable named `++` of type `x`. It must be rejected during semantic analysis.

Retokenization can be achieved by using the `std::parse` built-in meta-function, using which, the increment behaviour may be obtained by:

```
$std::parse("x "${plus}${plus}";");
```

Whether trivial concatenations are resolved during phase 6 is an implementation option. However, certain complicated concatenations involving substitutions can only be resolved during semantic analysis.

2.1.1.5 *character-literal*

In C++, a *character-literal* should comprise a single character, whereas a *string-literal* may contain any number of characters.

In order to support concatenation consistently, FOG generalises *character-literal*s to treat them equivalently to *string-literal*s during lexical and syntactic analysis. A *character-literal* may have any number of characters, however when a *character-literal* with other than one character is used (as a *literal* in a *primary-expression*), the behaviour is undefined.

This change supports concatenation consistently, but offers only minor functionality enhancements: an empty character can be used as a concatenation join or as a concatenation cast.

```
auto for (int i = 0; i < 10; ++i)
{
    static char digit_values[] = char($i);  // 0 to 9
    static char digit_codes[] = ''$i;       // '0' to '9'
}
```

2.1.1.6 FOG *tree-literal*s

Tree-literals support the use of a parsed AST from a meta-variable or meta-function.

Definition occurs using the meta-function or meta-variable declarations described in Section 2.1.5.6 and Section 2.1.5.5.

```
auto enum_specifier enumTree = enum f { F };
```

Access occurs through use of a `$` or `@` expression to provide a *tree-literal*$_{pp}$ token.

```
typedef $enumTree E;
```

This syntactic usage exploits the treatment of a pre-parsed AST *tree-literal* as an *identifier* by the main FOG grammar as described in Section 3.2.3.

```
tree-literal_pp:
    at-literal
    dollar-literal
    syntax-macro                    // Section 2.1.6 and Section 3.7

at-literal:                         // Loosely referred to as an @-expression
    @ tree-expression
    @ { tree-expression }

dollar-literal:                     // Loosely referred to as a $-expression
    $ tree-expression
    $ { tree-expression }
    $ dollar-literal
```

The first two forms of *tree-literal* are triggered by a $ or @ introducer with an optional pair of braces to surround the actual expression. Syntax ambiguities are avoided in the absence of braces by defining the lexical expression as the longest possible token sequence. The semantics of syntax-level substitution is described in Section 3.2. The name-resolution rules are described in Section 3.3, and the significance of repeated $'s in Sections 3.3.4 and 3.3.5.

The third form of *tree-literal* supports a user-defined syntax-macro comprising a triggering *identifier* and relatively arbitrary subsequent syntax. Syntax macros are described in Section 3.7.

Analysis of the *tree-literal* invocation sequence is initiated by recognition of the trigger token in the white-space sensitive phase 6 concatenation grammar. This activates a nested whitespace insensitive syntactic analysis to identify the syntax tree to be returned for use by the interrupted syntactic analysis. This syntax tree describes the invocation, not the resolution, since the invocation may not be directly or uniquely resolvable. The nested analysis to identify the *tree-literal* invocation obtains tokens from the same source as, and in the same way as, each token of the interrupted analysis. Nested recognition of *tree-literal*s is therefore supported with `$a$b$c` resolved as `${a${b${c}}}`.

```
primary-tree-expression:
      meta-scoped-id                                          // Section 2.1.5.1
      ( tree-expression )

postfix-tree-expression:
      primary-tree-expression
      postfix-tree-expression ( tree-argument-list_opt )      // meta-function call
      postfix-tree-expression [ expression ]                  // meta-array index
      postfix-tree-expression . scoped-id                     // meta-member selection
      postfix-tree-expression -> scoped-id                    // meta-member selection via iterator

tree-expression:
      postfix-tree-expression
      * tree-expression                                       // iterator indirection
```

meta-scoped-id is the optionally scoped name of a meta-object (in the meta-name-space). A meta-function-call invokes a user-defined or built-in meta-function, globally or on a selected meta-object. Array indexing selects a list element. Member-selection selects a named member of a meta-object. Indirect member selection and indirection apply only to meta-objects of `iterator` meta-type, since these are the only form of meta-pointer.

```
tree-argument-list:
      tree-argument
      tree-argument-list , tree-argument

tree-argument:
      tree-statement
      unterminated-tree-argument

tree-statement:
      terminated-tree-argument
      unterminated-tree-argument ;

compound-tree-statement:
      { tree-statement-seq_opt }

tree-statement-seq:
      tree-statement
      tree-statement-seq tree-statement
```

terminated-tree-argument:
asm-definition
control-statement
declaration-statement
explicit-instantiation
explicit-specialization
expression-statement
file-dependency-declaration
file-placement-declaration
filespace-declaration
include-declaration
linkage-specification
namespace-alias-definition
namespace-declaration
template-declaration
using-declaration
using-directive
`auto` *meta-class-declaration*
`auto` *meta-control-declaration*
`auto` *meta-expression-statement*
`auto` *meta-variable-declaration*

unterminated-tree-argument:
access-specifier
accessibility-specifier
base-specifier
built-in-type-id
class-specifier
compound-tree-statement
condition
cv-qualifier
decl-specifier
enum-specifier
enumerator-definition
expression
filespace-specifier
function-definition
function-try-block
handler-seq
initializer-clause
mem-initializer
namespace-definition
parameter-declaration
reserved-word
simple-type-parameter
storage-class-specifier
template-argument
template-parameter
type-parameter
`auto` *meta-class-specifier*
`auto` *meta-function-definition*

The possible syntaxes are split into terminated and unterminated categories according to whether a following semicolon would be out of place

```
if (a) f();    ;   // Extra ; out of place after selection-statement
{ f(); }       ;   // Extra ; acceptable after compound-statement
```

When used as a *tree-argument* in the comma-separated *tree-argument-list* the extra semicolon is optional for the unterminated productions.

```
f(int a;, int b, c, d;);
```

When used as a *tree-statement* as the initializer of a meta-variable the semicolon is mandatory for the unterminated productions.

```
auto statement s = { a(); b(); c(); };
auto declaration d = int a;
auto identifier i = if;
```

The trailing semicolon satisfies syntactic requirements, but has no semantic meaning. There is therefore no semantic difference between the *expression*:

```
a
```

and the *expression-statement*:

```
    a ;
```

They may be used interchangeably as *tree-argument*s, and subsequently as the values of a `statement`, `expression`, `name` or `identifier` meta-type (amongst others).

There is an ambiguity between multiple comma-separated components of a specific *tree-argument* and multiple *tree-argument*s.

```
    $f(int a, b, c, if (d) e, f; else g, h;)
```

comprises four arguments, the fourth of which is obviously a *selection-statement*, however the earlier three could be an *init-declaration-list*, a *parameter-declaration-clause*, or perhaps an *exception-specification* followed by a two element *expression*. The ambiguity is not resolved until the actual arguments are associated with formal parameters. When semantic interpretation occurs, the declaration of `f` may be found to be:

```
    auto statement f(parameter_declaration p1,
                     identifier p2[],
                     statement p3)
```

The available arguments are then associated from left to right, associating as many arguments as possible with one parameter before advancing to the next parameter. `int a` is associated with the *parameter-declaration* `p1`. The `p2` formal parameter can associate with an exposed list of *identifier*s (see Section 3.1.5). The second and third arguments are syntactically valid *identifier*s and so the list comprising `b` and `c` is associated as the list for `p2`. Finally the fourth argument is associated with `p3`.

The ambiguity resolution resulting from unconditionally associating arguments with the left-most parameter is consistent with similar disambiguation policies such as §5.3.4-2 for trailing `*`'s in *new-declarator*s.

In practice, meta-functions with argument lists that comprise an exposed list, followed by further arguments, should be avoided since left to right association may prevent subsequent arguments being passed. For instance, it is not possible to pass an *identifier* representing a simple *expression-statement* as the fourth argument to `f` above since the *identifier* would be associated with the preceding exposed list parameter. It is not possible to solve the problem by introducing parentheses or braces, since this would further overload existing punctuation and introduce ambiguities with respect to their existing meaning.

The productions listed for *terminated-tree-argument* and *unterminated-tree-argument* cover most of the C++ constructs and exhibit significant redundancy. In practice it is more helpful to identify what is not covered, why it is not covered, and how the limitation can be worked around:

- an anonymous bit-field cannot be specified because a *labeled-statement* is unconditionally preferred
  - give the bit-field a name or
  - provide an *access-specifier* or
  - use a `class` or `typename` prefix for the type
- do followed by semicolon is presumed to be the start of an iteration statement, so do as a reserved-word must not be followed by semicolon
  - omit the semicolon in a *tree-argument*
  - use "do" and implicit (or explicit) string to identifier conversion
- `operator` followed by a comma is presumed to be the sequencing function name, so `operator` as a reserved-word cannot be followed by a list-separating comma
  - use `operator;,`

- meta-declarations in *tree-argument*s cannot be specified without a leading `auto` to avoid ambiguity with conventional declarations and statements.
    - use `auto` prefix for all meta-declarations.
- major punctuation defines lexical structure and so cannot be passed as a punctuation argument. This affects: `{ } , ; " ' ( ) #`
    - use `$std::parse("{")`

2.1.1.7 Design Rationale

Lists and Trees

Support for list (or more strictly tree structured) arguments as described above is convenient and the disambiguation rule solves some inelegant practical problems, at the expense of being a little cute.

One simpler alternative policy identifies maximal length sub-list elements unconditionally without reference to the invoking context. This unfortunately requires additional semicolons to prevent unwanted grouping:

```
$f(int a;, b, c;, if (d) e, f; else g, h;)
```

and more unacceptably requires three identifiers to be passed as:

```
$g(a;, b;, c)
```

Another alternative policy prohibits transparent passing of list elements, which is obviously less flexible. `()`’s or `{}`’s must then be used to encapsulate the lists for those syntaxes where bracketing is permitted. A syntax extension is needed for other syntaxes.

All three policies are difficult to implement in practice because a generalised superset grammar needs to treat a potential constructor argument list

```
 a : b(c), d(e)
```

as an unresolved prefix that could be a bit-field followed by a variable, or a constructor and initializers. The solution in Appendix C exploits the generalised monomorphic characteristic of each argument to avoid premature parsing decisions.

Comma-separated lists of arguments that may themselves comprise comma-separated lists are therefore supported, since support imposes fewer programming constraints and no extra implementation difficulty.

```
auto declaration f(parameter_declaration_clause p);
```

can be invoked as:

```
$f(int a, char *b, ...);    // Ellipsis token as third argument
```

Sequences

Sequences (unseparated lists) of arguments are not supported to avoid a syntax ambiguity between a sequence and further syntax as the initializer of a meta-variable list:

```
 auto statement meta_variable[] = stmt1; stmt2; stmt3;
```

This is not a limitation since a sequence can be expressed in its compound form.

```
 auto statement meta_variable[] = { stmt1; stmt2; }; stmt3;
```

Labels

The goto form of a *labeled-statement*:

`label :` *statement*

and an anonymous bit-field

```
type : 5 ;
```

are ambiguous and have dissimilar syntax tree structure; a label decorates a statement, whereas a bit-field width decorates a declarator. It does not seem worth significant effort to unify such rare constructs. The label form is not incorporated in a *meta-control-declaration* to avoid changing the semantics of a *member-declaration*. However, when a bit-field is added to an *init-declarator* and consequently to a *statement*, the syntactic conflict must be resolved to the label. The same conflict arises for a *tree-argument* and so the same unconditional resolution of *identifier* `:` as a label is made.

2.1.2 Names

FOG syntactic analysis has to be context-free while operating on both potential and actual declarations, since it is inevitable that the context cannot be known for potential declarations, for which complete identification of all enclosing scopes is missing. How C++ can be lexically and syntactically analysed without knowledge of template or type name information is shown when the superset grammar is described in Chapter 4.

Since FOG does not use template or type name information, it is necessary to generalise the syntax of names to eliminate

~~typedef-name~~:
    identifier

~~namespace-name~~:
    original-namespace-name
    namespace-alias

~~original-namespace-name~~:
    identifier

~~namespace-alias~~:
    identifier

~~class-name~~:
    identifier
    template-id

~~enum-name~~:
    identifier

~~template-name~~:
    identifier

~~template-id~~:
    template-name < *template-argument-list* >

and use the much simpler concepts of

id:
    identifier
    identifier < *template-argument-list* >   *//* [2]
    `template` *identifier* < *template-argument-list* >

nested-id:
    id
    id `::` *nested-id*

scoped-id:
    `::`$_{opt}$ *nested-id*

at all times. Straightforward conversion to a context-free grammar replaces *xxx-name* by *id*. A further syntax generalisation permitting declarations to appear as interface style declarations inside class braces or implementation style declaration outside, requires that *scoped-id* be used instead.

2. Resolution of the context-dependency for < is discussed in Section 4.8.

In addition to the regular names based upon identifiers and templates, there are the special function names that involve punctuation characters:

special-function-id:
    ~ *id* // [3]
    conversion-function-id
    operator-function-id

nested-special-function-id:
    special-function-id
    id `::` *nested-function-special-id*

scoped-special-function-id:
    `::`$_{opt}$ *nested-special-function-id*

These new non-terminals replace the existing names for declarators, expressions and declarations:

declarator-id:
    ~~`::`$_{opt}$ *id-expression*~~
    ~~`::`$_{opt}$ *nested-name-specifier*$_{opt}$ *type-name*~~
    scoped-id
    scoped-special-function-id

primary-expression:
    literal
    `this`
    ~~`::` *identifier*~~
    ~~`::` *operator-function-id*~~
    ~~`::` *qualified-id*~~
    `(` *expression* `)`
    ~~*id-expression*~~
    declarator-id

using-declaration:
    `using` `typename`$_{opt}$ ~~`::`$_{opt}$ *nested-name-specifier unqualified-id*~~ *declarator-id* `;`
    ~~`using` `::` *unqualified-id* `;`~~

Type specifiers are simplified:

simple-type-specifier:
    ~~`::`$_{opt}$ *nested-name-specifier*$_{opt}$ *type-name*~~ *scoped-id*
    `char`
    ...

enum-specifier:
    `enum` ~~*identifier*$_{opt}$~~ *scoped-id*$_{opt}$ `{` *enumerator-list*$_{opt}$ `}`

elaborated-type-specifier:
    class-key ~~`::`$_{opt}$ *nested-name-specifier*$_{opt}$ *identifier*~~ *scoped-id*
    `enum` ~~`::`$_{opt}$ *nested-name-specifier*$_{opt}$ *identifier*~~ *scoped-id*
    `typename` ~~`::`$_{opt}$ *nested-name-specifier*$_{opt}$ *identifier*~~ *scoped-id*
    ~~`typename` `::`$_{opt}$ *nested-name-specifier*$_{opt}$ *identifier* `<` *template-argument-list* `>`~~

class-head:
    class-key ~~*identifier*$_{opt}$~~ *scoped-id*$_{opt}$ *base-clause*$_{opt}$
    ~~*class-key nested-name-specifier identifier base-clause*$_{opt}$~~

base-specifier:
    ~~`::`$_{opt}$ *nested-name-specifier*$_{opt}$ *class-name*~~ *scoped-id*
    ...

Namespace definitions are simplified dramatically after eliminating semantic distinctions from the syntactic grammar:

namespace-definition:
    ~~*named-namespace-definition*~~
    ~~*unnamed-namespace-definition*~~
    `namespace` *scoped-id*$_{opt}$ `{` *namespace-body* `}`

3. An unqualified destructor is ambiguous with respect to a complement expression, and very rarely valid. In practice it is easier to exclude the local destructor from *special-function-id* and introduce it only once nested in *nested-special-function-id*.

~~named-namespace-definition~~:
  original-namespace-definition
  extension-namespace-definition

~~original-namespace-definition~~:
  `namespace` *identifier* { *namespace-body* }

~~extension-namespace-definition~~:
  `namespace` *original-namespace-name* { *namespace-body* }

~~unnamed-namespace-definition~~:
  `namespace` { *namespace-body* }

Impact

The simplified naming defers semantic constraints so that the grammar defines just syntax. As a result, the grammar covers much that is illegal; in particular, every occurrence of a templated-name requires semantic validation and very occasionally correction as well.

2.1.3 Syntax Generalisation

The most significant language change in FOG is the relaxation of the One Definition Rule (§3.2) described in Section 3.4, so that multiple declarations compose to give an extended declaration rather than an error. This change is entirely semantic. A further generalisation is the unification of syntax to combine interface and implementation declarations. This is almost entirely semantic, since the C++ grammar for functions and variables embraces most of the required FOG generalisations within syntax that is only semantically invalid in C++. Some minor syntax generalisations to resolve anomalies are described in this section. Some slightly more significant enhancements appear in the next section.

2.1.3.1 Forward declaration for namespace

A FOG declaration may first appear as:

```
int Scope::name = 5;
```

`name` is added to `Scope`, which must be a previously declared namespace or class. A prior declaration of a class can be provided by

```
class Scope;
```

A similar declaration for a namespace should be possible. So FOG adds:

declaration: *// Extension of*
  namespace-declaration

namespace-declaration:
  `namespace` *scoped-id* ;

Impact

This is an unnecessary extension. It just provides consistency without introducing any problems. A namespace could alternatively be forward declared by:

```
namespace Scope {}
```

2.1.3.2 *access-specifier*s as *decl-specifier*s

The `public`/`protected`/`private` accessibility of a member declaration is necessarily specified in C++ by a preceding and relatively independent *access-specifier*. Other characteristics such as `inline` or `static` are specified by *decl-specifier*s.

```
class MyClass
{
protected:
    inline static void protected_method();
};
```

In order to avoid declarations relying on surrounding context and causing indeterminacies during composition, FOG allows the *access-specifier* to be specified as part of a declaration.

decl-specifier: *// Extension of*
    <u>*access-specifier*</u>

The presence of an *access-specifier* as part of a *decl-specifier* affects only the specified declaration. Subsequent declarations continue to use the prevailing default accessibility.

```
class MyClass
{
protected:
    public virtual void public_method();
    inline static void protected_method();
};
private void MyClass::private_method();
```

Impact

This generalisation introduces a syntax ambiguity whereby `public:` could introduce an implicitly `int` anonymous bit-field. There is no such thing and the ambiguity is resolved.

2.1.3.3 Pure-virtual

The strange C++ prohibition on joint declaration and implementation of a pure-virtual function is removed, in order to allow a complete function definition in one FOG declaration.

function-definition:
    decl-specifier-seq$_{opt}$ *declarator* <u>*pure-specifier*$_{opt}$</u> *ctor-initializer*$_{opt}$ *function-body*
    decl-specifier-seq$_{opt}$ *declarator* <u>*pure-specifier*$_{opt}$</u> *function-try-block*

pure-specifier:
    `= 0`

The following is invalid C++, but valid FOG.

```
class MyClass
{
    virtual int f() = 0 { return 0; }
};
```

Impact

The extra term poses no additional problem to the generalised superset grammar. It could pose significant problems to a conventional grammar. This may be the reason for the current exclusion.

2.1.3.4 `!static`

In order to ensure that composition can be more fully controlled, the inverse behaviour of certain keywords whose absence normally denotes their complement can be specified explicitly.

storage-class-specifier: *// Extension of*
    `static`
    <u>`! static`</u>

When composing declarations it is only necessary to supply enough of the declaration name to identify the declaration unambiguously. The remaining parts of multiple declarations compose. Unspecified `static` may compose with either `static` or `!static`. However an attempted composition of `static` and `!static` gives an error.

Implementation

The `!` operator in an expression using a generalised parse of a name can be ambiguous with only one token of look-ahead.

```
    (type) ! static a        // Cast of non-static a
    (type) ! static a        // Cast of complement of static a
```

Neither interpretation is semantically valid, so the generalised name parsing excludes *decl-specifier*s appearing as prefixes.

`!const` and `!volatile` were originally supported but withdrawn for reasons described in Appendix F.1.1.

2.1.3.5 !inline

In C++, an inline function may be explicitly inlined in the interface or in the implementation, depending on whether the `inline` keyword is associated with the interface or the implementation declaration. FOG merges these declarations and so the presence or absence of `inline` in a FOG definition cannot express the three C++ alternatives. The syntax of `inline` is therefore extended to express all three intents explicitly.

function-specifier: *// Extension of*
```
   inline
   ! inline
   inline / implementation
   inline / interface
```

inline

The default form of `inline` directs the function to be inlined if shorter than some simple threshold. The current implementation counts the number of non-trivial tokens in the function-body and compares this to a command line parameter which defaults to 10 tokens.

```
    class X
    {
        inline void f1();
    };
    inline void X::f2() { /* ... */ }
```

To provide a higher degree of compatibility with common coding style, an `inline` appearing within class braces as in `f1` is conditionally inlined within the interface, whereas a function such as `f2` with only an `inline` outside class braces is conditionally inlined in the implementation.

inline/interface

Directs that the function be inlined in the interface.

inline/implementation

Directs that the function be inlined in the implementation (and therefore not in the interface).

!inline

Specifies that the function is not to be inlined in the interface or in the implementation.

Impact

Parsing of a generalised name accepts a *decl-specifier* as a suffix and so

```
    name1 inline / interface ( name2 ) ;
```

could be an expression involving a division and a function call or a more explicitly positioned inline function declaration. This is a false conflict since `inline` can never occur in an expression, so `inline` followed by `/` is unconditionally resolved to an extended form of `inline`.

2.1.3.6 `!virtual`

The syntax for `virtual` is expanded to support its negation and to give an alternative and less cryptic way of specifying pure-virtual.

function-specifier: *// Extension of*
    `virtual`
    <u>`! virtual`</u>
    <u>`virtual / pure`</u>

Impact

The same ambiguity arises and the same resolution is used as for `inline` in Section 2.1.3.5.

2.1.4 Syntax Enhancements

In Section 2.1.3, generalisations and very minor enhancements were introduced to support composition consistently. In this section more significant enhancements are introduced again in support of composition.

2.1.4.1 Default member initializer

In C++, it is easy for a constructor to leave member variables of simple types uninitialised. For classes with a non-trivial number of variables and/or constructors this can be a maintenance problem. FOG allows one set of compositions to add member variables and another to add constructors. There is ample scope for two compositions to aggravate the problem of uninitialized member variables. FOG therefore allows a default initializer to be specified for member variables.

The (C++) syntax for the declaration of global variables and for the definition of any variable outside a class is:

simple-declaration:
    decl-specifier-seq$_{opt}$ *init-declarator-list*$_{opt}$ `;`

init-declarator-list:
    init-declarator
    init-declarator-list `,` *init-declarator*

init-declarator:
    declarator initializer$_{opt}$

initializer:
    = *initializer-clause*
    `(` *expression-list* `)`

initializer-clause:
    assignment-expression
    `{` *initializer-list* `,`$_{opt}$ `}`
    `{ }`

This is incorporated into the more restrictive syntax for definition of member variables within classes.

member-declaration:
    decl-specifier-seq$_{opt}$ *member-declarator-list*$_{opt}$ `;`

member-declarator-list:
  member-declarator
  member-declarator-list , *member-declarator*

member-declarator:
  declarator pure-specifier$_{opt}$
  ~~*declarator constant-initializer*$_{opt}$~~
  declarator initializer$_{opt}$
  identifier$_{opt}$: *constant-expression*

constant-initializer:
  = *constant-expression*

The default initialization of member variables may be declared:

```
class MyClass
{
    bool _satisfies_predicate = false;
    int _usages[3] = { 1, 2, 3 };
    That _that(*this);
};
```

Semantics

The default initializer provides an explicit initial value for use in every constructor that does not provide an initializer. The value is never used in a copy constructor, since a copy constructor provides an implicit initializer for each member.

C++ does not support direct initialization of array members during construction. The array initialization must therefore be synthesised by code placed at the start the constructor body. The construction order of array members is therefore not defined.

Syntax Ambiguity

The constructor form of *initializer* introduces the function-declaration/constructor-invocation ambiguity into a class (Section 4.5.3.2).

```
class A
{
    int a(a_type);        // member function declaration
    int b(not_a_type);    // member variable and initialization
};
```

It is amenable to exactly the same resolution as outside a class. Resolution favours the declaration perspective and so preserves upward compatibility with existing C++ code.

The assignment form is clearer but unable to express multi-argument construction.

2.1.4.2 gcc indexed array initializer

In order to support composition of arrays usefully, it is necessary to be able to specify the location of array initializers. The gcc [Stallman98] indexed array initialization syntax is therefore supported:

initializer-clause:
  assignment-expression
  [*constant-expression*] *assignment-expression*
  { *initializer-list* ,$_{opt}$ }
  { }

Each array initializer may be prefixed by an expression specifying its position.

Implementation

Parsing of this syntax causes no problems in a precise grammar, however an ambiguity arises in the superset grammar, since a prefix `[]` is already recognised to avoid a conflict for `delete []` *cast-expression*. Generalising the solution to the `delete[]` conflict, so that `[` *constant-expression* `]` is parsed as another form of cast, accepts the indexed initializer without extra syntax.

```
abstract_expression:
           parenthesis_clause       // Like '(' expression.opt ')'
    |      '[' expression.opt ']'
cast_expression:
           unary_expression
    |      abstract_expression cast_expression
```

2.1.4.3 compound-declaration

compound-declaration:
  `{` *declaration-seq*$_{opt}$ `}`

compound-declaration is introduced to support polymorphic use of multiple declarations as a single declaration, in the same way as multiple statements can behave as a single statement. However this is a purely lexical grouping; no nested declarative region is defined. The new region has no name; treating it as a declarative region would prevent access to names forming part of a declaration. The analogy is therefore with an anonymous union, whose names are externally visible, rather than with a function block, whose names are local.

2.1.4.4 using

A *using-declaration* supports the re-use of the name of a base-class declaration in a derived class. FOG generalises this concept in a *re-using-declaration* to support re-use of the name of any declaration and re-use (extension) of an existing declaration.

The new syntax supports:

- signature re-use
- function placement
- built-in functionality extension

The specific syntax for a *using-declaration* is removed and covered by adding `using` to *decl-specifier*.

~~*using-declaration:*~~
  `using typename`$_{opt}$ `::`$_{opt}$ *nested-name-specifier unqualified-id*
  `using ::` *unqualified-id*

decl-specifier: *// Extension of*
  `using`

This generalization covers the existing syntax as part of the declarations now contributing to:

simple-declaration:
  decl-specifier-seq$_{opt}$ *init-declarator-list*$_{opt}$ `;`

function-definition:
  decl-specifier-seq$_{opt}$ *declarator pure-specifier*$_{opt}$ *ctor-initializer*$_{opt}$ *function-body*
  decl-specifier-seq$_{opt}$ *declarator pure-specifier*$_{opt}$ *function-try-block*

A long form of the new *re-using-declaration* is present when

- *decl-specifier-seq*$_{opt}$ includes `using`
- *decl-specifier-seq*$_{opt}$ includes a *type-specifier*

A short form of the new *re-using-declaration*, that subsumes the existing syntax and functionality of *using-declaration*, is present when

- *decl-specifier-seq$_{opt}$* includes `using`
- *decl-specifier-seq$_{opt}$* includes no *type-specifier*

Signature re-use

In deep polymorphic, or wide isomorphic object hierarchies, it is common for the same function signature to recur in many, if not all, classes in the hierarchy. This incurs a little lexical redundancy, and acts as a barrier to code evolution; a change to a function signature may involve a very substantial amount of editing. FOG allows a function signature to be defined once and re-used many times, so that changes to the function declaration can be made in one place.

```
class A
{
    public void protocol(int a, double b) const { /* ... */ }
};
class B : public A {};
using B::protocol { /* ... */ }                  // Short form
using A::protocol B::protocol { /* ... */ }      // Long form
```

Both `using` lines contribute code to the function

```
public void B::protocol(int a, double b) const;
```

The short-form creates or extends the entity that must already be unambiguously visible with the name `B::protocol` Since `A::protocol` is visible as `B::protocol`, the function `B::protocol` is created with the same signature as `A::protocol`.

The long-form uses the signature of `A::protocol` (which could be a typedef) to create or extend the function named `B::protocol`. Overload resolution of `B::protocol` is performed using the signature from `A::protocol` to select one precisely matching alternative. There is no overload resolution for `A::protocol`, however distinct names can be associated with each overload by using typedefs to define `A::protocol`.

With either form, the signature of `A::protocol` is being re-used, enabling a change to the signature to be made in one place. However re-used signatures must also re-use parameter names, since there is no need for them to be respecified. This is a little inconvenient, since parameter names now have a more global import. However there may also be some advantages to the requirement for consistent parameter names in closely related functions.

Function placement and tuning of built-in functionality

The C++ compiler may automatically generate code for

- default constructor
- copy constructor
- assignment operator
- destructor
- dereferencing operator

With the exception of the destructor, whose functionality is only extensible, any attempt to modify the auto-generated code requires manual re-implementation of the entire functionality. Trivial modifications such as specification of the access, inlining, virtual or placement in a specific file should be possible. The extended *re-using-declaration* syntax supports this, since re-using a function involves composition with its existing functionality. Therefore within a class declaration, where @This resolves to the class name:

```
using virtual ~@This;
```

defines the destructor as `virtual`, without affecting any other declaration that specifies accessibility.

```
using !inline ~@This;
```

forces an out-of-line implementation of the destructor avoiding any unwanted include file dependencies that might result from the default inline version.

```
using protected operator=(const @This&) { _assigns++; }
```

specifies that the assignment operator is to have protected access and adds a counter update to the existing (default) functionality.

```
using @This(const @This&) : _share_count(1) {}
```

overrides the initialisation of one member in a copy constructor, leaving other members unaffected and therefore retaining their default member-wise copy, whereas:

```
@This(const @This&) : _share_count(1) {}
```

specifies an explicit initialisation of one member, and a default initialisation of all other members.

Member variable delegation

The re-use of a local member variable was at one point considered as a convenient way of requesting the automatic synthesis of code to perform delegation through that member variable. Why this was discarded in favour of a more general solution using meta-programming is described in Appendix F.1.2.

Semantics

The *re-using-declaration* may be elaborated with an *access-specifier*, *decl-specifier*s, *parameter-declaration-clause*, default arguments, *initializer*s, *function-body*s and *object-statement-clause*s. These compose with existing declarations as described in Section 3.4.8.

The long form of *re-using-declaration* applies functionality from the source declaration identified in the *decl-specifier* to the target declaration identified in (each) *init-declarator* or *declarator*. Source and target declarations may be independently scoped and resolved with respect to the surrounding declarative region.

The short form of *re-using-declaration* subsumes the existing *using-declaration*. It specifies both source and target declarations as the *init-declarator* or *declarator*. For compatibility with the existing *using-declaration*, any specified scope must serve to locate the source declaration. The target declaration is therefore necessarily part of the surrounding declarative region.

For both forms, the name provided as the source declaration must be visible within the surrounding declarative region. *decl-specifier*s forming part of the source declaration are copied to the target declaration, except that conflicting *decl-specifier*s are discarded in favour of those forming part of the *re-using-declaration*.

Implementation

When an overloaded signature is re-used, it is not clear which signature is required. The deprecated ARM C++ *access-declaration* suffered from this problem too. In FOG, the problem was originally solved by introducing a nick-naming capability so that overloads could be given alternate names:

```
MyClass::MyClass()/overload=default_constructor
MyClass::MyClass(const @This&)/overload=copy_constructor
MyClass& MyClass::operator=(const @This&)/overload=assign
```

```
using default_constructor { /* ... */ }
```

However it was realised that this was unnecessary as well as clumsy. A more generalised use of a `typedef` is better:

```
auto namespace HandySignatures           //4
{
    typedef default_constructor();
    typedef copy_constructor(const @This& thatObject);
    typedef @This assign(const @This& thatObject);
};
```

```
using HandySignatures::assign MyClass::operator= { /* ... */ }
```

The `typedef`s stretch the syntactic legality of C++, but only define what was previously meaningless. The constructor `typedef`s lack a *type-specifier*, and so there is a potential ambiguity for

```
typedef a(b());
```

between

- the constructor typedef of a taking a pointer to function argument
- the redundantly parenthesised function b returning a.

Only the latter is valid in C++. The former is a new alternative interpretation that must be ignored. The ambiguity does not arise when the missing parameter names are specified as is necessary for the definition of a constructor signature to be useful.

```
typedef a(b (*c));
```

The typedef approach supports sharing of signatures independent of inheritance. The nickname approach needed further elaboration to support this.

The parameter names are retained as part of the `typedef` so that they form part of the re-used signature.

2.1.4.5 `using template`

A `using` keyword may prefix a template-specialization to indicate that the subsequent declarations re-use and so compose with, rather than replace, the declarations from the less specialized template.

template-declaration:
  `export`*opt* <u>`using`*opt*</u> `template` < *template-parameter-list* > *declaration*

explicit-specialization:
  <u>`using`*opt*</u> `template` < > *declaration*

Impact

This usage introduces a parsing lookahead problem with respect to

```
using template name < args >;
```

The lookahead is eliminated by accepting any *decl-specifier* rather than just `using` during syntactical analysis and then rejecting the spurious alternatives during semantic analysis.

4. `class` could be used rather than `auto namespace`. The use of a meta-namespace just serves to eliminate unnecessary declarations emitted for compilation.

2.1.4.6 Object-statement-scopes

Specialized placement of a declaration in a specific file, or accurate resolution of dependencies may require use of FOG extensions to annotate the declaration. Adding the additional syntax to support these declarations is rather difficult, see Appendix F.1.

When composing declarations, it may be necessary to share meta-context between contributions. Defining meta-variables at class scope to share this context can lead to unpleasant interactions when similar composition policies or meta-programs affect more than one function.

Both of these problems are resolved by the introduction of the concept of an object-statement-scope: a declarative region exclusively for use at meta-compile-time. Meta-declarations may be placed in this scope, and shared between contributions to the object. The scope is defined by an *object-statements-clause*, within which, annotations can be placed without introducing syntactical conflicts. It is only necessary to identify one syntax extension that does not conflict with existing syntax. This is achieved by using `:{` and `}` to delimit the region.

init-declarator:
  declarator initializer$_{opt}$ *object-statements-clause*$_{opt}$

member-declarator:
  declarator pure-specifier$_{opt}$ *object-statements-clause*$_{opt}$
  declarator constant-initializer$_{opt}$ *object-statements-clause*$_{opt}$
  identifier$_{opt}$ `:` *constant-expression* *object-statements-clause*$_{opt}$

object-statements-clause:
  `: {` *object-statement-seq*$_{opt}$ `}`

object-statement-seq:
  object-statement
  object-statement-seq object-statement

object-statement:
  `;`
  initializer `;`
  function-used-block
  file-dependency-declaration // Section 5.4.5
  file-placement-declaration // Section 5.4.3
  filespace-declaration // Section 5.4.4
  meta-control-declaration // Section 2.1.5.8
  `auto` *meta-control-declaration* // Section 2.1.5.8
  `auto` *meta-expression-statement* // Section 2.1.5.10
  `auto` *meta-function-definition* // Section 2.1.5.6
  `auto` *meta-variable-declaration* // Section 2.1.5.5
  derived-clause object-statement // Section 2.1.4.8
  derived-clause `: {` *object-statement-seq*$_{opt}$ `}` // Section 2.1.4.8

The rule involving an *initializer* is only semantically valid for a variable-statement-scope (an object-statement-scope associated with a variable).

The rule involving a *function-used-block* is only semantically valid for a function-statement-scope (an object-statement-scope associated with a function).

Section 3.3.2 describes the revised search order for meta-name resolution within the object, then the object scope and then class scopes.

Impact

The syntax is unambiguous because colon is never followed by `{` in C++, however care is required to avoid a shift-reduce conflict with only one token of lookahead.

2.1.4.7 Function-statement-scopes

Object-statement-scopes or more specifically function-statement-scopes are essential for annotating the contributions to a composed function. Each contribution may have its own constructor initializers and function body re-interpreted in derived classes in accordance with a derivation rule.

The contribution may be further annotated to define its include 'file' dependencies, its positioning relative to other contributions and a position in the overall function structure.

function-used-block:
    ctor-initializer `;`
    ctor-initializer$_{opt}$ *function-body*
    function-try-block
    `using` *file-id-list function-used-block*   *// Section 5.4.5*
    segment function-used-block

Note that a function-statement-scope is a syntactic extension of *init-declarator* and consequently requires a trailing semicolon to form a *simple-declaration*. (Implementation as a *function-definition* leads to challenging conflicts.)

```
friend ostream& operator<<(ostream& s, const MyClass& myClass)
:{
    using ostream { /* ... */ return s; }
};
```

Usage of commas to separate multiple declarations with *object-statements-clause*s should be considered extremely bad style, however an outright prohibition appears to add a slight complexity rather than a simplification.

Segments

Program segments identify five distinct domains of composition. Code contributions are composed independently for each segment, but emitted as one contiguous code body. The segments are emitted in the order listed for:

segment:
    `entry`
    `pre`
    `body`
    `post`
    `exit`

The body is the default segment in which code is normally placed.

`entry` and `exit` segments bracket the rest of the code. The intention is that the `entry` segment contain any required declarations and the `exit` segment a `return` statement.

`pre` and `post` are intended for passive code that wraps pre-condition and post-condition checks or diagnostics around the active part of the function.

These default policies are informal. Composition of function bodies and the redefinition of function structure is described in Section 3.4.8.

2.1.4.8 Derivation Rules

A derivation rule determines how a declaration is interpreted so as to automatically generate derived declarations. A declaration is conventionally supplied for a specific scope, which is referred to as the root scope of that declaration. Derivation rules consider the inheritance hierarchy at and below that root scope, evaluating a predicate expression in the meta-name-space of each class to determine whether the derivation rule is enabled.

derived-clause:
    `derived` (*meta-conditional-expression*)

The following pair of declarations

```
protected inline static int Class::static_inheritance_depth()
:{
   derived (is_root())
      { return 0; }
   derived (!is_root())
      { return @{Super}::static_inheritance_depth() + 1; }
};
public virtual int Class::dynamic_inheritance_depth() const
:{
   derived (true)
      { return static_inheritance_depth(); }
};
```

define a pair of functions that are implemented for `Class` and all its derived classes so that invocation of `dynamic_inheritance_depth()` on p, a pointer to a `Class` object:

```
Class& p = ...
... = p.dynamic_inheritance_depth();
```

returns the actual inheritance depth of p. The implementation of the static inline function `static_inheritance_depth()` for the root scope, where the `is_root()` predicate is satisfied, provides a function body that just returns zero. The implementation for derived classes, where the `!is_root()` predicate is satisfied, returns the super-class depth +1. The `true` predicate is always satisfied, and so `dynamic_inheritance_depth()` is a virtual function for the root scope and all derived classes, which ensures that the correct depth is returned.

The predicate *meta-conditional-expression* may involve user-defined or built-in meta-functions and meta-variables. Two groups of built-in functions are provided primarily for use as derivation rule predicates. One group defines structural predicates upon the inheritance tree. The second group defines abstract predicates dependent upon the position of pure virtual functions in the inheritance hierarchy.

Structural Predicates

`is_root()`

Specifies that the declaration is to be applied to the root scope. This is the default derivation rule and ensures upward compatibility with C++.

`true`

Specifies that the declaration is to be applied throughout the inheritance hierarchy: in the root scope and all its derived classes.

`is_leaf()`

Specifies that the declaration is to be applied to all classes in the inheritance hierarchy that have no derived classes. In the degenerate case, this may be just the root scope.

Leaf-ness is independently determined within each meta-compilation session. If further derivation occurs unknown to one session, leaf-based decisions may be inaccurate.

More complicated conditions can be built-up using expression operators:

```
derived(!is_root() && !is_leaf())
```

selects all intermediate nodes in an inheritance hierarchy.

Abstract Predicates

Abstract predicates support efficient and appropriate generation of code depending on the presence of pure virtual functions. The `!is_pure()` predicate may be used to avoid generation of code that illegally attempts to new an abstract class. The `is_boundary()` predicate may place code just once at the inheritance boundary between abstract and concrete.

`is_pure()`

Specifies that the declaration is to be applied to all classes in the inheritance hierarchy that have at least one pure virtual function.

`is_boundary()`

Specifies that the declaration is to be applied to those classes in the inheritance hierarchy that have no pure virtual function, but all of whose base-classes derived from the root scope do have pure virtual functions.

This definition does not permit a second boundary to arise when a further pure virtual function arises at a level of inheritance below an earlier boundary, since the purpose of the predicate is to facilitate placement of code exactly once on every inheritance path from the root scope to each leaf class. The boundary therefore occurs at the least derived alternative.

Semantics

In the absence of a derivation rule, a declaration contributes to its root scope.

With a derivation rule, the declaration name is provisionally present in the root scope and all derived scopes. Once it can be determined that the derivation predicate cannot satisfied, the declaration is disabled, and consequently is not emitted.

Some predicates, such as `true` or `is_root()`, can be evaluated immediately.

Other predicates, such as `is_leaf()` or `is_pure()` can be affected by further declarations or meta-programming and so cannot be evaluated promptly. Resolution is therefore automatically deferred until the code emission phase of meta-compilation. Deferred evaluation of user-defined predicates may be enforced by use of an @ operator to delay evaluation until the body is resolved during code emission.

The derivation predicate gates the body of a declaration

- *function-body* of a function
- *ctor-initializer*s of a constructor
- *initializer* of a variable
- value of a typedef

The body is not evaluated until the predicate has been resolved, thereby avoiding problems that might occur through use of invalid manipulations, such as a base-class (`@Super`) of a root scope.

Deferred evaluation of a body is only possible if the body is not used. References to the body must therefore also use the @ operator to defer evaluation. Direct use of the body such as the use of a typedef to define a function signature are an error if the body is not immediately resolvable.

Implementation

The presence of derivation rules dependent upon the abstract context of a class leads to a potential ambiguity:

```
class Root
{
    virtual void f1() = 0
    :{
        derived(@is_boundary())
            {}
    };
};
class Branch : public Root {};
class Leaf : public Branch {};
```

Which, if any, of `Root::f1`, `Branch::f1` and `Leaf::f1` should be implemented to define the boundary? The `Root` class has an explicit pure virtual, is clearly abstract and so cannot constitute a boundary. Implementation of none or one of `Branch::f1` and `Leaf::f1` gives a consistent behaviour.

A unique behaviour is defined by the presumption that derivation predicates are satisfied until the converse is established.

During the final compilation phase described in Section 3.1, the derivation predicates are evaluated in construction order as each class is prepared for emission. While preparing the `Branch` class, the derivation predicate of `Branch::f1` is unresolvable and so presumed true. The function is therefore still enabled and provides an implementation of the inherited pure virtual. The `Branch` class has no residual pure virtual functions and is therefore concrete. The class state determined, the derivation predicate can be evaluated and found to be satisfied. The `Branch::f1` function remains enabled. Subsequent preparation of the `Leaf` class detects that no pure-virtual is inherited (or locally declared); the predicate is not satisfied, and so the `Leaf::f1` declaration is disabled and not emitted.

Implementation Note

Derivation rules were introduced in FOG before meta-programming. They offer little that cannot be achieved by meta-programming, but do so with a much more compact and manageable syntax for many common cases.

2.1.5 Meta-Programming

Meta-programming extends C++ run-time behaviour by providing very similar behaviour at (meta-)compile-time.

In order to introduce additional functionality without new reserved words, the existing syntax is heavily overloaded, which has the advantage of requiring very little new syntax to be learnt, but incurs the risk that the rather different behaviour may be overlooked.

The `auto` keyword is used to introduce meta-functionality. Its existing usage is only valid as a *decl-specifier* within a function and so all other usage of `auto` is retracted, at no cost to semantically valid programs.

storage-class-specifier: *// Part of*
    ~~`auto`~~

Meta-functionality extends each of three contexts in the C++ grammar where constructs are introduced. FOG adds a fourth context as object-statement-scopes.

namespace statements

Contributions to namespaces occur as a sequence of *declaration* within the top-level (unnamed global namespace), within namespaces, and within external linkages:

declaration-seq:
    declaration
    declaration-seq declaration

translation-unit:
    declaration-seq$_{opt}$

namespace-body:
    declaration-seq$_{opt}$

linkage-specification:
    `extern` *string-literal* { *declaration-seq*$_{opt}$ }
    `extern` *string-literal declaration*

member statements

Contributions to classes occur as a sequence of *member-declaration*

member-specification:
    member-declaration member-specification$_{opt}$
    access-specifier : *member-specification*$_{opt}$

class-specifier:
    class-head { *member-specification*$_{opt}$ }

function statements

Contributions to function bodies occur as a sequence of *statement*

function-body:
    compound-statement

compound-statement:
    { *statement-seq*$_{opt}$ }

statement-seq:
    statement
    statement-seq statement

object-statement-scopes

Contributions to object statements are described in Section 2.1.4.6.

The syntax of declarations is extended by the addition of meta-statements so that conditional and iterated compilation may embrace declarations. The existing statement syntax is re-used as is, with the result that declaration/expression ambiguities now occur at the declaration as well as statement level; extension of the existing disambiguation rule (§6.8) to favour declarations preserves compatibility. `goto`s and associated statement labels are excluded, primarily because the label syntax has a challenging ambiguity with respect to an anonymous bit-field.

Meta-variables and meta-functions are added by overloading the `auto` keyword to mean “meta”. Since the `auto` keyword has no meaning outside of functions this change merely gives meaning to constructs that are meaningless in C++.

Meta-statements are added to statement syntax by using an `auto` prefix.

Further lexical exposition to show *declaration*, *member-declaration* and *statement* is not particularly helpful. A tabular exposition is therefore provided in Table 2.1.

Non-terminal	*declaration*	*member-declaration*	*statement*	*object statement*	*tree statement*
Compound Statements					
compound-declaration	FOG	FOG			
compound-statement			C++		
compound-tree-statement					FOG
Control Statements and Declarations					
control-statement *try-block*			C++		FOG
meta-control-statement			auto		
meta-control-declaration	FOG	FOG		FOG	auto
Expression Statements					
expression-statement			C++		FOG
meta-expression-statement	FOG	FOG	auto	FOG	auto
Declaration Statements					
function-definition *template-declaration*	C++	C++			FOG
using-declaration	C++	C++	C++		FOG
simple-declaration 'simple-'*member-declaration*	C++	C++	C++		FOG
=assignment-expression	C++	FOG	C++		FOG
={ ... }	C++	FOG	C++		FOG
(expression-list)	C++	FOG	C++		FOG
=constant-expression	(C++)	C++	(C++)		(FOG)
bit-field	FOG	C++			FOG
meta-class-declaration *meta-function-definition* *meta-variable-declaration*	FOG	FOG		FOG	FOG
Others					
explicit-instantiation *explicit-specialization* *linkage-specification* *namespace-definition*	C++	FOG			FOG
asm-definition *namespace-alias-definition* *using-directive*	C++	FOG	C++		FOG
access-declaration[a] *access-specifier* :	FOG	C++[b]			FOG
file-dependency-declaration *file-placement-declaration* *filespace-declaration*	FOG	FOG		FOG	FOG
include-declaration *namespace-declaration*	FOG	FOG			FOG
syntax-macro-definition	FOG	FOG			
'most-other-things'					FOG

Table 2.1 Statement and Declaration Grammar

a. *access-declaration* is deprecated and therefore now appears as just *qualified-id* in the grammar.

b. *access-specifier* : is syntactically a *member-declaration* since it is interchangeable in its only usage which is as a sequence of *member-declaration*s in a *member-specification*.

Boxes marked

- `C++` identify the standard C++ grammar
- `FOG` identify FOG extensions
- `auto` identify FOG extensions disambiguated by an `auto` keyword
- () identify degenerate grammar

The *object-statement* column is discussed in Section 2.1.4.6.

The *tree-statement* column is discussed in Section 2.1.1.6.

In lexical form the FOG extensions are:

statement:	*// Extension of*
`auto` *control-statement*	
`auto` *meta-expression-statement*	

control-statement:	*// New non-terminal with old functionality*
labeled-statement	
selection-statement	
iteration-statement	
jump-statement	

declaration:	*// Extension of*
namespace-declaration	*// Section 2.1.3.1*
accessibility-specifier	
compound-declaration	*// Section 2.1.4.3*
meta-control-declaration	*// Section 2.1.5.8*
`auto` *meta-control-declaration*	*// Section 2.1.5.8*
expression-statement	*// Section 2.1.5.10*
`auto` *meta-expression-statement*	*// Section 2.1.5.10*
`auto` *meta-class-declaration*	*// Section 2.1.5.2*
`auto` *meta-function-definition*	*// Section 2.1.5.6*
`auto` *meta-variable-declaration*	*// Section 2.1.5.5*
syntax-macro-definition	*// Section 2.1.6*
include-declaration	*// Section 5.4.6*
file-dependency-declaration	*// Section 5.4.5*
file-placement-declaration	*// Section 5.4.3*
filespace-declaration	*// Section 5.4.4*

The syntactic difference between *declaration* and *member-declaration* is eliminated and consequently the *member-declarator* syntax must be added to *init-declarator* and an *accessibility-specifier* to *declaration*. The changes are syntactic only; C++ *declaration*s that are not syntactically valid as *member-declaration*s and conversely C++ *member-declaration*s that are not syntactically valid as *declaration*s are semantically invalid in FOG: accessibility and bit-fields may not be specified for namespaces and *explicit-instantiation*s or *using-directive*s may not be specified within classes.

init-declarator:
    declarator pure-specifier$_{opt}$ *object-statements-clause*$_{opt}$
    declarator initializer$_{opt}$ *object-statements-clause*$_{opt}$
    identifier$_{opt}$ `:` *constant-expression object-statements-clause*$_{opt}$

class-specifier:
    class-head `{` ~~*member-specification*~~$_{opt}$ *declaration-seq*$_{opt}$ `}`

accessibility-specifier:
    access-specifier `:`

~~*member-specification*~~*:*

~~*member-declaration*~~*:*

~~*member-declarator-list*~~*:*

~~*member-declarator*~~*:*

2.1.5.1 Meta-names

The built-in types have meta-classes and `auto` is the root of all meta-classes. We therefore define names that incorporate these alternative scopes, including their meta-constructors and meta-destructors.

built-in-type-id: // *e.g.* `unsigned int`
 built-in-type-specifier
 built-in-type-id built-in-type-specifier

meta-id:
 id
 meta-type
 `auto`

meta-nested-id:
 meta-id
 ~ *meta-id*
 meta-id `::` *meta-nested-id*

meta-scoped-id: // *e.g.* `::auto::symbol_table`
 `::`$_{opt}$ *meta-nested-id*

2.1.5.2 Meta-classes

Section 3.5 describes how every user-defined and built-in type has a meta-class with the same name. The meta-class defines additional functionality for use at compile-time. The meta-class is discarded after meta-compilation completes, so that the meta-class forms no part of the emitted code.

Meta-classes that have only meta-declarations are not needed at compile-time and so such classes are not emitted. In order to forward reference such a meta-class, or to diagnose any inadvertent use of compile-time declarations, an `auto` may prefix a very similar syntax to a *class-specifier*. This asserts that the meta-class alone is required, avoiding an empty class declaration cluttering the generated output.

meta-class-id:
 meta-id
 meta-id `::` *meta-class-id*

meta-class-key:
 class-key
 `namespace`

meta-class-specifier:
 meta-class-key meta-class-id
 meta-class-key meta-class-id base-specifier-clause$_{opt}$ `{` *declaration-seq*$_{opt}$ `}`

meta-class-declaration:
 meta-class-specifier `;`

declaration: // *Extension of*
 `auto` *meta-class-declaration*

*access-specifier*s are ignored for meta-declarations in a meta-class. All meta-declarations are therefore public. The presumption is that meta-functionality contributes to a tightly coupled pool of collaborating code, where access restrictions would be an inconvenience rather than an asset.

Impact

Re-use of reserved-words such as `class` as meta-type-names creates some inconvenient ambiguities with respect to *meta-variable-declaration* and *meta-function-definition*s. Most of these can be resolved by careful implementation to share common parsed prefixes. It is not obvious how to solve the problem of a global name as a *meta-class-id*, for which the leading `::` in

```
auto class ::MyClass { /* ... */ };
```

signifies the nested meta-class `class::MyClass` of the `class` meta-type. Introducing parentheses:

```
auto class (::MyClass) { /* ... */ };
```

satisfies a generalised syntax for the meta-constructor of the `class` meta-type. However this cannot be unified because the body of a meta-constructor comprises statements whereas the body of a meta-class comprises declarations.

Therefore FOG does not support explicitly global scoping in *meta-class-specifier*s. An inelegant workaround is:

```
    auto identifier globalScopeId = "";
    auto class ${globalScopeId}::MyClass { /* ... */ };
```

2.1.5.3 Meta-bases

Every user-defined and built-in type has a corresponding meta-class, whose meta-inheritance corresponds to the compile-time inheritance. The meta-base classes are the meta-classes of the base-classes. The meta-inheritance is augmented so that every meta-class without a meta-base class meta-inherits from the built-in root meta-class named `auto`. Additional meta-inheritance may be specified by using the `auto` keyword as an *access-specifier* in a *base-specifier*.

base-specifier:
    ~~`::`$_{opt}$ *nested-name-specifier*$_{opt}$ *class-name*~~
    ~~`virtual` *access-specifier*$_{opt}$ `::`$_{opt}$ *nested-name-specifier*$_{opt}$ *class-name*~~
    ~~*access-specifier* `virtual`$_{opt}$ `::`$_{opt}$ *nested-name-specifier*$_{opt}$ *class-name*~~
    scoped-id
    built-in-type-id
    access-specifier base-specifier
    `virtual` *base-specifier*
    `! virtual` *base-specifier*
    `auto` *base-specifier*

The rewritten recursion removes the syntactic limitations on multiple *access-specifier*s. It is therefore a semantic error for more than one of `auto` and the three distinct *access-specifier*s or for both `virtual` and `!virtual` to be supplied.

The `virtual` keyword is ignored for meta-base classes whose behaviour is always virtual; only one copy of a meta-class is inherited.

2.1.5.4 Meta-types

The arguments and returns of meta-functions and the values of meta-variables are defined by meta-types. All meta-types are built-in and there is no facility for user-defined meta-types. The defined set is

meta-type:
    meta-type-name
    built-in-type-id
    meta-class-key
    `enum`
    `typedef`
    `typename`
    `using`

The direct mention of `typedef`, and indirect mention of `class` and `signed` re-instates names that duplicate reserved words and which were consequently tokenized as a *reserved-word* rather than a *meta-type-name* (Section 2.1.1.4). All the built-in C++ numeric types are available for use as meta-types for compile-time calculations. (The current FOG implementation maps the 20 distinct C++ types to one of: `bool`, `unsigned`, `signed` or `double`.)

meta-type-name:
    intrinsic-meta-type-name
    actual-meta-type-name
    potential-meta-type-name

The intrinsic meta-types define concepts that do not depend upon their program context.

intrinsic-meta-type-name: one of

```
array_modifier              assignment_expression
character                   class_key
constant_expression         cv_qualifier
decl_specifier              expression
function_modifier           handler
identifier                  iterator
keyword                     meta_type
modifier                    name
nil                         number
pointer_modifier            punctuation
reference_modifier          reserved
string                      token
tree_literal                using_directive
```

Potential meta-types define concepts that have limited meaning until associated with some parent context.

potential-meta-type-name: one of

```
base_specifier              built_in_type_specifier
class_specifier             declaration
elaborated_type_specifier   enum_specifier
enumerator_definition       exception_specification
filespace_specifier         function_specifier
linkage_specification       meta_class_specifier
meta_function_specifier     meta_parameter_specifier
meta_variable_specifier     namespace_definition
namespace_alias_definition  object_specifier
parameter_specifier         scope_specifier
statement                   template_parameter_specifier
type_id                     type_specifier
typedef_specifier           using_declaration
variable_specifier
```

Actual meta-types define concepts that have been associated with a parent context, and in many cases correspond to emitted declarations.

actual-meta-type-name: one of

```
base                        built_in
class                       entity
enum                        enumerator
exception                   filespace
function                    linkage
meta_class                  meta_function
meta_parameter              meta_variable
namespace                   namespace_alias
object                      parameter
scope                       template_parameter
type                        typedef
typename                    using
variable
```

The semantics of these types is defined in Section 3.1.2.

Although meta-types are built-in, their names are not reserved words. There is therefore no compatibility problem when migrating C++ code that makes use of some of the meta-type names for its own identifiers. It is just a little confusing to read:

```
bool moveable;
typedef bool variable;
variable t = moveable;        // Assign value to run-time type
auto variable t = moveable;   // Assign name to compile-time meta-type
```

The values stored in meta-variables may be scalars or trees exploiting the polymorphic characteristics available at compile that are described in Section 3.1.5.

2.1.5.5 Meta-variables

Meta-variables support storage of values at compile-time. Their definition differs from conventional variables through the use of an `auto` prefix, the requirement for an initializer, and the acceptance of almost any syntactically valid construct as that initializer.

meta-variable-declaration:
  static$_{opt}$ `const`*$_{opt}$ meta-type meta-scoped-id exposed-tree$_{opt}$ = tree-statement*
  static$_{opt}$ `const`*$_{opt}$ meta-type meta-scoped-id exposed-tree$_{opt}$ object-statements-clause*
  static$_{opt}$ `const`*$_{opt}$ meta-type* (*meta-scoped-id*) *exposed-tree$_{opt}$ = tree-statement*
  static$_{opt}$ `const`*$_{opt}$ meta-type* (*meta-scoped-id*) *exposed-tree$_{opt}$ object-statements-clause*

exposed-tree:
  `[ ]`

declaration: *// Extension of*
  `auto` *meta-variable-declaration*

When `[]`’s are omitted, the syntax defines a scalar meta-variable with a single initializer. The single initializer may be a tree of initializers with *compound-tree-statement*s used as *tree-statement*s to create the tree structure. All leaves in the initializer tree must satisfy the syntax of the meta-type. When subsequently assigned to an iterator, the iteration domain comprises the one root element.

When `[]`’s are present, the syntax similarly defines a meta-variable but from a compound initializer. When subsequently assigned to an iterator, the iteration domain comprises the first generation of children, thereby treating the tree as an array.

The semantics of the composed list and tree types are discussed in Section 3.1.5.

`static` meta-variables have a single value shared by all derived meta-classes.

`!static` meta-variables have distinct copies for each derived class.

`const` meta-variables may have only a single unchanging value. Assignment or redeclaration is illegal.

`!const` meta-variables may change value either by assignment or by composition with a further declaration.

Meta-variables are `!const` and `!static` by default.

Impact

The *meta-expression-statement* and *meta-variable-declaration* syntaxes exhibit expression/declaration ambiguities. The existing disambiguation rule is extended. An ambiguity is resolved in favour of the meta-declaration.

Members of the meta-classes of the built-in types and members of globally scoped meta-classes can be difficult to specify:

```
auto int int::a = 0;
auto int ::MyClass::a = 0;
```

In the first case, the disambiguation rule for built-in types maximises the length of built-in type specifiers, treats `int int` as a single type and so the example is an assignment expression.

In the second case, extrapolation from the §7.1-2 disambiguation rule, that maximises the length of a *type-specifier-seq*, maximises the length of the alternating names and scopes and so once again the example is an assignment to the semantically illegal nested class of `int`.

Parentheses must be used to define valid declarations.

```
auto int (int::a) = 0;
auto int (::MyClass::a) = 0;
```

Each of these examples is now syntactically valid as both a meta-expression and a meta-declaration, so the disambiguation rule resolves in favour of the required meta-declaration.

There is no ambiguity with respect to meta-constructor declarations, since there are no pure virtual meta-functions and meta-constructors have no parameters.

2.1.5.6 Meta-functions

Meta-functions provide code for execution during the meta-compilation process.

declaration: *// Extension of*
    `auto` *meta-function-definition*

meta-function-definition: *// Part of*
    static$_{opt}$ *meta-type meta-scoped-id* `(` *meta-parameter-list*$_{opt}$ `)` *exposed-tree*$_{opt}$
        compound-tree-statement
    static$_{opt}$ *meta-type meta-scoped-id* `(` *meta-parameter-list*$_{opt}$ `)` *exposed-tree*$_{opt}$
        object-statements-clause

meta-parameter-list:
    meta-parameter
    meta-parameter-list `,` *meta-parameter*

meta-parameter:
    meta-type identifier exposed-tree$_{opt}$
    meta-type identifier exposed-tree$_{opt}$ *= tree-argument*

The `static` keyword is ignored, since there is no distinction between static and non-static meta-functions. There is one function for the meta-class which may be inherited or overridden by derived classes.

The syntax is very similar to a conventional function declaration, except for the use of the much simpler and restrictive meta-type system. Default parameter values are supported, but not overloading or exception specifications.

The lack of support for overloading was once necessary because resolution of the meta-function was necessary to identify the syntax with which each argument was parsed. The evolution to a context free grammar removes this constraint. Overloading could now be supported. It is just a matter of defining overload resolution rules that are in keeping with existing C++ overload resolution policies, but which are also appropriate for the dynamic rather than static type information available during meta-compilation.

An alternative form of definition and invocation is provided by the syntax-macro defined in Section 2.1.6.

2.1.5.7 Meta-constructor and Meta-destructor

Meta-constructors and meta-destructors provide for relatively independent meta-programs. They are invoked automatically during the meta-construction and meta-destruction compilation phases. They therefore have no parameters.

meta-nested-constructor-id:
    meta-id
    meta-id `::` *meta-nested-constructor-id*

meta-scoped-constructor-id:
    `::`$_{opt}$ *meta-nested-constructor-id*

meta-nested-destructor-id:
    ~ meta-id
    meta-id `::` *meta-nested-destructor-id*

meta-scoped-destructor-id:
    `::`$_{opt}$ *meta-nested-destructor-id*

meta-function-definition: *// Part of*
    meta-scoped-constructor-id `( )` *compound-tree-statement*
    meta-scoped-constructor-id `( )` *object-statements-clause*
    meta-scoped-destructor-id `( )` *compound-tree-statement*
    meta-scoped-destructor-id `( )` *object-statements-clause*

2.1.5.8 Meta-control-statements and meta-control-declarations

Meta-statements control compilation. The control part of a *meta-control-statement* or *meta-control-declaration* is evaluated, as the source text is analysed or a meta-program executed, to affect the interpretation of the child *statement* or *declaration*.

Within a *statement-seq*, the conventional program control statements retain their run-time meaning. Additional meta-programming control applies when an `auto` prefix is used.

statement: *// Extension of*
    control-statement
    `auto` *control-statement*

control-statement:
    labeled-statement
    selection-statement
    iteration-statement
    jump-statement

Within a *declaration-seq*, C++ provides no program control and so the existing statement syntax is re-used with an optional `auto` prefix to define a *meta-control-declaration*.

declaration: *// Extension of*
    meta-control-declaration
    `auto` *meta-control-declaration*

meta-control-declaration:
    `case` *constant-expression* `:` *declaration*
    `default :` *declaration*
    `do` *declaration* `while (` *expression* `) ;`
    `for (` *for-init-statement condition*$_{opt}$ `;` *expression*$_{opt}$ `)` *declaration*
    `if (` *condition* `)` *declaration*
    `if (` *condition* `)` *declaration* `else` *declaration*
    `switch (` *expression* `)` *declaration*
    `while (` *condition* `)` *declaration*
    jump-statement

The *meta-control-declaration* syntax repeats nearly all the program control syntax of *statement* replacing child *statement*s by child *declaration*s. The `goto` form of *labeled-statement* is omitted to avoid introduction of a syntax ambiguity with respect to an anonymous bit-field. `goto` is therefore not supported for meta-programming.

A `return` meta-statement has no meaning, since the semantics of meta-function execution involve a return of the entire meta-function body as a tree for interpretation in the calling context.

Impact

Interspersing compile-time and run-time control apparently introduces more ambiguities:

```
do
{
    if (...)
    {
        auto do
        {
            auto if (... ) {}
        } while (...);
    }
    else {...}
} while (...);
```

How do the `if`s, `else`s, `do`s and `while`s pair up?

The meta-syntax is integrated with the language. Statements nest, and so the pairing is exactly the same as would be the case if the `auto` keywords were removed so that all meta-statements change to statements. The dangling `else` ambiguity is resolved as always to the nearest `if`, or `auto if`.

It is unnecessary and highly undesirable to introduce an `auto` prefix for `else` or the `while` of `do...while`. This would permit the meta-programming control flow to interleave programming control flow. Although this is permitted by the C preprocessor, it leads to difficult to understand code.

```
        if
        {
            a();
#if B
        }
        else
        {
#endif
            b();
#if !B
        }
        else
        {
#endif
            c();
        }
```

If permitted in FOG, the use of almost identical syntax for meta-statements and statements would render interleaved code unintelligible.

There is also a pragmatic reason to avoid prefixing `else` with `auto`: 2 tokens of lookahead are required to resolve a dangling `auto else` ambiguity.

The *declaration*s in selection and iteration meta-statements have a data-dependent interpretation. These productions must therefore be parsed with the appropriate number of side-effects. This may be achieved either by using a simple lookahead parser to cache the unparsed tokens for deferred parsing under control of the parent statement. Alternatively, AST nodes may be constructed directly, but without any resolution of tree-literals. Resolution of tree-literals is initiated by the parent statement. The earlier multi-pass FOG implementation was context-dependent and took the former approach. The superset grammar is context-free and takes the latter.

2.1.5.9 Meta-typedef

A typedef defines a more convenient name for a run-time type during compilation. That name has no meaning at run-time.

A meta-typedef could therefore define a more convenient name for a compile-time type during (meta-)compilation. That name also has no meaning at run-time.

meta-typedefs are not currently implemented in FOG. There is no obvious reason why they should not be. Arguably, the typedefs in `HandySignatures` on page 32 are meta-typedefs.

2.1.5.10 Meta-expression-statement

A *meta-expression-statement* takes two forms, each of which is preceded by an `auto` prefix to distinguish from the very similar syntax of an *expression-statement*. The first form performs some operation as a side effect:

```
auto a++;
auto f();
```

The second form updates a meta-variable

```
auto a = 5;
auto listVariable += { "a", a * b, g[] };
```

meta-primary-expression:
    literal
    `this`
    meta-scoped-id
    meta-type meta-nested-id
    (*tree-argument-list*$_{opt}$)

meta-postfix-expression:
  meta-primary-expression
  meta-postfix-expression (*tree-argument-list*$_{opt}$)
  meta-postfix-expression [*expression*$_{opt}$]
  meta-postfix-expression . *declarator-id*
  meta-postfix-expression -> *declarator-id*
  meta-postfix-expression ++
  meta-postfix-expression --

meta-unary-expression:
  meta-postfix-expression
  ++ *meta-unary-expression*
  -- *meta-unary-expression*
  * *meta-unary-expression*
  + *meta-unary-expression*
  - *meta-unary-expression*
  ! *meta-unary-expression*
  ~ *meta-unary-expression*
  `sizeof` *unary-expression*

meta-multiplicative-expression:
  meta-unary-expression
  meta-multiplicative-expression * *meta-unary-expression*
  meta-multiplicative-expression / *meta-unary-expression*
  meta-multiplicative-expression % *meta-unary-expression*

meta-additive-expression:
  meta-multiplicative-expression
  meta-additive-expression + *meta-multiplicative-expression*
  meta-additive-expression - *meta-multiplicative-expression*

meta-shift-expression:
  meta-additive-expression
  meta-shift-expression << *meta-additive-expression*
  meta-shift-expression >> *meta-additive-expression*

meta-relational-expression:
  meta-shift-expression
  meta-relational-expression < *meta-shift-expression*
  meta-relational-expression > *meta-shift-expression*
  meta-relational-expression <= *meta-shift-expression*
  meta-relational-expression >= *meta-shift-expression*

meta-equality-expression:
  meta-relational-expression
  meta-equality-expression == *meta-relational-expression*
  meta-equality-expression != *meta-relational-expression*

meta-and-expression:
  meta-equality-expression
  meta-and-expression & *meta-equality-expression*

meta-exclusive-or-expression:
  meta-and-expression
  meta-exclusive-or-expression ^ *meta-and-expression*

meta-inclusive-or-expression:
  meta-exclusive-or-expression
  meta-inclusive-or-expression | *meta-exclusive-or-expression*

meta-logical-and-expression:
  meta-inclusive-or-expression
  meta-logical-and-expression && *meta-inclusive-or-expression*

meta-logical-or-expression:
  meta-logical-and-expression
  meta-logical-or-expression || *meta-logical-and-expression*

meta-conditional-expression:
  meta-logical-or-expression
  meta-logical-or-expression ? *meta-conditional-expression* : *meta-conditional-expression*

meta-expression-statement:
  meta-conditional-expression ;
  meta-logical-or-expression assignment-operator tree-statement

The *expression* syntax is repeated so that *meta-primary-expression* resolves *meta-scoped-id* in the meta-name-space, whereas *primary-expression* resolves *declarator-id* in the conventional name-space. The semantics of meta-assignment differ and some inappropriate operators are omitted. A conventional assignment is right associative:

```
a = b = c = d;
```

is equivalent to:

```
a = (b = (c = d));
```

A meta-assignment does not associate. The entire right-hand side is analyzed as a *tree-statement* and is assigned to the left-hand side as a literal syntactic element:

```
auto a = b = c = d;
```

assigns `b = c = d` to `a`. Of these names, only `a` is resolved in the meta-name-space. `b`, `c` and `d` are not resolved in any namespace, since they form part of a literal. Eventually when the value of `a` is used, the name-space in which to resolve `b`, `c` and `d` may become clear.

```
$a;                         // b = c = d;
                            // b, c, d in normal name-space.
auto $a;                    // auto b = c = d;
                            // b resolved in meta-name-space
                            // c and d still literals
```

The generality required to parse the *tree-statement* necessitates a simplification of *meta-conditional-expression* to exclude assignments and comma-separated lists of meta-expressions.

The meta-expression grammar has the following further incidental differences:

- no casts
- no `typeid`
- no `new`
- no `delete`
- no unary `&`
- no `.*` or `->*`

Removing explicit casts simplifies the implementation. The remaining constructs are inappropriate for the current language definition.

Impact

The *meta-expression-statement* and *meta-variable-declaration* syntaxes exhibit expression/declaration ambiguities. The existing disambiguation rule (§6.8) is extended. An ambiguity is resolved in favour of the meta-declaration.

The *tree-statement* is initially analyzed without knowledge of the expected syntactic type. When assigned to a meta-variable, used as a meta-parameter or returned from a meta-function, a further analysis checks that the value is compatible with the required meta-type. The program is ill-formed if it is incompatible.

```
auto class_specifier x = class X {};
auto class_specifier y = class Y {};
y = class X {};
y = $x;
y = x;                         // Error x is not a class-specifier
auto y = class X {};
```

2.1.6 Syntax macros

Syntax macros provide an alternate form of meta-function that may be invoked automatically via a simple pattern match rather than by a $-expression.

syntax-macro-definition:
    `explicit` `auto` *meta-type identifier* (*syntax-macro-parameter-list*$_{opt}$) *exposed-tree*$_{opt}$
        compound-tree-statement

syntax-macro-parameter-list:
syntax-macro-parameter
syntax-macro-parameter-list `,` *syntax-macro-parameter*

syntax-macro-parameter:
meta-type identifier exposed-tree$_{opt}$
identifier
reserved-word
punctuation

Syntax macros are described in Section 3.7.

2.2 Built-In Functionality

The built-in functionality is described at length in Appendix E, and so only a very brief summary is provided here.

2.2.1 Built-in Root Meta-class

`auto` is the root meta-class for all other meta-classes (including those for namespaces and built-in types). It has no functionality, but user-defined functionality can be added and thereby affect all classes.

2.2.2 Built-in Meta-variables

Meta-type-specific meta-variables support access to declarations.

2.2.3 Built-in Meta-functions

Meta-type-specific meta-functions support access to declarations.

2.2.4 std meta-namespace

The `std` meta-namespace is used as a repository for useful language support meta-functions.

`nil std::diagnostic(string aString)`
`nil std::error(string aString)`
`nil std::warning(string aString)`

generate diagnostic, error and warning messages.

`string std::get_cpp(string aString)`

resolves a definition within the preprocessor/command line name-space.

`string std::get_env(string aString)`

resolves a definition within the external environment.

`string std::date()`
`string std::file()`
`string std::time()`

replace the ANSI C `__DATE__`, `__FILE__` and `__TIME__` macros.

`tree std::parse(string aString)`
`tree std::parse_tokens(token[] someTokens)`
`token[] std::tokenize(string aString)`

support character-level substitution and re-entrant analysis.

2.3 Incompatibilities

In principle the FOG grammar is a superset of the C++ grammar. In practice there are some very minor incompatibilities.

2.3.1 Semantic Errors

Most of the significant FOG enhancements occur by defining a meaning for constructs that are semantic (or syntactic) errors in C++. Therefore a C++ program with semantic errors may be error-free in FOG.

This is an inevitable consequence of enhanced semantics.

2.3.2 Transparency

FOG generates multiple output files with a default disposition to an interface and an implementation file per top level class. If FOG is used to preprocess C++ code requiring a more sophisticated file allocation, the file structuring may be lost. *file-placement-declaration*s may be needed to create the required structure. This is an inconvenience that occurs when porting C++ code to FOG. C++ code used via `#include` or `using/utility` is not regenerated and so retains its file structure.

2.3.3 `auto`

FOG meta-constructs are defined by overloading the `auto` keyword. This causes no incompatibility outside functions, where the keyword is semantically invalid. The keyword is required to disambiguate syntax within a function.

```
int i = 5;            // Assignment of 5 to (integer) i
auto e = i = 5;       // Assignment of i = 5 to (expression) e
```

2.3.4 Incompatible concatenation

Extension of the ANSI C string concatenation policy to characters, identifiers, numbers and strings potentially introduces a major incompatibility for FOG. However the definition of the major syntax elements of C++ (whitespace, punctuation, characters, numbers, strings, identifiers and keywords) and accidental properties of the C++ grammar limit the problems.

An incompatibility arises wherever FOG concatenates but C++ does not. This may occur when characters, identifiers (including keywords), numbers and strings occur without intervening whitespace or punctuation. There are 16 combinations of adjacent characters, identifiers, numbers and strings to consider:

Character-anything only occurs in C++ when a *character-literal* arises as a *literal* in a *primary-expression* and the subsequent operator is not punctuation. In ARM C++ [Ellis90] there were no non-punctuation operators, however the standard [C++98] introduced the alternative tokens (`and`, `and_eq`, `bitand`, `bitor`, `compl`, `not`, `not_eq`, `or`, `or_eq`, `xor`, `xor_eq`).

Anything-character occurs in C++ when a *character-literal* arises as a *literal* in a *primary-expression* and is preceded by non-punctuation. This occurs for the alternative tokens and following a `return`, `sizeof` or `throw` keyword.

The same possibilities occur for number-anything, anything-number, string-anything and anything-string.

String-string concatenates in FOG, and corresponds to ANSI C behaviour.

```
"This " "is"" ""a" " single"" string" // "This is a single string"
```

Identifier-identifier cannot occur since whitespace or punctuation is required to terminate the first identifier.

For identifier-string and string-identifier there is a further problem that arises in the *linkage-specification* syntax.

```
extern"C"size_t f;
```

must be written in FOG as

```
extern "C" size_t f;
```

This is unlikely to cause many problems since few programmers would choose to be so economical with whitespace in these cases.

The FOG concatenation extension therefore introduces minor incompatibilities. Whitespace cannot be omitted around the *string-literal* of a *linkage-specification*, following `return`, `sizeof` or `throw` or around alternative tokens, if the omission conflicts with a concatenation interpretation.

2.4 Cpp Replacement

Stroustrup, in The Design and Evolution of C++ [Stroustrup97], identifies elimination of the preprocessor as a major goal for C++, devoting the final chapter to a discussion of its weaknesses, and identifying some remedies that C++ provides. In the final paragraph, Stroustrup writes

> "I'd like to see Cpp abolished. However the only realistic and responsible way of doing that is first to make it redundant ...".

In order to justify the claim that FOG makes Cpp redundant, we must briefly review the deficiencies and facilities of Cpp to determine to what extent FOG resolves and replaces them.

2.4.1 Cpp limitations

Cpp supports the definition of and replacement of object-like and function-like macros.

An object-like macro associates an identifier with a replacement sequence of preprocessor tokens.

```
#define OCTAL_CASES \
      '0': case '1': case '2': case '3': case \
      '4': case '5': case '6': case '7'
```

The replacement tokens replace the macro identifier wherever it occurs.

```
switch (c)
{
   case OCTAL_CASES: /* ... */ break;
   case ALPHABETIC_CASES: /* ... */ break;
   case '%': /* ... */ break;
}
```

Substitution occurs at a very low level, offering the programmer considerable flexibility. In the above example, the replacement sequence omits an initial case keyword and a trailing colon token. The missing tokens accompany the instantiation. Readers may form their own opinion as to whether the unusual definition leads to a dangerously obscure or aesthetically pleasing implementation of the `switch` statement.

A function-like macro associates an identifier and a list of formal parameters with a replacement sequence.

```
#define MAX(a,b) ((a) > (b) ? (a) : (b))
```

Invocation of the macro provides the actual arguments that replace the formal parameters in the replacement sequence.

Macro substitution is simple but prone to accidents. Substitution occurs at a lexical level and so ignores any logical structure that may be present in the source code.

The intention that `MAX` returns an expression from a pair of expression arguments is only realised when the actual usage is appropriate.

The apparently redundant parentheses in the macro definition avoid the surprising evaluation that would otherwise result from the interpretation of

```
price + MAX(current_rate, fixed_rate) * commission
```

as

```
(price + current_rate) > fixed_rate)
   ? current_rate : (fixed_rate * commission)
```

2.4.1.1 Unwanted substitution

Perhaps the most serious problem with the preprocessor is that of name capture. All names occur in a single namespace and so every conventional use of a name that is defined as a macro can malfunction. For instance the enumeration

```
struct Options
{
   enum { LEFT, RIGHT, UP, DOWN, MAX };
};
```

should operate quite satisfactorily with `Options::MAX` denoting the number of options. This definition will typically be placed in some include file. However if another include file contains the earlier definition of `MAX`, any reference to `Options::MAX` will fail whenever both include files are used. If the macro is defined before the enumeration, a syntax error will spring up in the enumeration.

This form of error is obscure and confusing. It can appear to be intermittent since compilations that do not use both include files succeed. Novice programmers are baffled. Experienced programmers may take a little time to detect the handiwork of the preprocessor.

This problem is resolved in FOG by changing to a policy of invited substitution. The replacement functionality for `MAX` substitutes only as part of a `$` or `@` expression.

2.4.1.2 Language independence

The independence of macros from the underlying language is achieved in FOG by use of meta-types to constrain the syntax of the values of meta-variables and of the arguments and return of meta-functions. Parentheses are not required.

When a meta-function is invoked, the argument is represented by a parsed AST and eventually validated against the required syntax. The tree is substituted for each reference within the meta-function, and since the tree has already been parsed there is no possibility of re-interpretation in conjunction with surrounding tokens.

2.4.1.3 Side-effects

When a macro such as `MAX` is invoked with an argument that causes a side effect

```
c = MAX(a++, ++b)
```

one of the arguments is evaluated twice, and so receives a double increment. If the first argument is greater, the result is obtained after one increment has occurred. It is unlikely that the program will function as required.

FOG does not resolve this problem. FOG passes the syntax tree for a++ into the meta-function and instantiates the tree for each reference. FOG does not ‘evaluate’ the argument once and then pass it. This is not possible because the argument is a syntax tree, not a value. It may be that the syntax happens to correspond to a value as in the `MAX` case, but not in general.

`MAX` is a poor example for a syntactic meta-function. `MAX` is a computational operation and so can and should be implemented by some form of inlined function.

2.4.1.4 Substitution level

Many implementations of the original Kernighan and Ritchie C preprocessor [Kernighan78] performed character-based substitution. The replacement characters were inserted between the surrounding characters and the resulting character stream was then re-analysed. This offered considerable flexibility, but different implementations varied in their treatment of obscure recursions and encountered difficulties when a composite token such as += arose as a result of character concatenation.

The ANSI C preprocessor changed to token-based substitution. The preprocessor identifies the tokens and substitution replaces a sequence of tokens. Composite tokens can only arise through explicit use of the ## operator.

Although the FOG substitution is syntax-based, character-based substitution is also supported through use of the `std::parse` built-in meta-function and the `token` meta-type[5]. `token` is the most primitive and generic terminal of the FOG grammar. Every number, string, identifier or piece of punctuation such as >>= is a *token*.

```
auto string OCTAL_CASES[] =
    "'0': case '1': case '2': case '3': case "
    "'4': case '5': case '6': case '7'";
```

The required tokens are represented as a concatenation of two strings to fit the available line length. This is then incorporated into its overall context by further string concatenation before `$std::parse` is invoked to convert to a syntax tree.

```
$std::parse(
            "switch (c)"
            "{"
            "   case " $OCTAL_CASES ": /* ... */ break;"
            "   case " $ALPHABETIC_CASES ": /* ... */ break;"
            "   case '%': /* ... */ break;"
            "}"
            );
```

A cleaner solution is available by meta-programming:

```
auto statement multi_case(signed lo, signed hi)
{
    auto for (signed i = $lo; i <= hi; ++i)
        case $i: ;
}
...
switch (i)
{
    $multi_case('0', '9') /*...*/ break;
    $multi_case('a', 'z') /*...*/ break;
    case '%': /* ... */ break;
}
```

This exploits the non-structured definition of a case statement to treat `case 'a':;` as a *statement* that drops through to the next case. The extra semicolon avoids the need for a case meta-type.

```
    case '0': ;
    ... ... ...
    case '9': ;
        /* ... */ break;
```

5. Token-based substitution could easily be supported, but it is difficult to conceive of an example that is not better resolved by syntax-based substitution, or more clearly resolved by character-based substitution.

2.4.1.5 Backslash continuations

The C preprocessor requires all directive lines to comprise exactly one complete line. This restriction is alleviated by the ability to continue a line with a trailing backslash, but results in less readable and difficult to maintain code.

FOG integrates replacement functionality into the main grammar and so the replacement facilities are free format.

2.4.2 Concatenation and Stringizing

The character-based substitution of the K&R C preprocessor enabled composite tokens (normally an extended identifier) to be formed by causing the character sequences to abut.

```
begin/**/_and_/**/end                         // begin_and_end
```

An identifier could be converted to a string by substitution within a macro:

```
#define STRINGIZE(s) "s"
STRINGIZE(text)                               // "text"
```

In the ANSI C preprocessor, the change to token-based substitution, and the requirement that a comment be replaced by a whitespace character, lost this flexibility necessitating the introduction of the ## operator to request concatenation, and the # operator to support stringizing.

```
#define CONC3(a,b,c) a ## b ## c
CONC3(begin,_and_,end)                        // begin_and_end

#define STRINGIZE(s) #s
STRINGIZE(text)                               // "text"
```

In FOG character-based concatenation occurs between characters, numbers, strings, identifiers (including keywords) without intervening whitespace. An empty string or character can be used to provide separation between elements that require concatenation.

```
begin""_and_''end                             // begin_and_end
```

An empty string (or character) can be used as a meta-cast for stringizing. The subsequent text acquires the string (or character) characteristics of the start of the sequence.

```
""text                                        // "text"
```

Alternatively, meta-functions can be defined with similar behaviour to the ANSI C approach

```
auto identifier CONC3(identifier a, identifier b, identifier c)
        { ${a}${b}$c }
auto string STRINGIZE(identifier s) { $s; }

$CONC3(begin,_and_,end)                       // begin_and_end
$STRINGIZE(text)                              // "text"
```

$ invocations in the body of `CONC3` access the formal parameters. The first two invocations use the `${}` form to ensure evaluation as `${a}${b}${c}`, since the default of taking the longest possible interpretation of $a$b$c would evaluate `${a${b${c}}}`.

The stringize meta-function apparently does nothing, however the distinct parameter and return meta-types arrange for an `identifier` to `string` conversion, which also does nothing. The stringizing occurs at the point of usage; when a string representation is required, the internal representation is formatted as a valid string.

2.4.3 #define directive

Preprocessor object-like definitions

```
#define PI 3.14159
```

and function-like definitions

```
#define MAX(a, b) ((a) > (b) ? (a) : (b))
```

are replaced by meta-variable

```
auto double PI = 3.14159;
```

and meta-function

```
auto expression MAX(expression a, expression b)
    { $a > $b ? $a : $b }
```

definitions in FOG. The FOG definitions have syntactical types that can be checked and avoid the need for protective parentheses.

2.4.4 #include directive

The historical semantics of the #include directive permit an arbitrary sequence of tokens to be incorporated more than once in almost any context. This flexibility is excessive and almost never needed. Most programmers have learnt to tame the directive by placing an include file guard within each include file to inhibit multiple inclusion:

```
#ifndef FRED_H_INCLUDED
#define FRED_H_INCLUDED
//...
#endif
```

although many have not learnt how much compile time can be saved by inhibiting the include as in

```
#ifndef FRED_H_INCLUDED
#include "fred.h"
#endif
```

FOG supports a more disciplined form of inclusion like the #import of Objective C [Cox86]:

```
using "fred.h";
```

performs the inclusion provided the file has not already been included. Guards are unnecessary.

2.4.5 #if, #ifdef, #ifndef, #else, #elif, #endif directives

Preprocessor conditional compilation

```
#if defined(UNIX)
    static const char *temp_path = "/tmp/";
#else
    static const char *temp_path = "C:\\Temp\\";
#endif
```

is replaced by the use of meta-statements

```
auto bool unix = $std::get_cpp(UNIX);
if (unix)
    static const char *temp_path = "/tmp/";
else
    static const char *temp_path = "C:\\Temp\\";
```

with built-in meta-functions of the std meta-namespace providing support when necessary.

2.4.6 #line directive

The #line directive is not used in source programs generated by human beings. It provides a very simple but useful mechanism for automatic source code generators to ensure that compilers and debuggers refer to the original source lines rather than some scrambled intermediate. #line performs this role adequately and needs no replacement. Once Cpp has been discontinued, an extension that uses a more cryptic free format spelling and releases the # token could be considered. This is examined in Appendix F.1.4.

2.4.7 #error directive

FOG provides the built-in meta-functions

```
auto nil std::diagnostic(string someText)
auto nil std::error(string someText)
auto nil std::warning(string someText)
```

to support emission of meta-compile-time messages.

2.4.8 #pragma directive

FOG provides no explicit counterpart for pragmas. However the effects of pragmas may be achieved by vendor-defined built-in meta-functions.

A compilation system that supports intrinsic functions might recognise

```
#pragma intrinsic(memset)
```

to request use of the intrinsic rather than function call implementation of memset.

In a compatible meta-compilation system, a built-in meta-function to implement the pragma could be hypothetically declared:

```
auto nil std::intrinsic(identifier anIdentifier);
```

and invoked as

```
$std::intrinsic(memset);
```

Invocation on a meta-compiler without support for the pragma could be dummied out by defining the meta-function to have no functionality:

```
auto nil std::intrinsic(identifier anIdentifier) {}
```

or to redirect to a different pragma:

```
auto nil std::intrinsic(identifier anIdentifier)
    { $std::use_intrinsics(""$anIdentifier); }
```

3 FOG Semantics

Chapter 2 presented the FOG grammar and discussed many of the minor semantic issues, but deferred the more major ones for resolution in this chapter.

We first describe the compilation stages that transform a FOG source program into C++ files, and distinguish between the potential declarations appearing in the FOG source and the actual declarations that are finally emitted. This leads on to a discussion of the meta-types and meta-objects used to represent these declarations, and the type-constructors used to support iterators and trees.

We then describe alternative models for substitution, justify the choice of syntax-level substitution in FOG, and examine the constraints imposed by supporting context free parsing of arbitrary syntax.

The distinction between the conventional name-space used for C++ declarations and the meta-name-space of meta-declarations is examined. Consideration of the context in which substitution occurs shows the need for more than one substitution operator, and a requirement to use lexically nested scopes rather than structurally nested scopes.

The flexibility that FOG provides for redeclarations is a critical distinction between FOG and C++ that is resolved by composing the multiple contributions to a single composite C++ declaration. The composition policies for different categories of declaration are described.

FOG has a simple meta-class model. Other languages have different models. We review the alternate models and show that the FOG meta-class model is a natural and consistent extension of the run-time programming perspective to (meta-)compile-time.

User-defined code may contribute to decisions made at (meta-)compile-time through static meta-programming either through direct invocation from source code or through provision of meta-programs. Meta-programming by direct invocation is discussed in Sections 3.2, 3.3 and 3.4. Independent meta-programs are discussed in Section 3.6.

Finally the behaviour of syntax macros is described.

3.1 Meta-compilation stages

The significant processing stages of meta-compilation are shown in the left-hand column of Figure 3.1.

The right-hand column identifies the distinct representations used, with solid arrows indicating the flow of data between the processing stages. As will shortly be explained, the anomalous data-to-data arrow is the result of automatic creation of an actual declaration from a potential declaration in a determined context.

Chapter 4 describes the successive stages of lexical, syntactic and semantic analysis that steadily increase the precision of the internal representation of the following trivial program

```
    int x;
```

as

- six source characters for `int x;`
- three tokens for *identifier* `int`, *identifier* `x` and *punctuation* `;`
- a syntax tree for the generalised expression `int x`
- a potential declaration for the variable-specifier `int x`
- an actual declaration for the global variable `int ::x`

Actual declarations are emitted to the C++ output files, after construction of the usage dependency graph by the compilation stage as discussed in Chapter 5.

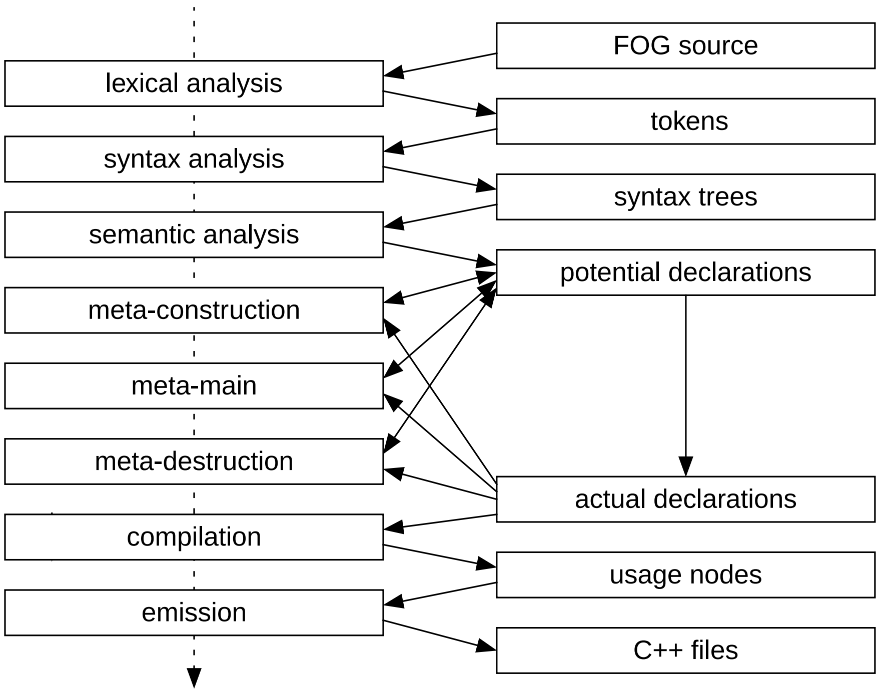

Figure 3.1 FOG Meta-compilation Stages

The distinction between potential and actual declarations is not necessary for C++, since every declaration has a well-defined context. FOG supports passing of declarations to meta-functions, and the regeneration of declarations in derived contexts by derivation rules. Potential declarations therefore describe a declaration after semantic analysis. Transformation to an actual declaration occurs as soon as the declaration is associated with an actual context.

Static meta-programming may occur during

- semantic analysis
- meta-construction
- meta-main
- meta-destruction
- compilation

Meta-programming during semantic analysis occurs through interpretation of meta-statements:

```
auto if (generateDebugCode)
    /* debug code */
```

or more generally to service meta-function calls

```
$CachedString::flyweight("pointer");
```

The meta-construction, meta-main and meta-destruction stages are described in Section 3.6. They consist exclusively of application functionality.

The final opportunity for meta-programming occurs during the compilation stage. At this point meta-functions and meta-variables used within the derivation predicates and bodies of declarations are evaluated, and meta-programming must be restricted to the acquisition of information from the pool of declarations. New declarations cannot be added, although meta-variables can be updated to cache results or maintain context.

3.1.1 Potential and Actual Declarations

In C++, each (non-template) class, function, type or variable declaration provided as source text is analysed in, and contributes once to, its surrounding context. Templates provide a macro facility that enables certain parametric references to be resolved automatically, but each template reference contributes at most once to its surrounding context.

In FOG, derivation rules and more generally meta-programming may cause one source declaration to make multiple contributions to a variety of contexts. Since there is no longer a one to one relationship between source text declarations and implemented declarations, we need to introduce new terminology.

A declaration appearing as source text is a potential declaration. A potential declaration has the potential to be resolved to an arbitrary number of actual declarations.

A declaration resolved to its eventual context is an actual declaration. Actual declarations correspond to the declarations emitted for subsequent compilation by the C++ compiler.

The two forms of declaration have significant differences. Consider an `enum`, which may occur in source text and consequently as a potential declaration for:

- a definition

  enum-specifier:
  `enum` *identifier*$_{opt}$ { *enumerator-list*$_{opt}$ }

- a to be resolved reference

  elaborated-type-specifier: *// Part of*
  `enum` `::`$_{opt}$ *nested-name-specifier*$_{opt}$ *identifier*

both of which are resolved with respect to the prevailing context, whereas an actual declaration has already been resolved. There is no point in instantiating an actual declaration at the source level, since this can only create a duplicate of the resolved declaration. Conversely, it is meaningless to probe the unresolved context of a potential declaration. There are necessarily three distinct concepts to be represented by the meta-type system:

- a potential definition
- a (potential) unresolved reference
- an (actual) reference resolved to an actual definition

The distinctions between potential and actual declarations can be seen in the following example.

```
class Actual {};
auto declaration nest(identifier anId, class_specifier aClass)
{
    class $anId
    {
        $aClass;
    };
}
auto declaration cached = $nest(Nested, class Potential {});
class ReNested
{
    $cached;
};
```

The first potential declaration, for `class Actual`, can be resolved immediately and so results in an actual declaration of `class ::Actual`.

The meta-function definition similarly results in the actual declaration of the meta-function ::nest. Analysis of the meta-function body creates a potential declaration for the *class-specifier* using the first formal parameter anId, for which preliminary semantic analysis can verify compatibility of the identifier meta-type. Syntactic analysis of the *tree-literal* in the *class-specifier* encounters the usage of the second formal parameter. This is presumed to be an *identifier* which is a degenerate *declaration*. Again a preliminary semantic analysis can verify the meta-type compatibility of a class_specifier.

Analysis of the invocation of the nest meta-function creates potential declarations for each argument: the two generic syntax elements Nested and class Potential {}. The first is an *identifier*, for which there is no potential/actual distinction. The second is a *class-specifier* whose context cannot be resolved, so the potential declaration cannot be converted to an actual. Further analysis verifies that the arguments can be converted to satisfy the identifier and class_specifier syntaxes of the formal parameters.

Interpretation of the meta-function replaces the formal parameters and returns the meta-function-body for execution in the calling context as if the source were:

```
auto declaration cached = class Nested
                          {
                              class Potential {};
                          };
```

Further potential declarations for class Nested and class Nested::Potential are created as the initializer for the potential and actual declarations of ::cached.

A potential and actual declaration for class ::ReNested are then created and the cached variable replaced as if the source were:

```
class ReNested
{
    class Nested
    {
        class Potential {};
    };
};
```

so that potential and actual declarations are also created for class ReNested::Nested and class ReNested::Nested::Potential.

3.1.2 Meta-types

Execution of a meta-program involves manipulation of declarations that have some syntactical (meta-)type.

A very simple form of meta-execution could be supported by defining just a single meta-type such as syntax_element, which could exhibit polymorphic behaviour for all possible syntax elements. However source syntax checking is normally a rigorous activity and type-less meta-execution is not philosophically consistent with C++, which is (in principle) a strongly typed language.

FOG should therefore support strongly typed meta-execution. The obvious set of types correspond to the grammar productions: class_name, expression, statement, template_parameter, etc. The set of types provided by FOG will be described shortly.

Casting

C++ supports type-widening but requires some form of cast to narrow a type.

```
class A {};
class B : public A {};
```

```
void f()
{
    B b;
    A *pa = &b;
    B *pb = &b;
    pa = pb;                        // Widening ok.
    pb = static_cast<B *>(pa);      // Narrowing needs a cast.
}
```

This is necessary since it is not generally possible to determine at compile-time that a type-narrowing is safe. Static meta-programs execute at compile-time and so it is always possible to determine safety at compile-time. Precise replication of run-time behaviour at compile-time is therefore unnecessary and undesirable: casts are not available for meta-programming. Assignments are checked dynamically and if possible the assigned value is converted to a value compatible with the target meta-type.

```
auto identifier i = label;
auto statement s = $i;
auto expression e = $s;
```

No error occurs on the assignment of a statement to an expression, because `label` retains its identity as an *identifier*; it is not coerced to a *statement* since an *identifier* is a valid *expression-statement*.

Meta-type as syntax predicate

It is appropriate to regard the meta-type as a syntax predicate, since the polymorphism available at compile-time can be exploited to convert the result of an assignment to the specified meta-type.

In order to keep the FOG grammar context-free, it is necessary to be able to parse the argument of a meta-function, or the initializer of a meta-variable without knowledge of the required syntax (or meta-type). A single syntax must cover all possibilities. This syntax satisfies the weakest predicate of the `token` meta-type, and also the stronger predicates corresponding to the meta-types used to create the AST.

Since each meta-type corresponds to a syntax predicate, a meta-type could be defined for each non-terminal in the FOG grammar.

There are three problems:

Excluding the lexical productions, there are about 150 non-terminals in the C++ grammar (more in FOG).

The non-terminals are not orthogonal. For instance after parsing the source text containing the isolated letter a, what is its syntactic type? It could be an *identifier*, *template-argument*, *expression*, *abstract-declarator* etc. Choosing the more primitive meta-type would be possible if the grammar was completely context-free and so the productions corresponded to nodes of a tree, unfortunately

```
a < b > ( c )
```

could be a *postfix-expression* or a *declarator*, for which *postfix-expression* is a shared common syntax type in the superset grammar implementation of FOG. However there is no shared syntax type in the C++ (or FOG) grammar.

Distinct meta-types are needed to describe potential and actual declarations.

We therefore need extra meta-types to capture the potential/actual distinction, but want fewer of them to reduce the definition and documentation effort. There seems little point introducing meta-types to distinguish between a *logical-and-expression* and a *logical-or-expression.* A more pragmatic approach is taken, defining meta-types only for those concepts for which a meta-program may reasonably need to make a distinction.

Implemented meta-types

Each meta-type corresponds to a non-terminal, whose production rules define the valid syntax for syntax trees with the meta-type. Many non-terminals are specialisation of others, thus, *identifier* is a (specialisation of) *id-expression* which is a *primary-expression* which is an *expression* which is (semantically) an *expression-statement* which is a *statement*. This specialisation hierarchy resembles an Object Oriented inheritance hierarchy, and is implemented as such in FOG.

The UML [Booch99] inheritance diagram in Figure 3.2 shows the externally visible general structure[1] and some of the more significant meta-types.

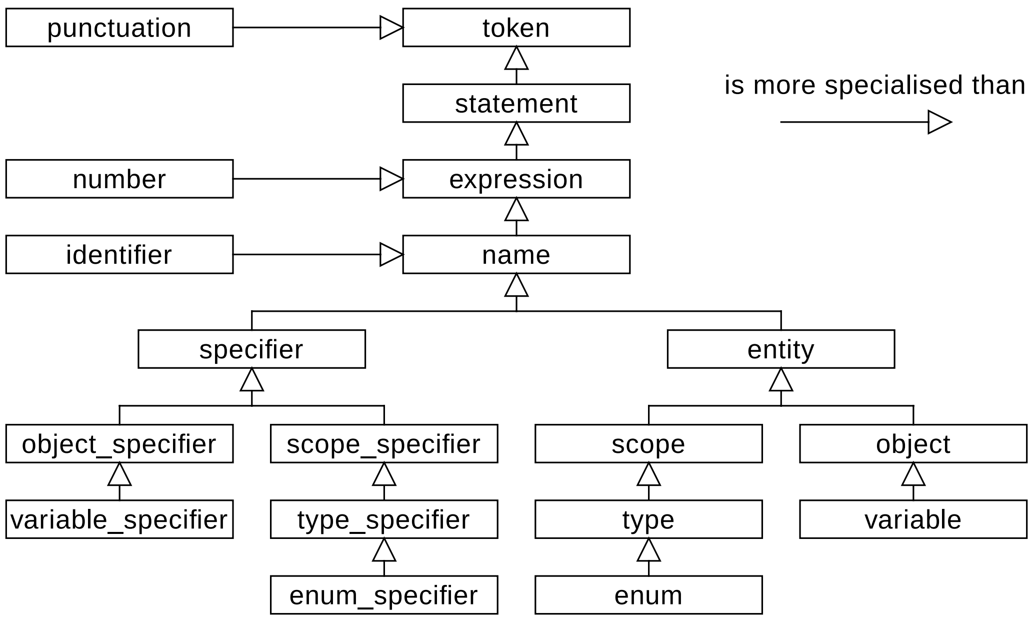

Figure 3.2 Meta-type inheritance

The upper half of Figure 3.2 shows some of the intrinsic meta-types. Intrinsic meta-types define a syntax that can be directly identified and require no distinction between potential and actual declarations. A fuller listing of the intrinsic meta-types is provided in Table 3.1, with specialisation indicated by the partitioning into more specific meta-types across the “intrinsic meta-types” columns.

The lower half of Figure 3.2 shows some potential and actual meta-types. Meta-objects of a potential meta-type more specialised that `specifier` are created by the (syntactic and) semantic analysis to describe potential declarations. These meta-objects are converted to have an actual meta-type more specialised than `entity` once the context supports the conversion from a potential to an actual

1. The internal implementation is very similar, adding a couple of further levels of inheritance for implementation details. The resulting 10 deep inheritance hierarchy violates recommended OO design practices. Earlier implementations of FOG used multiple flatter (but still deep) hierarchies and encountered anomalies through lack of polymorphism between names, potential and actual types and expressions. The superset grammar mandated a very high degree of polymorphism and led to the slightly surprising discovery that a *variable-specifier* is an *expression*. This arises through the treatment of a *decl-specifier* as the decoration of a name, comparable to a *ptr-operator* prefix or array index, or function call suffixes.

Intrinsic Meta-Type				Grammar	
				example	non-terminal
punctuation				`=`	*'punctuation'*
modifier		array_modifier		`.. []`	*ptr-operator* *'declarator-operator'*
		function_modifier		`.. (int x) throw y`	
		pointer_modifier		`*T::Q const ..`	
		reference_modifier		`& ..`	
name	keyword	reserved		`case`	*'reserved-word'*
		meta_type		`function`	*meta-type*
			class_key	`class`	*class-key*
		decl_specifier		`static`	(non-type) *decl-specifier*
			cv_qualifier	`volatile`	*cv-qualifier*
		identifier		`i`	*identifier*
				`X::i`	*declarator-id*
	using_directive			`using namespace X}`	*using-directive*
expression		assignment_expression		`a = b ^ c`	*assignment-expression*
			constant_expression	`5`	*constant-expression*
			character	`'a'`	*character-literal*
			number	`4.0`	*number-literal*
			string	`"string"`	*string-literal*

				`a,b`
				
				`throw (a)`
				`int`
				`x(5)`
					expression
					… … …
					throw-expression
					template-argument
					mem-initializer

statement
declaration
				`label: x;`
				`if (a) ;`
				`while (1) ;`
				`goto label;`
				`try {} catch ... {}`
					labeled-statement
					selection-statement
					iteration-statement
					jump-statement
					try-block

				`a = b;`
				`int a;`
				`template T<Q> a;`
				
				`asm "";`
					expression-statement
					declaration-statement
					explicit-instantiation
					explicit-specialization
					asm-definition

				`{}` `{}` `{}`	*compound-statement* *compound-declaration* *function-body*
handler				`catch (a) ;`	*handler*
exception_specification				`throw (a)`	*exception-specification*

Table 3.1 Intrinsic Meta-types

declaration. A fuller listing of the potential and actual meta-types is shown in Table 3.2 again using partitioning to denote specialisation.

Declarations and Statements

The syntax generalisations in FOG remove the principal distinctions between declarations and statements, however both concepts are widely used and so both `declaration` and `statement` meta-types should be provided. Implementation of these types within a simple inheritance hierarchy requires one to be more specialised than the other. But within a function, a *declaration-statement* is a specialised *statement*, whereas in a class a *meta-expression-statement* is a specialised *declaration*.

The dilemma is resolved from the external perspective by treating the two names as synonyms for the same meta-type. A subtle, and almost irrelevant, distinction

Potential Meta-Type				Grammar		Actual Meta-Type			
				example	non-terminal				
specifier	`namespace_alias_definition`			`namespace X = Y`	*namespace-alias-definition*	`namespace_alias`			`entity`
	`scope_specifier`	`namespace_definition`		`namespace X {}`	*namespace-definition*	`namespace`		`scope`	
		`linkage_specification`		`extern "C" {}`	*linkage-specification*	`linkage`			
		`filespace_specifier`		`namespace/file X {}`	*filespace-specifier*	`filespace`			
		`meta_class_specifier`		`auto class X {}`	*meta-class-specifier*	`meta_class`			
		`type_specifier`	`class_specifier`	`class X {}`	*class-specifier*	`class`	`type`		
			`elaborated_type_specifier`	`class X`	*elaborated-type-specifier*				
				`typename X`		`typename`			
				`enum X`		`enum`			
			`enum_specifier`	`enum X {}`	*enum-specifier*				
			`built_in_type_specifier`	`long int`	*'built-in-type-specifier'*	`built_in`			
	`base_specifier`			`public X`	*base-specifier*	`base`			
	`enumerator_definition`			`E = 5`	*enumerator-definition*	`enumerator`			
	`object_specifier`	`using_declaration`		`using X::y`	*using-declaration*	`using`		`object`	
		`function_specifier`		`void f() {}`	*function-definition*	`function`			
				`void f()`	*simple-declaration init-declarator declarator abstract-declarator member-declarator*				
		`variable_specifier`		`int v = 0`		`variable`			
				`int errno`					
		`typedef_specifier`		`typedef a b`		`typedef`			
		`parameter_specifier`		`int * = 0`	*parameter-declaration*	`parameter`			
		`meta_function_specifier`		`type f() {}`	*meta-function-definition*	`meta_function`			
		`meta_variable_specifier`		`name n = n`	*meta-variable-declaration*	`meta_variable`			
		`meta_parameter_specifier`		`identifier i`	*meta-parameter*	`meta_parameter`			
		`exception_declaration`		`int * = 0`	*exception-declaration*	`exception`			
		`template_parameter_specifier`		`class T = X`	*template-parameter*	`template_parameter`			

Table 3.2 Potential and Actual Meta-types

is made that a `declaration` resolves an ambiguous labeled-statement/anonymous-bit-field to the bit-field, whereas a `statement` resolves to the label.

Internally, each of `declaration` and `statement` is a specialisation of a more abstract meta-type, which replaces the use of `declaration` or `statement` for specialisations of `declaration` or `statement`.

3.1.3 Meta-objects

Potential and actual declarations and their constituent elements are represented by meta-objects within the meta-compiler, and many of these meta-objects are available for manipulation by meta-programs. Meta-objects are created by the presence of the corresponding declaration, and are managed automatically as part of a hierarchical pool of meta-objects maintained by the meta-compiler. There is no need or facility for explicit creation or deletion of meta-objects; there is no meta-`operator new` or meta-`operator delete`. Garbage collection of objects that are not required during the code emission phase occurs automatically.

Each meta-object has a corresponding meta-type, with distinct meta-types for potential and actual declarations. When

```
    enum Enum { E1, E@Second };
```

is parsed to create a potential declaration, a meta-object of enum_specifier meta-type is created with two child meta-objects: an *identifier* of identifier meta-type and a list of *enumerator-definition*s. The list in turn comprises two child objects of enumerator_definition meta-type, each of which has a child meta-object to define the name. The unresolved name of the second enumerator is represented by an expression tree to capture the deferred resolution. The root of the tree has the name meta-type. The meta-types (and some values) for the meta-objects defining the potential declaration are shown in Figure 3.3.

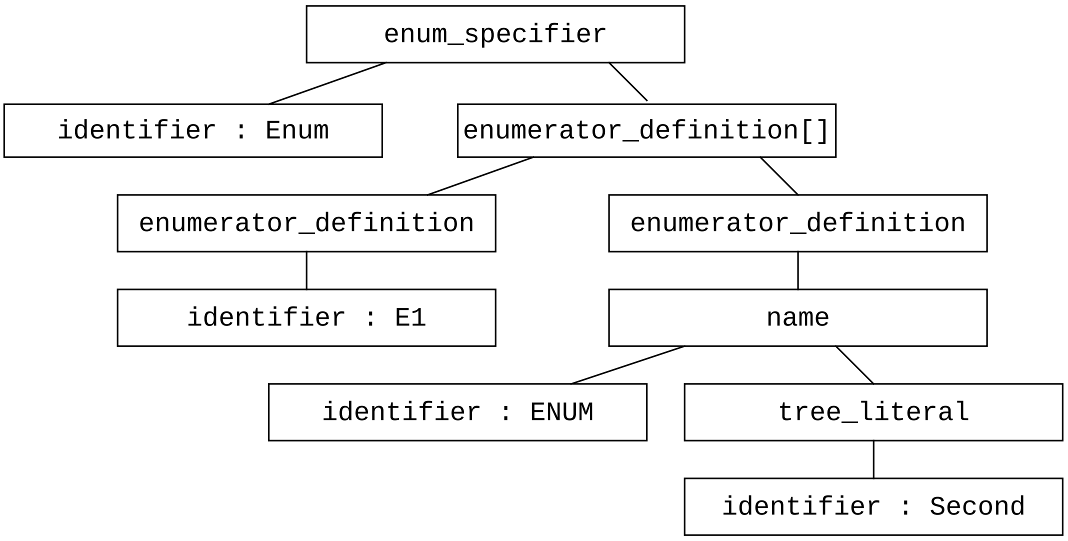

Figure 3.3 Example Potential Declaration Tree

When the potential declaration is converted to an actual declaration, the unresolved name is evaluated in its actual context to yield a meta-object of identifier meta-type, and the enum_specifier and enumerator_definition meta-objects are converted to the meta-objects of enum and enumerator meta-type shown in Figure 3.4.

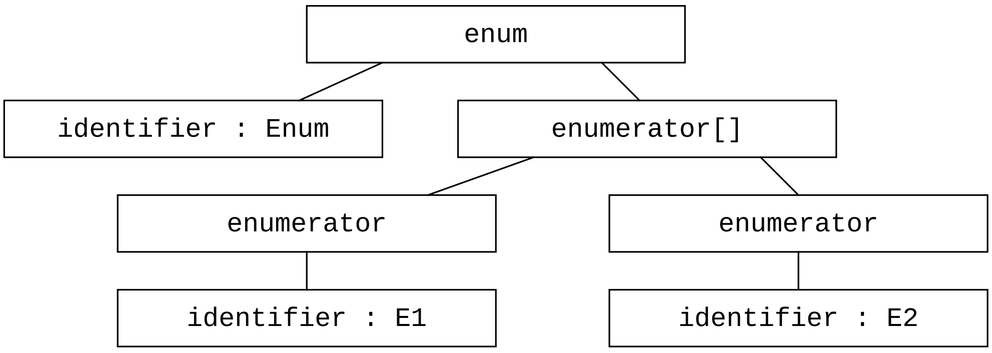

Figure 3.4 Example Actual Declaration Tree

Access to these automatically managed meta-objects occurs by establishing a reference. At the time of creation:

```
    auto enum_specifier potentialDef = enum Enum { E1, E@Second };
    $potentialDef;
```

by subsequent direct reference

```
auto enum_specifier potentialRef = enum Enum;
auto enum actualRef = enum Enum;        // See Section 3.1.6.1
```

or by indirect reference

```
auto enumerator enum2a = $Enum::E2;
auto enumerator enum2b = $Enum.enumerators()[1];
```

3.1.4 Working meta-variables

‘Temporary’ meta-objects may be used for numeric calculations and iterators.

```
class X
{
    auto int i = 0;
    auto for (iterator p = $Enum.enumerators(); p; ++p)
        /* ... */ ++i /* ... */
};
```

The declaration of `i` has no syntactical indication that `i` is a temporary variable, so it isn’t. The meta-variable `X::i` is therefore not temporary.

The iterator is established within a for loop, and so might be expected to be temporary. Again it isn’t. This could be regarded as a limitation, since FOG does not support local blocks for meta-programming. However, if loops did establish local blocks, a syntactic mechanism would then be needed to identify that a declaration in a local block was non-local.

Temporary meta-objects therefore remain as permanent contributions to their prevailing context. This may cause minor problems since re-use of a temporary variable composes the re-use, and re-use with a different meta-type is an error. However, an intended re-use with a changed initializer operates as expected, since the new value replaces the old.

3.1.5 Scalars, Arrays, Lists and Trees

The elemental meta-types and their use for compile-time calculation and syntax validation has been described. Use of scalar types alone is inadequate for meta-programming, since many declarations involve lists of child declarations, and in some cases trees of descendant declarations. Traversal of these structures must be supported.

C++ supports constructed types through pointer-to, array-of, function-returning and record-of type constructors, and perhaps the same type constructors should be available for meta-types.

The C++ pointer system inherited from C is necessary to support uncontrolled memory access and arbitrary memory allocation through the type-less `malloc()`. This lack of discipline causes many problems through the use of null, dangling or stale pointers and allows memory to leak. C++ introduced `operator new` and references and thereby alleviated some of the problems. Java supports only references and consequently has no comparable memory access problems. It is not clear that there is any need for genuine dynamic memory allocation at meta-compile time, since FOG allocates and manages meta-objects automatically. A more pragmatic set of type constructors is therefore implemented, exploiting the freedom to define all meta-types and lists of meta-types as specialisations of the `token` meta-type.

Meta-Array-Of

By imposing the stronger constraint that a meta-type is polymorphic to a list of the same or more specialised meta-type, we may use nested lists to handle arbitrary tree structures. The list has many similarities to a C++ array and so we can safely re-use the nested `{}` syntax for list initialization and `[]` for indexing.

```
auto string s = { "a", { "b", { "c", { "d" }}}};
```

defines a tree with a first, second, third and fourth generation descendant, as shown in Figure 3.5.

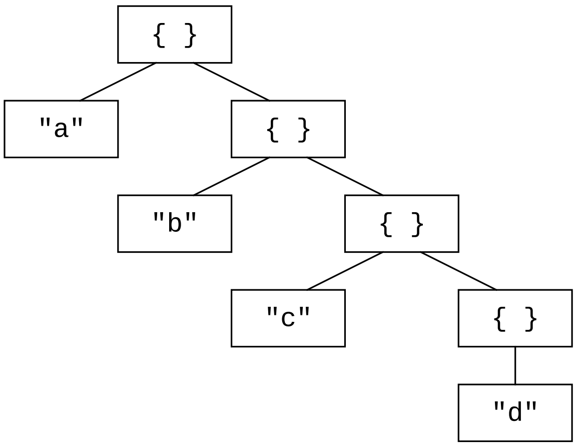

Figure 3.5 Example Tree initializer

`s`	is the element	`{ "a", { "b", { "c", { "d" }}}}`
`s[0]`	is the element	`"a"`
`s[1]`	is the element	`{ "b", { "c", { "d" }}}`
`s[1][1][1]`	is the element	`{ "d" }`

There is no provision for uninitialized meta-variables and so no need to declare tree shapes independent of initializers. The initializers define the shape: explicitly as in the above example, implicitly when a declaration tree is assigned to a meta-variable.

Meta-programs need to be able to iterate over lists of declarations. A polymorphic iterator meta-type is therefore provided that acts as a pointer to a list.

With this system an iterator could be used as:

```
for (iterator i = $bases(); i; ++i)
    i->do_something();
```

`bases()` returns a list of base class specifiers. The iterator is initialized by the elements of the iteration domain, and so `iterator i = $bases()` unfortunately establishes a one element iteration for the list rather than a multi-element iteration for the elements of the list.

An operator is necessary to expose the contents of the list, so that continuing the earlier example

`s[]`	is the two element list	`"a", { "b", { "c", { "d" }}}`
`s[1][1][1][]`	is the elemental list	`"d"`

allowing the iteration to be written:

```
for (iterator i = $bases()[]; i; ++i)   // Non-FOG example
```

The extra `[]` changes the initialization to the elements of the list ensuring an iteration over each base class. This is clumsy and prone to errors, since the `[]` is too easily omitted.

We therefore extend the system to distinguish exposed and encapsulated lists.

An encapsulated list variable comprises a single element, which may be a list.

```
identifier encapsulatedList = { a, b, c, d };
```

An exposed list variable identified by a `[]` declarator suffix comprises the arbitrary number of elements of a list.

```
identifier exposedList[] = a, b, c, d;
```

The {} and [] encapsulate and expose symmetrically, so that

```
    exposedList is-the-same-as encapsulatedList[]
    encapsulatedList is-the-same-as { exposedList }
```

Defining the initialization of an iterator and the return from built-in meta-functions as exposed lists eliminates the need for the clumsy [] and allows the natural programming style:

```
    for (iterator i = $bases(); i; ++i)
        i->do_something();
```

or even

```
    for (iterator i = $A::bases(), $B::bases(); i; ++i)
        i->do_something();
```

to achieve iteration over the concatenation of two exposed lists.

The hypothetical built-in declaration for bases is therefore

```
    auto base class::bases() [];
```

and for iterator initialization:

```
    auto iterator::iterator(token []);
    auto nil iterator::operator=(token []);
```

Meta-Pointer-To

This approach supports scalars, arrays and more generally trees, with all names behaving as references. The limited need for a pointer is handled by the polymorphic iterator.

Meta-Function-Returning

There is no direct support for pointers to functions, however functions and meta-functions can be manipulated under the guise of the function or meta_function meta-types, or more generally as declarations.

```
    auto function_specifier add_constant(number fixedValue)
    {
        inline double ${unique_name()}(double functionArgument)
        {
            return functionArgument + $fixedValue;
        }
    }
    auto function plusTwo = $add_constant(2);
    a = ${plusTwo.name()}(b);              // a = b + 2;
```

add_constant is a meta-function that returns the function-specifier (potential declaration) for an inline function which adds fixedValue to the functionArgument. The name of the returned function is determined by a unique name-generating meta function.

A potential declaration for a function that adds 2 is created by the invocation to initialize the plusTwo meta-variable. Since the required meta-type is for an actual function the inline function is created in the current (global) context.

The name of the function is then used in the final line.

Meta-Record-Of

User defined meta-types are not supported in FOG. This seems a very natural extension when more substantial compile-time programming is required. A little careful thought is necessary to distinguish (or unify) the conflicting perspectives of user-defined meta-types as types constructed from existing meta-types, and as predicates upon extended syntax.

3.1.6 Meta-type conversions

When a meta-object is used as an intermediate term in a meta-expression, the meta-object may be suitable for direct use:

```
$metaObject.meta_function();
auto int i = $metaObject1 + $metaObject2;
```

Alternatively, a conversion to a more suitable form may be required, which may involve:

- conversion of a potential declaration to an actual one
- resolution of the value of a meta-object
- conversion to a character, identifier, number or string meta-type.

3.1.6.1 Potential to Actual meta-type conversion

When a potential meta-type occurs in a context where an actual is required, an actual declaration is created from the potential in the prevailing context.

```
auto enum_specifier potentialEnum = enum E;// No conversion
auto enum actualEnum = enum E;              // Creates ::E
```

3.1.6.2 Resolution of the value of a meta-object

A *tree-literal* defines an expression to be resolved in the meta-name-space resulting in a meta-object whose value replaces the *tree-literal*. The replacement value must be a meta-object that has meaning beyond the meta-level; it must therefore be a *character-literal*, *identifier*, *number-literal* or *string-literal*. This value is obtained by invoking the built-in conversion meta-function, `operator identifier()`, unless the *tree-literal* forms part of a concatenation in which case one of `operator character()`, `operator number()` or `operator string()` may be invoked instead.

For the `character`, `identifier`, `number` and `string` meta-objects these conversion meta-functions perform conversions as described in Section 2.1.1.3.

For named meta-objects, the unqualified name is first expressed as an `identifier`, and then further converted if necessary. This supports the idiomatic

```
const char *class_name = ""$Scope;
```

rather than the more explicit

```
const char *class_name = ""$name();
```

or even more explicit

```
const char *class_name = ""$Scope.name();
```

Meta-variables, meta-iterators and meta-functions have no run-time object and so the appropriate conversion operator is applied to their `value`. The value of a meta-function is the function body.

The value of a meta-iterator is a boolean meta-object, whose value is true while the iterator is valid. This supports:

```
auto for (iterator p = $functions(); p; ++p)
    /* ... */
```

3.2 Substitution

Substitution is triggered when a reference such as

```
$name
```

is detected that satisfies the *tree-literal* syntax. The reference clearly cannot be fully resolved until the definition is available, and provision of that definition requires that the semantic processing of the definition has completed.

The C preprocessor has an independent definition syntax using `#define` that can easily be fully analysed before processing continues. In FOG, the definition syntax is integrated with the rest of the language, where alternative substitution semantics impose distinct requirements on the overall processing.

3.2.1 Substitution levels

Substitution of a segment of source code by a replacement is often called macro processing. A variety of models and classifications of macro processing are discussed in Section 7.1.2. The most appropriate for discussing substitution in FOG is the distinction between character-, token-, and syntax-level substitution.

These alternate levels will be considered while attempting to implement the following example meta-program:

```
auto for (unsigned i = 0; i < 10; ++i)
    static const char *digits[] = { ""$i };
```

The example involves repeated interleaved definition and reference of a meta-variable. The statement should be equivalent to the C++ declaration:

```
static const char *digits[] =
    { "0", "1", "2", "3", "4", "5", "6", "7", "8", "9" };
```

In each iteration, the identifier `i` must be resolved to the appropriate meta-variable, and evaluation must use the prevailing value of the meta-variable. (In most examples, such as this one, `i` resolves to the same meta-variable on each iteration, however this cannot be assumed.)

3.2.1.1 Character-level substitution

Character-level substitution supports replacement of a text macro by an arbitrary run of characters, allowing perverse programs such as:

```
auto string s = "= 5;";
x +$s               // x += 5;        (Not valid in FOG)
```

The original K&R C preprocessor used this form of substitution.

Character-level substitution requires lexical analysis of the replacement to be (at least partially) deferred until the adjoining characters are available, and so requires the lexical analysis to be repeated in each substitution context. Therefore preliminary syntactic analysis of the example *iteration-statement* cannot identify more structure than:

```
auto for (unsigned i = 0; test-chars; step-chars)
    body-chars
```

The preliminary analysis of the *xxx-chars* character sequences for each of the three sub-clauses must invoke a special scanner that is sensitive only to syntax involving the structuring tokens: `( ) { } ; if else do while`. Resolution of replacements cannot occur immediately, or even in lexical order, since the replacement for use in *step-chars* is not necessarily known until *body-chars* has been interpreted.

This imposes the restriction that replacement character streams involving structuring tokens may behave unpredictably, since the assumptions about embedded punctuation may be unjustified.

Once the three character sequences have been cached, interpretation may then proceed by repeated invocation of lexical, syntactic and semantic analyses for each of the cached *test-chars*, *body-chars* and *step-chars*. Eventually the interrupted lexical analysis may resume, following on from the `for` statement.

Since character-level substitution during lexical analysis requires semantic interpretation, it is necessary to perform each of the lexical, syntactic and semantic analyses re-entrantly for each level of statement nesting.

An early version of FOG used this form of substitution, concurrently with Cpp substitution at the cost of considerable implementation difficulty and rather ill-defined behaviour.

FOG now supports explicit invocation of character-level substitution from the semantic analysis, rather than re-entrant semantic analysis within the lexical analysis. The perverse example can be realised as

```
auto string s = "= 5;";
$std::parse("x +"$s);
```

3.2.1.2 Token-level substitution

Token-level substitution performs substitution after lexical analysis has completed the conversion of source text into a token stream. This improves efficiency and provides for a more predictable environment but loses flexibility.

Operation at the token-level prevents the formation of composite punctuation tokens such as +=. This is probably beneficial. However, losing the ability to create extended identifiers and the ability to convert character sequences to strings is restrictive. The ANSI C preprocessor uses this substitution model and introduces the # and ## operators to provide a slightly clumsy way around the restrictions.

Token-level substitution prohibits syntactic analysis of the replacement tokens in isolation, since adjacent tokens may influence the meaning. Syntactic analysis must be repeated for each replacement and error diagnostics associated with instantiation rather than definition.

Token-level substitution simplifies the example only slightly, requiring the preliminary parse to resolve:

```
auto for (unsigned i = 0; test-tokens; step-tokens)
    body-tokens
```

The special scanner now caches token sequences rather than character sequences, but is still vulnerable to unexpected replacement behaviour that involves the structuring tokens.

Re-entrant invocation of syntactic and semantic processing is again necessary to interpret the cached token sequences during syntactic analysis of the loop.

A form of this token-level substitution was used by the multi-pass grammar implementation of FOG, which performs syntax-driven re-entrant analysis. The difficulty of defining the token `{` (for use in a replacement) rather than the punctuation `{` (for defining program structure) was at one point assisted by introducing the additional lexemes `\{` and `\}`. This fudge was then replaced by an identifier to punctuation conversion fudge in the phase 7 tokenization grammar. Neither fudge is required in the next approach:

3.2.1.3 Syntax-level substitution

Syntax-level substitution uses a syntax tree from an earlier syntax analysis as the replacement. The replacement is therefore constrained to satisfy an explicit syntactical type, rather than the much weaker constraints of sequence-of-character or sequence-of-token.

Syntax level substitution

- further improves efficiency since no source code experiences repeated lexical or syntactic analysis
- supports improved error diagnosis since the definition of a replacement can be checked syntactically
- avoids the syntactical interpretation changing in response to its instantiation context (the parenthesis problem of Section 2.4.1).

The informal restriction on the occurrence of structuring tokens is removed, or rather becomes intrinsic to the substitution model; it is difficult to define structuring tokens as replacements, since partial syntax does not satisfy syntactical requirements.

Direct syntactic analysis of the entire example is supported by the creation of unresolved reference-to-`i` nodes that act as place-holders for resolution during each iteration of the loop. With syntax-level substitution it is possible to complete syntactic analysis of the entire program before any semantic analysis starts. Re-entrancy occurs, but only within the confines of the semantic analysis.

The superset grammar implementation of FOG uses syntax-level substitution and avoids all context dependencies. Substitution below syntax-level is supported by deferred character-level substitution:

```
auto string braced_else = "} else";
$std::parse("if (a) { b; " $braced_else " c;");
                                    // if (a) { b; } else c;
```

Although this supports dubious concatenation, the prevailing syntax type must always be syntactically valid. The mismatching `}` preceding the `else` is only mismatched while represented as a `string`. Conversion to a useful syntactic element must wait until string concatenation provides valid source for `std::parse` to convert to a generic syntax element: in this case a *selection-statement*.

For the C preprocessor, there is no looping and so references to replacements necessarily occur after their definitions. There is no flexibility in the location of a resolved definition. Replacements occur directly, with complexity arising only for formal parameters of function-like macros. The same is not true for FOG.

While syntax-level substitution is efficient and predictable, it inhibits character-level and token-level substitution. In practice, many substitutions would satisfy the constraints of an invariant definition lexically preceding a reference from a determined context. It is tempting to allow such backward reference substitutions in which replacement characters or tokens could be substituted while the source was at character or token level. However this would be a special behaviour that could change unexpectedly as a program evolved. Provision of `std::parse` to allow an explicit recursion from semantic level back to the lexical level seems to satisfy the potential requirements for lower level substitution without introducing irregular behaviour. Other requirements are satisfied by the concatenation mechanism.

3.2.1.4 Semantic-level substitution

The three preceding levels of substitution correspond to different degrees of validity for the definition of a replacement:

A character-level replacement is lexically indeterminate:

- an arbitrary sequence of valid characters

A token-level replacement is lexically valid but syntactically indeterminate:

- an arbitrary sequence of valid tokens

A syntax-level replacement is syntactically valid but semantically indeterminate:

- an arbitrary tree satisfying a valid syntax

A fourth semantic-level substitution is possible using a replacement that is semantically as well as syntactically valid. This level is not possible in FOG, since semantic analysis requires type information which may come from the surrounding context. The replacement cannot be semantically analysed in isolation. Overall semantic validity is determined where the replacement is used, although some checks can be made at the definition site.

3.2.2 Syntactic Polymorphism

Replacement of syntactically consistent trees allows some rather dubious programs to be written:

```
auto expression decl0 = i;
auto declaration decl1 = int i;
for (unsigned j = 0; j < 2; ++j)
    $decl$j = k;
```

On the first iteration `decl0` is selected and so the loop body is an expression:

```
i = k;
```

On the second iteration using `decl1`, the loop body is a declaration:

```
int i = k;
```

The variation in syntactical type is legal, since there is no requirement for uniformity.

The declaration form is only legal in FOG because `int i` satisfies the generalised naming that resolves `int i` as a name before assigning k to it. This is inconsistent with the exposition of the C++ grammar in the standard, for which `i = k` is resolved as an *init-declarator*, before the `int` prefix is applied as the *decl-specifier-seq* of a *simple-declaration*.

The ordering in the FOG grammar is necessary to support context free syntactic analysis. The FOG grammar should therefore be expressed in a way more closely resembling the practical implementation in Appendix C rather than the changed C++ grammar of Appendix A.

3.2.3 FOG substitution

The C++ reference model (§2.1) provides for each of the minor translation phases to be performed in sequence. However, the many responsibilities of phase 7 (the main compilation phase) are just bundled together, reflecting the apparent need for syntactic and semantic analyses to be tightly coupled.

In FOG, resolution of a *tree-literal* is detected during the phase 6 (string concatenation) processing but requires the use of phase 7 syntactic processing to identify the *tree-expression* syntax and any meta-function arguments. FOG originally used syntax-driven parsing to resolve each argument according to its known syntax. This imposed tight semantic coupling and required deferred analysis in contexts where the syntax was initially unknown. FOG is now context-free and so no syntactical knowledge or deferral is required.

The early translation stages naturally operate in a demand-driven fashion with the phase 7 syntactical analysis, in the main grammar, making repeated requests to a phase 6 procedure to provide the next token for syntactic analysis. When phase 6 detects a *tree-literal*, phase 7 processing is activated re-entrantly to acquire and analyze the source tokens of the *tree-literal*. These are returned to the interrupted phase 7 processing as a *tree-literal* token, which describes the unresolved *tree-expression*. Recognition of *tree-literal*s may occur re-entrantly to arbitrary depth.

FOG therefore introduces a need for re-entrant invocation of phase 7 syntactic processing, but achieves an overall simplification through the separation of syntactic and semantic processing. All syntactic processing can be completed before any semantic processing starts[2].

This is important since, in general, semantic information is not available. When the *tree-litera*l is encountered in the meta-function body of:

```
auto declaration defer(class clientClass)
{
    ${clientClass}.deferred($Scope);
}
```

it is only known that `clientClass` identifies a class. It is not known which class, so the existence or signature of the `deferred` meta-function cannot be determined, and the required syntax type of the parameter is unresolvable. The invocation context is unknown, so the syntactical type of the $Scope argument is also unknown. Syntax-driven parsing is not possible when the syntactical type of either a formal parameter or an actual argument cannot be determined. In the example, neither can be known. The FOG grammar must therefore be context-free. The example is resolved by initially parsing only for a generic syntax element. Eventually the syntax element is used in a deterministic context where semantic interpretation ensures compliance with the syntax types.

tree-literal definition

Definition of a *tree-literal* in one context, generally involves names that are resolved in another context.

```
auto expression e = a(*b);
```

The initializer `a(*b)` is therefore parsed against the generic *tree-statement* syntax, which accepts almost any C++ sentence, including many that are not *expression*s. Association of the initializer with the meta-variable performs only a weak semantic validation to verify that the initializer could satisfy the syntax of an *expression*. It does not matter that the initializer could satisfy more than one syntax: a function call or a function parameter with a redundantly parenthesised parameter.

When the meta-variable is used in a determined context:

```
class X
{
    typedef int b;
    class $e;
};
```

two errors can be detected. First that the value of e does not satisfy a strong semantic check of the syntax of an *expression*, and secondly that this particular expression is not a valid name for a class.

The first of these errors can be subverted by use of a syntactically weaker meta-type.

```
auto token f = $e;
class $f;
```

The original initializer is checked to verify that it could satisfy the syntax of a *token* (which everything satisfies). When `class $f` is checked, the strong check then detects that this particular token is not a valid name for a class.

tree-literal syntactical analysis

The deferred invocation example

2. Semantic processing of syntax macros must occur at the syntactic level.

```
auto declaration defer(class clientClass)
{
    ${clientClass}.deferred(@Scope);
}
```

showed that the syntactic type of *tree-literal*s is not necessarily known during syntactic analysis of sentences involving *tree-literal*s. This would appear to preclude syntactic analysis of the meta-function body at the definition site. Even if the syntax type could be known, its use introduces a tight coupling between the semantic analysis that determines syntax types and the syntactic analysis that uses these types.

Revising the C++ grammar to incorporate indeterminate contributions is not possible. The result would be totally ambiguous:

```
$a $b $c $d ;
```

could be a *namespace-alias-definition*:

```
namespace x = y ;
```

amongst very many alternatives.

A useful compromise leaves the C++ grammar almost untouched, but supports most practical replacements. The compromise makes the simple assumption that each occurrence of a *tree-literal* represents an *identifier*. Syntactical analysis and the grammar is therefore only affected by the categorisation of a *tree-literal* as an *identifier*.

This assumption imposes minor limitations. *tree-literal*s cannot be used to source keywords or punctuation where these define program structure. The use of $a and $c for the *namespace-alias-definition* example is therefore impossible.

The assumption is much less restrictive than it might appear. It does not require the *tree-literal* to be an *identifier*, merely to be used where an *identifier* could be used.

identifier is a degenerate sentence for many syntaxes:

- a type name (since there is no type/non-type discrimination)
- any form of expression (*identifier* is a *primary-expression*)
- a *parameter-declaration(-clause)* (*identifier* is a type)
- a *decl-specifier(-seq)* (*identifier* is a type)

identifier `;` is a degenerate sentence for most other syntaxes:

- any form of statement (*identifier* `;` is a degenerate *expression-statement*)
- any form of declaration (*identifier* `;` is a degenerate *simple-declaration*)

It is only for very narrow syntaxes involving just reserved words or punctuation that a substitution cannot be allowed, for example:

- *access-specifier*
- *class-key*
- *cv-qualifier*
- *function-specifier*
- *operator*
- *ptr-operator*
- *storage-class-specifier*
- *unary-operator*

These limitations do not prevent definition and usage of these syntactic types, however they do prevent parameterisation of syntax that uses them. For instance, a `class`/`struct`/`union` tag cannot easily be used directly:

```
    auto class_key ClassKey = struct;             // Ok
    $ClassKey Class { /* ... */ };                // Illegal
```

Some of these restrictions can be worked around by using a wider syntactical type such as *decl-specifier* rather than *cv-qualifier*. Others could be worked around by relaxing the assumption that all *tree-literal*s are *identifier*s so that a functional cast can specify the built-in meta-type.

```
    $class_key($ClassKey) Class { /* ... */ };
```

Restrictions involving punctuation or reserved words acting as punctuation cannot sensibly avoided. There is no point introducing single-valued ‘casts’ to reserved words or punctuation, when the reserved words or punctuation could be more easily used directly:

```
    $'namespace'($a) $b $'='($c) $d;    // Silly non-FOG example
```

3.3 Name Resolution

In C++, there are three categories of name-space (as distinct from namespace).

- the hierarchical name-space of (run-time) program declarations
  - template-names
  - type-names
  - non-type-names
- the per-function name-space of labels available for use by `goto`s.
- the C preprocessor macro definition name-space

FOG adds the

- hierarchical meta-name-space of (compile-time) program declarations.

FOG removes the distinction between template, type and non-type names at the meta-programming level, since meta-programming may occur before such distinctions are possible. The distinction is preserved where C++ semantics apply.

The C++ name-spaces are used for resolving names of conventional and extended C++ declarations, and for the scopes of meta-names.

The meta-name-space is explicitly used when resolving *tree-expression*s

```
    $Scope                       @bases()[0].is_virtual()
```

and implicitly by the left-hand side of meta-expressions (within meta-statements)

```
    auto if (meta_variable) /* ... */;
```

Resolution of a *tree-expression* involves four phases

- location of an object in the run-time name-space
- location of the meta-object that describes it
- evaluation of expression operators that use meta-objects
- conversion of the resultant meta-object to suit the invoking context

A meta-object for use in *tree-expression* is identified by a *meta-scoped-id*

meta-id:
    id
    meta-type
    `auto`

meta-nested-id:
    meta-id
    ~ *meta-id*
    meta-id `::` *meta-nested-id*

meta-scoped-id:
    `::`$_{opt}$ *meta-nested-id*

The names are resolved in the conventional name-space, with the minor generalisation that meta-classes such as `unsigned int` or `auto`, which have no conventional classes are treated as having empty classes.

Thus, considering the meta-objects in Figure 3.6, the tree-literal

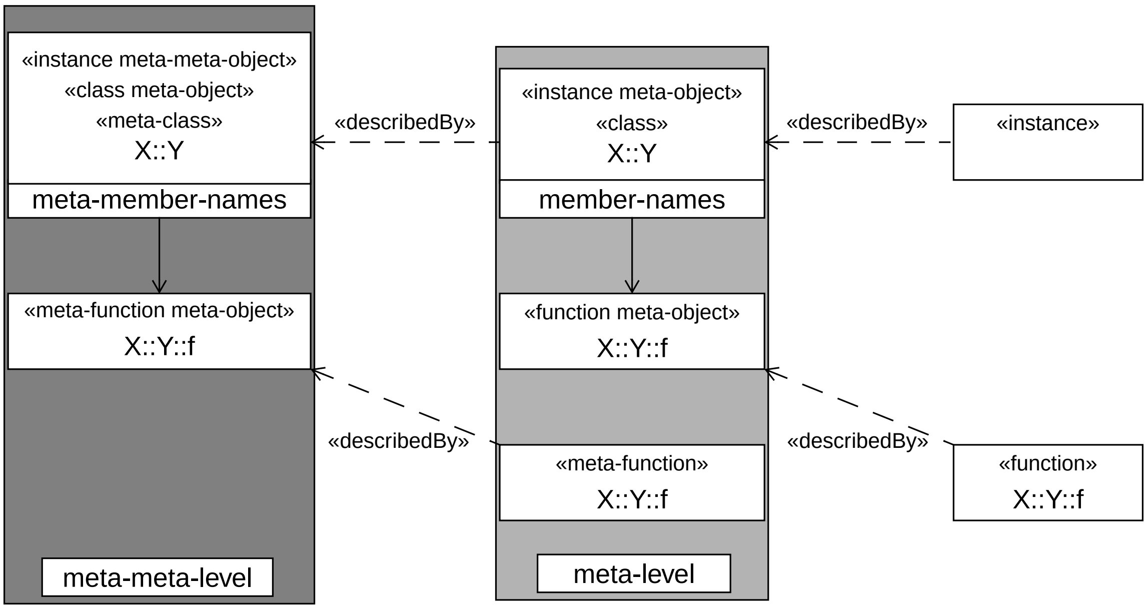

Figure 3.6 Meta-function and function meta-object distinction

```
$X::Y::f.is_static()
```

is resolved by successively locating `X`, `Y` and `f` using the conventional name-space to identify the meta-object that describes (the function) `X::Y::f`. Invocation of `is_static()` upon this function meta-object tests whether the function `X::Y::f` has a `static` qualifier. Whereas in:

```
$X::Y.f.is_static()
```

`X`, `Y` are successively located using the conventional name-space to identify the meta-object that describes (the class) `X::Y` (not instance, since `X::Y` is a class-name not an instance-name). Selection of the `f` member selects the child meta-function meta-object named `f`. Invocation of `is_static()` upon this meta-object tests whether the meta-function `X::Y::f` has a `static` qualifier.

These two mechanisms can coexist without conflict, since the meta-object describing a function is not a meta-function. This is shown in Figure 3.6 and discussed in greater detail in Section 3.5. The meta-objects of the members of a class are not members of their corresponding meta-class. Therefore `::` always performs resolution of references in the class/namespace hierarchy, and `.` or `->` are resolved within the prevailing name-space. When defining a meta-variable or meta-function, `::` rather than `.` may be used for the final scoping:

```
auto int Class.meta_variable = 0;       // Correct
auto int Class::meta_variable = 0;      // Also correct
```

Note that, as might be expected,

```
$v.is_static()
```

is a short form of

$*the-current-context*[3].v.is_static()

so that `v` is located in the meta-object describing the current context. The meta-object describing `v` is not the start of the resolution.

The meta-name-space is used for meta-class, meta-function and meta-variable names. It is not used for meta-function arguments or the right-hand side of a meta-assignment, which, in the absence of further $ or @ operators, represent a literal AST to be interpreted in the usage context.

Search for a name is always restricted to the expected name-space, however it should be noted that template parameters are present in both name-spaces.

A $ or @ prefix is necessary to access the meta-name-space when the normal run-time name-space is used.

3.3.1 Search Locations

C++ has complicated rules (§3.4) for determining when and where names should be resolved:

A name may be resolved within

- the enclosing local block hierarchy
- the local object
- a parameter list
- the local class
- the base class hierarchy
- the enclosing (structural) class hierarchy
- the enclosing (structural) namespace hierarchy

FOG adds further complications. The resolution of a name may be available within

- the meta-name-space
- the enclosing lexical hierarchy
- the root or derived scope of an actual declaration
- the definition or invocation context of a meta-function

The C++ name resolution rules are complicated, and so arbitrary extension to cover new possibilities would cause considerable confusion. Resolution of names within different name-spaces in conventional contexts is therefore signalled explicitly by the @ or $ operators.

3.3.2 Meta-name-space contents

Resolution of a name within the conventional run-time name-space proceeds according to the rules defined in the C++ standard. Resolution within the meta-name-space is triggered by the use of a $ or @ prefix to establish a *tree-literal*, or by the use of a *meta-scoped-id* on the left-hand side of a meta-expression.

Resolution of a name within the compile-time meta-name-space proceeds according to similar principles as run-time resolution, but without the context-dependent visibility of type-names. All names in a meta-name-space context are visible to all searches of that context. The search for a name proceeds in stages, with different candidate sets of locations considered in each stage. The search terminates at the end of the first search stage in which a definition is found, or when all stages have been completed. It is an error for more than one distinct resolution to be found in a search stage.

The search stages are in order:

3. *the-current-context* is `This`, and so `$This.v.is_static()` may be used. However, it is still necessary to use the meta-object describing the current context to locate `This`, since `This` is defined only in the meta-name-space.

Local object-scope

The search for a meta-name in a meta-object may find (using a function as an example):

- meta-functions declared in an *object-statements-clause*
- meta-variables declared in an *object-statements-clause*
- the built-in meta-variables of the `function` meta-type
    - `This` resolving to the meta-object of the function
- the built-in meta-functions of the `function` meta-type

Meta-class

The search for a meta-name in a class meta-object occurs either through use of the class, or through failure to resolve the name in a local-object scope. The search for a meta-name in a class meta-object may find:

- meta-functions declared in a *class-specifier*
- meta-variables declared in a *class-specifier*
- the built-in meta-variables of the `class` meta-type
    - `This` resolving to the (meta-)class
    - `Scope` resolving to the (meta-)class
    - `OuterScope` resolving to the less nested (meta-)class
- the built-in meta-functions of the `class` meta-type
- the formal template parameters.

Search of a class meta-object does not find

- the class name or base class names
- nested types (classes, enums or typedefs)
- member functions
- member variables
- enumerators

or their meta-objects.

Meta-base-classes

If a meta-name is not found directly in a meta-class, the search proceeds recursively by searching each meta-base-class and recurses for each of the base classes, stopping individual searches when a name is found or after the root meta-base class has been searched. Repeated resolutions with the same value are discarded. Ambiguous resolutions are an error.

Less nested meta-base-classes

If a meta-name is not found in a meta-class or its meta-base-class, the search is repeated successively for the meta-classes of each hierarchically less nested class stopping once resolved, or when the namespace scope is reached.

Meta-namespace

The search for a meta-name in a meta-namespace may find:

- meta-functions declared in a *namespace-body*
- meta-variables declared in a *namespace-body*
- the built-in meta-variables of the namespace meta-type
  - `This` resolving to the (meta-)namespace name
  - `Scope` resolving to the (meta-)namespace name
  - `OuterScope` resolving to the (meta-)namespace name
  - `Namespace` resolving to the (meta-)namespace name
  - `OuterNamespace` resolving to the less nested (meta-)namespace name
- the built-in meta-functions of the namespace meta-type

Less nested meta-namespaces

If a meta-name is not found in the namespace, the search is repeated successively for each structurally less nested namespace stopping once resolved, or after the unnested global namespace has been searched.

The built-in functions and variables for the meta-types are summarised in Appendix E.

3.3.3 Derived context resolution

A declaration specified using a derivation rule may need to use names referring to the root scope or the derived class. The following example (elaborated in Section 6.1.4) defines a `clone` method for a `RootClass` and a derivation rule to implement the protocol automatically for all derived concrete classes.

```
public virtual $Scope *RootClass::clone() const = 0
:{
    derived(!is_pure())
       { new @{Scope}(*this); }
};
```

The body of the `clone` method must refer to the derived class, but the return type must use the root scope (on pre-standard compilers).

The two requirements are resolved by the different substitution operators.

A $-expression is resolved within the defining context of a declaration. This is `RootClass` in the above example.

An @-expression is resolved as late as possible, which is generally within the actual context of a declaration. This is the derived class in the above example. ‘As late as possible’ may arise at three distinct times, depending upon the context of the @-expression:

@-expression in scope

An @-expression forming part of the scope of a declaration is resolved when the declaration is interpreted in a determined context in order to install the declaration in the determined scope. This applies to each @-expression in:

```
class @Outer
{
   int @Inner::i;
};
```

There is no distinction between a (single $) $-expression and an @-expression for this form of usage that creates an actual declaration. However a distinction does arise when the conversion to actual is deferred; the @-expression is resolved in the actual context, whereas a $-expression is resolved in the defining context.

```
auto class_specifier cachedClass = class @Outer
                                   {
                                       int @Inner::i;
                                   };
```

@-expression in name

An @-expression occurring as part of the naming of a declaration is resolved when the signature is resolved to install a declaration in its actual scope. Resolution of the @-expression may then respond to any change of scope caused by derivation rules, or meta-programming. The naming comprises the name, types and parameter names, but not default initializers. (Parameter names do not form part of the unique signature of a declaration.) This therefore applies to each @-expression in:

```
@Return Scope::accept_@{Scope}(@Scope& @visitorName);
```

The lexical concatenation of accept_ and @{Scope} although syntactically analysed as an *identifier* remains unresolved until the declaration is installed in its actual scope.

@-expression in body

An @-expression occurring as part of the body of a declaration is resolved when the body is compiled in the final compilation phase prior to emission. This therefore occurs after all meta-programming, except that caused by other @-expressions in declaration bodies. The body of a declaration comprises

- the body of a function
- the handler of a *function-try-block*
- the initializer of a *ctor-initializer*
- the parameters of *exception-specification*s
- the initializer of a variable
- the default-initializer of a function argument
- the predicate of a derivation rule

```
static int X::member_variables = @variables().size();
static bool X::f(int i = @default_init())
:{
    derived(@predicate()) { return i == @threshold(); }
};
```

Thus:

```
protected virtual @{Scope}::~@{Scope}() :{ derived(true) {} };
```

enforces protected access on the destructor throughout a class hierarchy. In this case, since the scope must be resolved in order to identify the root class, the first @Scope is equivalent to $Scope. The second @Scope is resolved in the derived context.

Note that:

```
${Scope}::@{Scope}(const $Scope&)
    :{ derived(true) { /* ... */ } };
```

is a copy constructor in the root scope where @Scope and $Scope are the same class, but an ordinary constructor in derived classes. The final $Scope must be replaced by @Scope to specify a copy constructor throughout the class hierarchy.

3.3.4 Lexical scope resolution

In C++, there is little need to use lexical nesting for name resolution since structural nesting is substantially the same.

```
class Outer                         // ::Outer
{
public:
    /*...*/ name /*...*/;           // ::Outer::name
    class Inner                     // ::Outer::Inner
    {
        /*...*/ name /*...*/    // resolved in structural scope.
    };
};
```

Resolution of name searches in `::Outer::Inner`, then `::Outer` (and finally `::`).

The FOG generalisation to support interleaved declarations and arbitrary scopings means that a lexically enclosing scope is not necessarily a structurally enclosing scope.

```
class Outer                         // ::Outer
{
public:
    /*...*/ name /*...*/;           // ::Outer::name
    class ${Namespace}::Sibling     // ::Sibling
    {
        /*...*/ $name /*...*/       // unresolved.
    };
};
```

Resolution of $name searches in `::Sibling` and `::`, but not in `::Outer`, so the definition is only visible when explicitly qualified as `$Outer::name`. The lack of visibility introduces significant problems for a meta-function such as:

```
auto declaration create_sibling()
{
    /*...*/ name /*...*/;
    public class ${Namespace}::Sibling
    {
        /*...*/ $name /*...*/
    };
}
```

The meta-function establishes some declarations, such as name, in its invocation context. These should be visible to other declarations in lexically nested contexts. The surrounding context is available as $Scope, which could be used to explicitly qualify the reference, except that the reference occurs within a lexical nesting that redefines Scope. Caching the value of Scope in a temporary meta-variable across the lexical boundary is possible but hardly elegant.

A solution is provided by extending the $ operator.

$x resolves x in the prevailing context.

$$x resolves x in the surrounding lexical context.

$$$x resolves x in the surrounding surrounding lexical context.

etc.

The meta-function can therefore be written in re-usable fashion as

```
auto declaration create_sibling()
{
    /*...*/ name /*...*/;
    public class ${Namespace}::Sibling
    {
        /*...*/ $$name /*...*/
    };
}
```

3.3.5 Defining or invoking resolution

Name references within C++ functions normally occur as if resolved within the run-time execution context. With the exception of virtual functions, these resolutions actually occur at compile time, and therefore make use of the defining context which is known at compile-time.

Name references within FOG meta-functions occur when the meta-function defined in a (definition) context is invoked from another (invocation) context. Both contexts are known at compile-time, and since meta-programming is manipulating declarations, resolution within the invocation context as well as the definition context can be useful. The semantics of meta-function execution is that formal parameters are replaced and then the body of the meta-function is returned for incorporation as part of the invocation context. This ensures that the invocation context can be manipulated but prevents the definition of a meta-function reacting to its defining context. The following example attempts to conditionalise the behaviour of a meta-function in accordance with a DEBUG command line variable:

```
auto bool debug = $std::get_cpp("DEBUG");

auto declaration f()
{
    if ($debug)                 // Not the way to do it
        /* ... */
    /* ... */
}

class X
{
    auto declaration debug() { /* ... */ }
    $f();
};
```

Resolution of $debug occurs within the invocation context and so the conditionalisation is resolved by the non-zero body of the meta-function X::debug.

Although this is a different problem to the lexical scoping problem, it is amenable to the same solution. Defining a meta-function declaration as establishing a nested lexical scope, causes the rewritten meta-function:

```
auto declaration f()
{
    if ($$debug)
        /* ... */
    /* ... */
}
```

to perform the resolution of debug in the surrounding lexical context ensuring the intended resolution to ::debug.

3.3.6 Multi-$-expression resolution

The generalisation of $-expressions is logical and predictable. The following example has nested meta-functions defined within nested classes. The outer meta-function `A::B::f()` is invoked within `D::E` to define the inner meta-function `D::E::A::g()`, which is invoked within `F::G`.

```
class A
{
    class B
    {
        auto declaration f()         // A::B::f
        {
            class A
            {
                auto declaration g()
                {
                    /*reference*/
                }
            };
        }
    };
};
class D
{
    class E
    {
        $A::B::f();
    };
};
class F
{
    class G
    {
        $D::E::A::g();
    };
};
```

Considering alternative invocations at `/*reference*/`, for which `Scope` is a convenient built-in meta-variable to demonstrate the context in which name resolution occurs:

$Scope resolves to `::F::G`, the invocation context of `::D::E::A::g()`

$$Scope resolves to `::D::E::A`, the surrounding context of `::D::E::A::g()`

$$$Scope resolves to `::D::E`, the invocation context of `::A::B::f()`

$$$$Scope resolves to `::A::B`, the surrounding context of `::A::B::f()`

$$$$$Scope resolves to `::A`, the surrounding context of `::A::B`

$$$$$$Scope resolves to `::`, the surrounding context of `::A`

$$$$$$$Scope is an error since `::` has no surrounding context

And for completeness:

Scope is an *identifier*.

@Scope resolves to `::F::G`, or a class derived from `::F::G`, if the reference occurs within a declaration that is regenerated by a derivation rule or meta-program.

This is clearly not a complete set of all contexts, since neither `D` nor `F` appear although they constitute lexically surrounding scopes of contexts that do appear. However the existence of `D` and `F` are implementation details that cannot be known to the author of the meta-functions. A more elaborate 2-dimensional scheme supporting access to the third surrounding lexical context of the second surrounding meta-function invocation would therefore be inappropriate.

The relevant lexical context is readily identified by counting back through braces surrounding the `/*reference*/` invocation expression. It is independent of the subsequent usage. Each additional `$` causes resolution in a less nested context: the invocation context for a meta-function body or the internal context of a class body.

Nested lexical contexts are established between:

- { and } of a class (or namespace) body
- { and } of a meta-function body
- :{ and } of an *object-statements-clause*

Nested lexical contexts are not established for:

- { and } or :{ and } within :{ and }
- structural nesting established by `::`
- compound statements
- compound declarations
- filespaces
- initializer clauses

3.3.7 Transferred lexical scope for *object-statements-clause*s

When meta-function or class bodies nest within an *object-statements-clause*, the *object-statements-clause* establishes a lexical context. The function bodies do not establish a further context.

```
auto statement f()
:{                    // Outer grouping :{} is a lexical scope
   derived(true)
   :{                 // Grouping :{} is not a lexical scope
      entry
      {               // Grouped {} is not a lexical scope
         /* ... */
      };
   };
};
```

3.3.8 No lexical scope for `::` nesting

It might seem that `::` should establish a nested lexical scope so that there is no difference between the following pair of meta-functions:

```
auto identifier A::B::f() { $$Scope; }
class A
{
   class B
   {
      auto identifier g() { $$Scope; }
   };
};
```

However this would introduce an inconsistency for

```
auto $X A::B::h() /* ... */
```

for which C++ would suggest that `X` be resolved in its surrounding (global) context rather than the nested `A::B` context. There is no strong reason why `f()` and `g()` should behave in the same way, and good reasons why they should not. Therefore `::` does not establish a lexical scope.

3.3.9 No Lexical scope for initializers and arguments

Not establishing lexical contexts for initializer clauses seems obvious since in

```
int scalingMatrix[2][2] = { { $Scale, 0 }, { 0, $Scale } };
```

the braces are a grouping operator, they do not create a lexical context in which declarations could be differently resolved.

It is less clear whether the = and ; of a variable initializer or the (or , and , or) surrounding a meta-function argument should establish a nested lexical scope.

At first sight it is clear that in

```
class X
{
    auto identifier p = $Scope;
    $f($Scope);
};
```

$Scope should be resolved as X using the prevailing context. However, the semantics of meta-function execution return the body of a meta-function for interpretation in the invoking context. Perhaps the initializers and arguments should also be interpreted within their usage context. It is then necessary to use $$Scope to ensure that the above example has the obvious behaviour.

Considering a more complicated meta-function call in which a *class-specifier* is passed as a meta-function parameter to create the class To::X.

```
auto declaration f(class_specifier c)
{
    class ::To
    {
        $c;
    };
}
class Call::From
{
    $f(class X
        {
            ... $Scope ...
            ... $$Scope ....
        });
};
```

which expands to

```
class Call::From
{
    class ::To
    {
        class X
        {
            ... $Scope ...
            ... $$Scope ....
        };
    };
};
```

$Scope should clearly resolve to ::To::X, rather than Call::From (which is not visible from ::To::X), or Call::From::X (which may not ever exist).

$$Scope resolves to a surrounding context, but is it Call::From (that surrounding the definition of the argument) or ::To (that surrounding the instantiation as ::To::X)?

Either resolution is tenable, however resolution as To in the invoking context involves the use of a surrounding context that is not lexically identified where a meta-variable is initialized or where a meta-function argument specified. It does not seem necessary to make this scope easily accessible. This invisibility of remote contexts mirrors the invisibility of the remote lexical scopes for D and F in the nested class and meta-function example of Section 3.3.6. Therefore initializers and meta-function arguments do not establish a nested lexical scope: resolution of $-expressions has the obvious behaviour.

Although the remote surrounding lexical context is not directly accessible, it can be accessed indirectly. When X is structurally as well as lexically nested, Scope can be resolved in the surrounding structural scope by $OuterScope. Even when X is not structurally nested, access is possible by creating a dummy lexical and structural nesting:

```
class From
{
    $f(auto class ExtraNesting
        {
            class ::X
            {
            ... $Scope ...
            ... $$OuterScope ....
            };
        });
};
```

The structurally and lexically nested ExtraNesting meta-class establishes an additional lexical context surrounding X, which can be reached using $$, and from which OuterScope can traverse to the surrounding structural (and lexical) context.

3.3.10 Formal parameters

Formal parameters are visible throughout a meta-function-body, including any nested lexical contexts, so it is unnecessary to use additional $'s to access formal parameters from within nested lexical contexts.

```
auto declaration X::f(identifier outerParameter)
{
    $resolved_in_invocation_context;
    $$reaches_out_to_defining_context;
    auto declaration h(identifier innerParameter)
    {
        class $outerParameter
        {
            $innerParameter;          // one $ is enough
            $outerParameter;          // one $ is enough
        };
    }
}
```

The semantics of meta-function execution, that resolves tree-literals primarily in the invocation context, makes the definition context less accessible and requires the use of a multi-$-expression. This is not adequate for all forms of access. For instance in the above example, definition as X::f rather than class X { ... f }; prevents access to X using a built-in meta-variable. If the meta-function is invoked in a derived class, neither form of definition supports access to the derived definition class name. Two built-in formal parameters are therefore supplied to circumvent the problems.

The Static built-in formal parameter resolves to the static definition scope of the meta-function (X in the example).

The Dynamic built-in formal parameter resolves to the dynamic definition scope of the meta-function (which could be a class derived from X in the example).

Formal template parameters are treated as formal parameters supporting use of the definition parameterisation in the invocation context.

These additional built-in parameters may hide names in the invocation context, necessitating the use of `$This.Static` to resolve `Static` in the invocation context rather than as a built-in formal.

3.3.11 Meta-function and substitution semantics

Declarations within meta-function bodies are analysed in three contexts

- when the meta-function is defined
  - to resolve (multi-)$-expressions that use formal parameters
  - to resolve multi-$-expressions in the definition context
- each time the meta-function is invoked
  - to resolve residual $-expressions in the invocation context
- when any generated declarations are compiled
  - to resolve @-expressions in the actual context

Formal parameters may hide declarations in nested classes, and may be hidden by formal parameters of nested meta-functions. A formal parameter occluded in this way may be accessed by using sufficient $’s to reach out to a lexical context in which the occluding formal is not visible.

```
auto declaration g(identifier id)
{
    auto declaration h(identifier id)
    {
        $j($$id, $id);        // j(outer-id, inner-id)
    }
}
```

It is an error if a $-expression reaches out to an external context where it cannot be resolved.

The usage of the formal parameter names is identified during analysis in the defining context. The formal parameter names are therefore not visible while resolving $-expressions in the invocation context.

```
auto declaration k(identifier aParameter)
{
    ${$nestedInvocation};
    ${$$nestedDefinition};
}
```

Therefore, if in the above example `$nestedInvocation` has the value `aParameter`, the subsequent access to `aParameter` is resolved in the invocation context, ignoring the formal parameter. Furthermore, since `$$nestedDefinition` reaches out to the external context, it is resolved in the defining context and so if `nestedDefinition` has the value `aParameter`, the $-expression is resolved to the formal parameter.

Upon invocation of a meta-function, all accesses to formal parameters within the meta-function body are replaced by their corresponding actual arguments before the entire body and residual $-expressions are interpreted as part of the invocation context.

3.4 Composition

The C++ One Definition Rule (§3.2) permits only one appearance of each declaration, and requires the declaration to have the same meaning in all compilation sessions in which it is used. A few exceptions to the rule permit repeated forward references and typedefs.

Violations of the rule within a single compilation should be trapped by the compiler. Violations between compilation sessions may go undetected and lead to unpredictable program behaviour. Some of these inter-session violations are diagnosed by the practice of using name mangling for function declarations. Most opportunities for inter-session violations are eliminated by the practice of placing interfaces within include files that are shared between compilation sessions. As a result a program malfunction due to violation of the ODR arises mainly through the undue enthusiasm of some incremental compilers.

The ODR in combination with the hierarchical nature of C++ class declarations prevents declarations from more than one class being interleaved. This is a severe impediment to the implementation of patterns or Aspect Oriented Programming and so FOG relaxes the ODR with respect to FOG source code, requiring only that it be possible to satisfy the ODR after translation has been completed.

Multiple declarations are permitted and the contributions from each are combined hierarchically to form composite declarations. If the contributions are incompatible, the inconsistency is diagnosed and the resulting behaviour is unpredictable.

Composition is performed for the actual declarations of classes (including namespaces, structs and unions), enums, arrays, variables, functions, meta-variables and meta-functions.

When any potential declaration is associated with a determined context, an existing actual declaration is first located. If one is found, the new potential declaration is composed with the existing actual declaration. When searching for such a declaration, the search for the declared name, but not its scopes, is restricted to the specified scope.

Thus a declaration for `A::B::f` creates `f` within the scope visible as `B` within the scope visible as `A`. It may compose within an existing `f`. It hides, rather than composes with, an `f` that is visible in (by inheritance) but not part of `A::B`. This implies that `A::B::A::f` is legal and probably refers to `A::f` since the more nested `A` is visible as the less nested `A`. The unlikely alternative with an occluding nested class is demonstrated by:

```
class A;
public class A::B;
private class A::B::A;
```

in which case `A::B::A::f` and `A::f` would be distinct entities.

Names must be made visible before they are used. In particular scopes must be forward declared as in the cascade of declarations leading to `A::B::A`. Direct declaration of a nested scope without its less nested scopes could be interpreted as an implied forward declaration, but it leaves `class`/`struct`/`union` distinctions and nested access constraints unclear. It also requires assumptions to be made about the nature of the intervening names. These assumptions may be invalidated by typedefs or additional base classes.

3.4.1 Class composition

Most class composition occurs hierarchically: the class grows to accommodate distinct member declarations, or to compose repeated member declarations.

3.4.1.1 Nested contexts

Nested *class-specifier*s, *namespace-definition*s and *linkage-specification*s compose hierarchically.

Nested filespaces are not nested contexts. The filespace just associates file-placement with its enclosed declarations. Declarations within a filespace are therefore composed without reference to the filespace beyond retention of their required placement. It is an error for composed declarations to have conflicting placement requirements.

Compound declarations are not nested contexts. They are just a syntactical grouping of multiple declarations as a single declaration. Declarations within a compound-declaration are therefore composed individually ignoring any compound structure.

3.4.1.2 Base classes

A *base-specifier-list* comprises an ordered list of *base-specifier*s. Composition occurs as one or more lists of potential *base-specifier* declarations (each being a `base_specifier` meta-object) are transformed into the single composed list of actual *base-specifier* declarations (each being a base meta-object). Additional potential *base-specifier*s whose class-name is already on the list compose with the existing actual. Additional potential *base-specifier*s for new class-names are converted to actuals and appended to the list.

Composition of a *base-specifier* involves merging the *access-specifier* and the `virtual` keywords. An error arises if a conflict arises such as `private` with `protected`, or `virtual` with `!virtual`. A conflict does not arise when merging `private` with unspecified access or `virtual` with unspecified virtual.

```
    class A : B, C, B, !virtual D {};
    class A : E, protected B, F, private D, public E {};
```

composes to:

```
    class A : protected B, C, private !virtual D, public E, F {};
```

3.4.1.3 Miscellaneous declarations

accessibility-specifier

An *accessibility-specifier* changes defaults for subsequent declarations, within the prevailing context. An *accessibility-specifier* does not affect and is not composed with other contributions to the same context.

***meta-expression*s and *meta-control-declaration*s**

Meta-programs are interpreted directly and so there is nothing to compose.

include-declaration

An *include-declaration* is interpreted directly and so there is nothing to compose.

syntax-macro-definition

It is not possible to compose *syntax-macro-definition*s because the respecified trigger word triggers the existing definition:

```
    explicit auto declaration macro() { /* ... */ };
    explicit auto declaration macro() { /* ... */ };
```

There is no suppression of syntax-macro recognition within syntax-macro definitions. Perhaps there should be.

using-directive
namespace-alias-definition
file-dependency-declaration
explicit-instantiation
explicit-specialization

Multiple declarations are gathered together. Duplicates are eliminated.

asm-definition

Multiple declarations are gathered together. Duplicates are preserved.

file-placement-declaration

Multiple declarations are gathered together. Only one distinct location may be specified for each interface and implementation file.

3.4.2 Object statement composition

Object-statement-scopes provided a limited form of scope at meta-compile time for functions and variables. Composition of declarations proceeds in the same way as composition for equivalent concepts in classes.

The presence of derivation rules supports multiple bodies in a potential declaration. Composition of these bodies is deferred until the compile compilation stage at which point only those bodies that are enabled by derivation predicates are retained.

3.4.3 Enum composition

In C++, an *enum-specifier* comprises a list of *enumerator-definition*s each of which is a name-value pair. The value part may be omitted, in which case it assumes the value zero for the first enumerator, or the preceding value plus 1 for subsequent values.

```
enum Enums
{
    ZERO,                           // Implicitly 0
    TWO = 2,                        // Explicitly 2
    THREE                           // Implicitly TWO + 1
};
```

In FOG, composition occurs as one or more potential declarations comprising meta-objects of `enum_specifier` meta-type and lists of meta-objects of `enumerator_definition` meta-type are converted to an actual declaration of `enum` meta-type and an actual list of `enumerator` meta-type.

As each additional potential *enum-specifier* is composed, each additional potential *enumerator-definition* is appended to the list of actual enumerators. The missing enumerator value is resolved as zero for the first enumerator or one plus the value of the most recent addition to the list for all subsequent values.

```
enum Enums                          // Composing with above
{
    FOUR,                           // Implicitly THREE + 1
    FIVE,                           // Implicitly FOUR + 1
    TWO = 2,                        // Explicitly 2
    THREE,                          // Implicitly TWO + 1
    ONE = TWO - 1                   // Explicitly TWO - 1
};
```

Enumerator names may be repeated provided the value associated with the enumerator is the same for each repetition. Repeated names are discarded so that a meta-program traversal of the enumerators and the final code emission iterates over the domain

```
{ ZERO = 0, TWO = 2, THREE = 3, FOUR = 4, FIVE = 5, ONE = 1 }
```

The “same value” involves a direct comparison of numeric values. There is no support for deferred evaluation and comparison of Abstract Syntax Trees.

3.4.4 Object composition

Composition of objects first identifies the object (function, typedef or variable) to be composed using its unique signature, and then composes the remaining parts of the declaration.

The unique signature comprises template arguments, scope, name, type and *cv-qualifier*s of the object. The type includes the recursive types and *cv-qualifier*s of any function parameters. The unique signature does not include other *decl-specifier*s such as `static`, `virtual` or `inline`, or function parameter names or default function arguments[4]. Type names using typedefs are resolved. Redundant *cv-qualifier*s for by-value arguments are ignored (§13.1-3).

The `friend`, `typedef` and `using` *decl-specifier*s are not composed. `friend` and `typedef` distinguish between different categories of object. They must be present for each declaration of a name. `using` indicates that only part of the declaration is provided, the remainder is obtained from the re-used declarations (see Section 2.1.4.4).

The `private`, `protected` and `public` *access-specifier*s are composed using a four-valued algorithm. Matching values compose to preserve the value. An omitted value composes with an explicit value to preserve the explicit value. Conflicting explicit values are an error. A missing value for a new declaration specified within class braces automatically acquires the prevailing access for declarations in its scope. Repeated declarations within class braces, or any declaration outside class braces retain unspecified access until composed with a declaration that has a defined access. It is an error for a declaration forming part of a class to have no *access-specifier* from any source. *access-specifier*s for declarations forming part of a *namespace-body* or *linkage-specification* are discarded.

The `static` and `!static` *decl-specifier*s are composed using a three-valued algorithm. Matching values compose to preserve the value. An omitted value composes with an explicit value to preserve the explicit value. Conflicting explicit values are an error.

The `explicit`, `export`, `extern`, `mutable` and `register` *decl-specifier*s are composed using a two-valued algorithm. Matching values compose to preserve the value. An omitted value composes with an explicit value to preserve the explicit value. There are no conflicting values to cause an error.

3.4.5 Value composition

A pair of values may need to be composed as initializers for variables, dimensions of arrays, or default initializers for function arguments. Successful composition selects a value from the available sources and verifies that alternative values do not conflict. Conflict is determined by a syntax tree comparison after any tree-literals and constant values have been resolved. No symbolic interpretation of the syntax tree is performed, and so a redundant pair of parentheses or the interchange of binary operands is sufficient to cause a composition failure.

3.4.6 Variable composition

Declarations of (non-array) variables are composed by composing the residual *decl-specifier*s as outlined above in Section 3.4.4, and initializers as described in Section 3.4.5.

4. Exclusion of `static` from the signature is possible since the standard specifically excludes overloading static and non-static member functions (§13.1-2).

3.4.7 Array composition

Declarations of array variables are composed by composing the residual *decl-specifier*s as outlined above in Section 3.4.4, and dimensions as described in Section 3.4.5. Composition of array initializers is performed by extending the enumerator composition policy to multiple dimensions and introducing the gcc [Stallman98] indexed initializer syntax (Section 2.1.4.2) to define a specific placement of an initialization value. Explicitly positioned initializers take their assigned locations. Other initializers take the location adjacent to the previous initializer, which is `[0][0]...` for the first element of the first array declaration, and a data dependent value for a value following a specifically positioned initializer, or a subsequent array declaration.

Multiple initializers for the same array index are composed provided the values are consistent as described in Section 3.4.5. Missing initializers are given a zero value when the equivalent C++ declaration is emitted for the composite initializer.

The default incremental composition is most useful within the idiomatic loop

```
class X
{
    public static const char *variableNames[];
    for (iterator p = $variables(); p; ++p)
        using variableNames = { ""$p->name() };
    using variableNames = { 0 };
};
```

The first line fully declares an array variable but without a dimension or any initializers. The loop iterates over the member variables of `class X`. The body of the loop contains a *re-using-declaration* of the array name and provides a single initializer comprising the concatenation of the empty string and the member variable name. Each iteration composes the additional initializer with the existing declaration, so that the array gradually builds up a list of strings. Finally the last line adds a null terminator to the list.

Whether to avoid a potentially conflicting redeclaration is a matter of style. The loop could be specified more explicitly, saving a line:

```
auto for (iterator p = $variables(); p; ++p)
    public static const char *variableNames[] = { ""$p->name() };
public static const char *variableNames[] = { 0 };
```

Indexed initializers are useful for applications such as automatically creating an array of debug text strings from an enumerator.

```
auto declaration EnumTextArray(name textsName, enum enumDecl)
{
    for (iterator p = $enumDecl.enumerators(); p; ++p)
        using $textsName = { [$p->value()] ""$p->name() };
}
class EnumClass
{
    public enum Enums { A, E = 4, F, B = 1 };
};
static const char *enum_names[];
$EnumTextArray(enum_names, EnumClass::Enums);
```

is equivalent to and eventually emitted as

```
class EnumClass
{
public:
    enum Enums { A, E = 4, F, B = 1 };
};
static const char *enum_names[] = { "A", "B", 0, 0, "E", "F" };
```

3.4.8 Function composition

Function composition involves composition of residual *decl-specifier*s, parameter names, default arguments and function bodies. Composition of non-*function-specifier decl-specifier*s proceeds as above. The composition policies specific to functions are:

The `inline`, `!inline`, `inline/interface` or `inline/implementation` *decl-specifier*s are composed using a five-valued algorithm. Matching values compose to preserve the value. An omitted value composes with an explicit value to preserve the explicit value. Conflicting explicit values are an error, except that composition of plain `inline` with the more specific `inline/interface` or `inline/implementation` composes to preserve the more specific value.

The `virtual`, `!virtual` and `virtual/pure` *decl-specifier*s are composed using a four-valued algorithm. Matching values compose to preserve the value. An omitted value composes with an explicit value to preserve the explicit value. Conflicting explicit values are an error. For the above algorithm `virtual/pure` may be explicitly specified as `virtual/pure` or by use of `virtual` and a subsequent `= 0` *pure-specifier*. The existence of a corresponding virtual function in a base class is ignored during composition, however during the compilation phase the virtual attribute is propagated to derived functions, resulting in an error message for a conflicting requirement for a `!virtual` function derived from a `virtual` function, and a warning message for a `virtual` function derived from a non-`virtual` function.

Function parameter types do not compose, since distinct types represent distinct overloaded functions.

Function parameter names compose using a two-valued algorithm. Matching names compose to preserve the name. An omitted name composes with an explicit name to preserve the explicit name. Conflicting names are currently an error. This is a necessary constraint imposed by the potential for re-use of parameter names by multiple function bodies or derived functions. A less restrictive implementation should tolerate a local respecification of parameter names in function bodies and automatic renaming to those in the first declaration.

Default function arguments compose as described in Section 3.4.5.

Constructor initializers compose by gathering all initializers together. Multiple initializations for the same member variable compose as described in Section 3.4.5, however since the initializers form part of the function body, they are guarded by derivation predicates. Disabled initializations are ignored. When the constructor is emitted, the constructor initializers are emitted in constructor initialization order using explicit values, where available, with implicit values from member variable initializers as defaults.

Composition of *exception-specification*s and *function-try-block*s goes well beyond the considered policies for FOG. It would appear that *exception-specification*s for a particular function should just be gathered together with duplicates discarded. It seems that *exception-specification*s should propagate up the inheritance hierarchy to extend non-empty *exception-specification*s, so that the *exception-specification* of a derived virtual function should never be wider than its inherited *exception-specification*. This may lead to errors when inheriting from library classes, whose specifications cannot be changed, but these errors diagnose a problem rather than imposing a restriction. *function-try-block*s should probably just be concatenated on a per-handled type basis, with handlers organised to ensure handlers for more derived types precede those for less derived types. It may be necessary to extend the syntax for a default member-variable initializer to wrap a *try-block* around it.

The overall function body is formed from the concatenation of contributions to five named segments within which contributions are independently composed.

```
return-type function-name(function-arguments) cv-qualifiers
                                     exception-specification
{
    entry-segment-contribution
    pre-segment-contribution
    body-segment-contribution
    post-segment-contribution
    exit-segment-contribution
}
```

Contributions to each segment are concatenated and by default, contributions are made to the body segment, so that

```
void f() { i++; }
void f() { i--; }
```

composes to

```
void f()
{
    i++;
    i--;
}
```

More explicit control of contributions requires the use of an *object-statements-clause* in which the extra annotation syntax does not cause conflicts. The `entry` and `exit` segments are useful for establishing a function framework:

```
class Manager
{
    public Manager() {}
    public bool do_it()
    :{
        entry { bool exitStatus = true; };
        exit { return exitStatus; };
    };
};
```

Other sources modules may contribute code. An independent module could specify:

```
class Manager
{
    private MyContext& _context = MyContext::make(*this);
    using do_it { if (!_context.do_it()) exitStatus = false; }
};
```

Composition of the framework established by `entry` and `exit` segments together with the extra body results in the composed function:

```
bool Manager::do_it()
{
    bool exitStatus = true;
    if (!_context.do_it())
        exitStatus = false;
    return exitStatus;
}
```

The framework can be safely extended by any number of independent contributions. As well as extending the do_it function, an additional member variable was specified with a default initializer. The composed class therefore looks like:

```
class Manager
{
private:
    MyContext& _context;
public:
    Manager();
    bool do_it();
};
```

The default initializer is automatically supplied for all non-copy constructors:

```
Manager::Manager()
:
    _context(MyContext::make(*this))
{}
```

The five segments provide sufficient flexibility for many applications, but are not adequate for all. For instance, wrapping an `if` around a body:

```
pre { if(...) { }          // Not valid in FOG
post { } }                 // Not valid in FOG
```

requires partial syntax that violates the syntactical requirement for *statement*s. There is also a parsing ambiguity between { and } as partial syntax and as syntax structuring. (An earlier version of FOG used \{ and \} for partial syntax and allowed any list of tokens as a composed contribution.)

The structure of a function body is defined with two levels of indirection through meta-variables:

```
auto statement function::value[] = { @function_structure };
auto static statement token::function_structure[] =
    { @entry; @pre; @body; @post; @exit; };
auto statement function::entry[] = {};
auto statement function::pre[] = {};
auto statement function::body[] = {};
auto statement function::post[] = {};
auto statement function::exit[] = {};
```

The segment-name/body syntax

```
derived(is_root()) entry { bool exitStatus = false; }
```

is therefore syntactic sugar for:

```
derived(is_root()) auto entry += { bool exitStatus = false; };
```

An `if` may therefore be wrapped around the existing body without violating syntax constraints by:

```
auto body = { if (...) @body; };
```

Braces are optional around @body since it is treated as a single statement. The use of @ rather than $ is very important to defer evaluation until the compilation phase. A $ would be evaluated when the potential declaration is associated with an actual function, allowing further potential declarations or meta-programs to extend the body following rather than within the `if`.

The structure of all functions can be changed by redefining `token::function_structure`: a practice to be avoided like redefining `::operator new`.

The structure of an individual function can be changed by defining `structure` within an *object-statements-clause*. Code segments may be added and removed, but the syntactic sugar for the five built-in segments is unaffected; user-defined segments can only be extended using an expression operator such as +=.

Statements can be appended to a segment using +=. The contents of a segment can be replaced by just using =. Statements can be prefixed by

```
auto segment = { /*prefix*/ @segment; };
```

Declaration of a meta-variable as const precludes multiple assignments, and so may be used to provide some protection against unforeseen activities by interfering meta-programs. Thus

```
auto const statement body[] = {};
```

(re)defines the body segment to be empty, detects an error if the existing content is anything other than the built-in empty default, and detects an error when any subsequent attempt is made to change the content.

As semantic analysis and meta-programming proceeds, the meta-objects for each function build a list of object-statements with their associated derivation predicates. @-expressions in these object-statements remain unevaluated.

During the compilation stage, classes are compiled in least derived first order, and individual members of each class are compiled in an unspecified order. The list of object-statements is scanned in the order in which potential declarations contributed to actual declarations, with the further inherited lists of object-statements applied in destruction order: a base-class can therefore wrap code around its derived implementations predictably.

Historical Note

The multi-pass implementation of FOG was able to, but did not, parse function bodies which were therefore treated as an arbitrary list of brace-delimited tokens. Code composition was supported by arbitrary concatenation of such lists imposing total flexibility and anarchy since there was no syntactical constraint upon the composition, and no discovery of error until a subsequent compilation failed.

The superset implementation parses function bodies and imposes syntactical consistency for each composition. These constraints provide much needed integrity and respond to some of the hazards outlined by [Ossher98] and impose some of the discipline discussed by [Mulet95].

3.4.9 Meta-variable composition

Meta-variables are working variables for use at meta-compile-time. A redeclaration of a meta-variable is therefore very similar to assignment to the same meta-variable, differing only in the requirement for the redeclaration to have the same meta-type and const-ness as the preceding declaration. The redeclaration changes the value of the meta-variable.

Redeclaration of a const meta-variable with the same value is permitted, although reassignment of a const meta-variable with the same value is an error.

3.4.10 Meta-function composition

Meta-functions compose in the same way as functions. By default, contributions are gathered into the body list of declarations. When a meta-function is invoked, the body returned for incorporation in the invoking context comprises the hierarchical composition of segment contributions with the formal parameters replaced by actual arguments.

Declaration of a meta-function with a const declaration list can be used to inhibit a redefinition:

```
auto declaration f()
:{
    auto const statement body[] = { /* ... */ };
};
```

3.5 Meta-classes

Conventional programming involves programs that operate on application entities. Meta-programming involves programs that operate on program entities: the class, function and variable declarations that define a program.

In the same way that newcomers to Object-Orientation are easily confused by loose usage of the terms class, instance and object, newcomers to meta-programming are easily confused by loose usage of the term meta-class. One confusion arises because, in Object-Oriented Programming, the phrase *is-a* denotes an inheritance relationship, but in meta-programming *is-a* can alternatively be used to denote an instantiation relationship. Further confusion arises because the object models available to the programmer do not correspond to the underlying abstraction.

The exact one to one relationship between classes and meta-classes in FOG is easy for the programmer to appreciate and natural to use, but apparently in conflict with some of the more traditional perspectives of meta-classes and meta-meta-classes. We will therefore describe the very pure Object Model exemplified by ObjVlisp [Cointe87], before describing the C++ Model and the enhancements provided by FOG in order to justify the FOG model.

A very simple three-class hierarchy is shown in the central column of Figure 3.7 in which class X inherits from class Y, and class Y inherits from class Z. A single instance of each class is shown to its right and named respectively anX, aY and aZ.

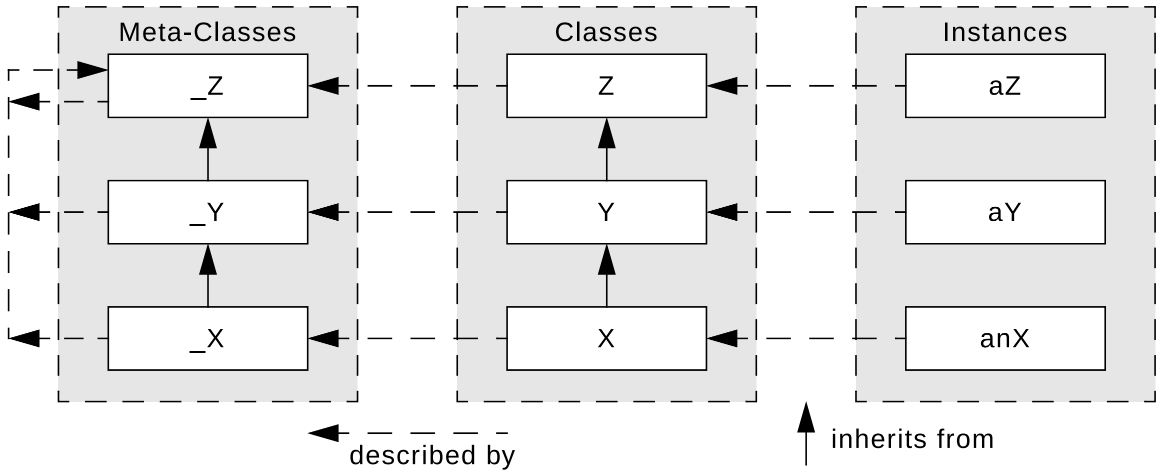

Figure 3.7 Pure OO Object Model

An Object Oriented program performs computation as a result of the interaction of its object instances at run-time, and in the simplest Object Model, instances have meaning only as object instances at run-time; classes exist solely as an abstraction at compile time.

A more sophisticated Object Model enables the run-time objects to make use of class information, and in a pure Object Model, this information is provided by a run-time object for each class. Each such class object provides a description of the instances of its class.

Every object must be an instance of some class, so it is necessary to define meta-classes that are instantiated as the class objects. The meta-classes are labelled _X, _Y and _Z in the figure.

The pure Object Model requires a run-time object to describe each of these meta-classes, and so there are corresponding meta-class objects to describe each class. A potentially infinite recursion is avoided by ensuring that the instance of _Z is also a valid description of _Z. Every box in Figure 3.7 corresponds to a run-time object; vertical arrows denote an inheritance relationship; horizontal arrows denote an *instance-of* or *described-by* relationship from right to left or a *describes* relationship from left to right.

The pure Object Model just described is that of ObjVlisp [Cointe87]. Few other languages comply to it. Smalltalk provides a similar but more restrictive model; the meta-classes of meta-classes are not visible [Briot89]. CLOS and SOM provide considerably more generality. The programmer is free to choose a different inheritance relationships for meta-classes and classes. This leads to a number of significant compatibility issues [Graube89], whose resolution seems to create further problems [Danforth94], [Forman94].

Meta-classes were originally introduced for languages such as Smalltalk and CLOS to assist in the construction of instances whose layout was entirely defined at run-time. More efficient languages such as C++ or Eiffel define object layouts at compile-time and compiler writers have no need to provide meta-classes. The available facilities in C++ are limited. The C++ components corresponding to Figure 3.7 are shown in Figure 3.8.

The instances in the right hand column comprise

- a contiguous piece of memory for the member variables
- a hidden pointer to the instance description often called `vptr`, which has been arbitrarily labelled `-rtti-`
- discontiguous memory areas for the member function code.

Each class (and meta-class) ‘object’ comprises a potentially contiguous area of memory containing the compiler generated run-time type information as an instance of `std::type_info`, and generally discontiguous areas of memory for the static member variables and functions. Most of the behaviour of `std::type_info` is implementation defined, but it will typically comprise

- a pointer to the class description (`-rtti-`)
- a list base-class instance descriptions (`-bases-`)
- a class name
- a dispatch table for virtual functions (`-vtable-`).

Very limited functionality is exposed for the `std::type_info` class, whose sole instance both describes and is an instance of the `std::type_info` class. This is the only meta-class in C++.

The class object of any suitable type T is returned by the `typeid(T)` operator. The solitary meta-class object is returned by `typeid(typeid(T))` or just `typeid(std::type_info)`.

Class variables

The symmetry of Figure 3.7 and the presentation in Figure 3.8 implies that the class variables (static member variables) form a logical part of their corresponding class object. This follows postulate 6 of [Cointe87] that the class variables of an instance are the instance variables of its meta-class. However, in common with other programming languages, C++ does not require the programmer to distinguish between use of class variables and instance variables. This programming convenience leads to some confusion between class and instance meanings, and undermines the pure perspective of the meta-class structure. The class variables are logically shared parts of each instance. The alternative detailed presentation of Figure 3.9 and its simplified presentation in Figure 3.10 is more appropriate. All visible names now appear in one rather than two columns.

At the source level, a member of a class or instance is accessed by name. The access is resolved by consulting the instance description5 to convert the name into the address of a variable or function. In C++, this conversion is almost entirely

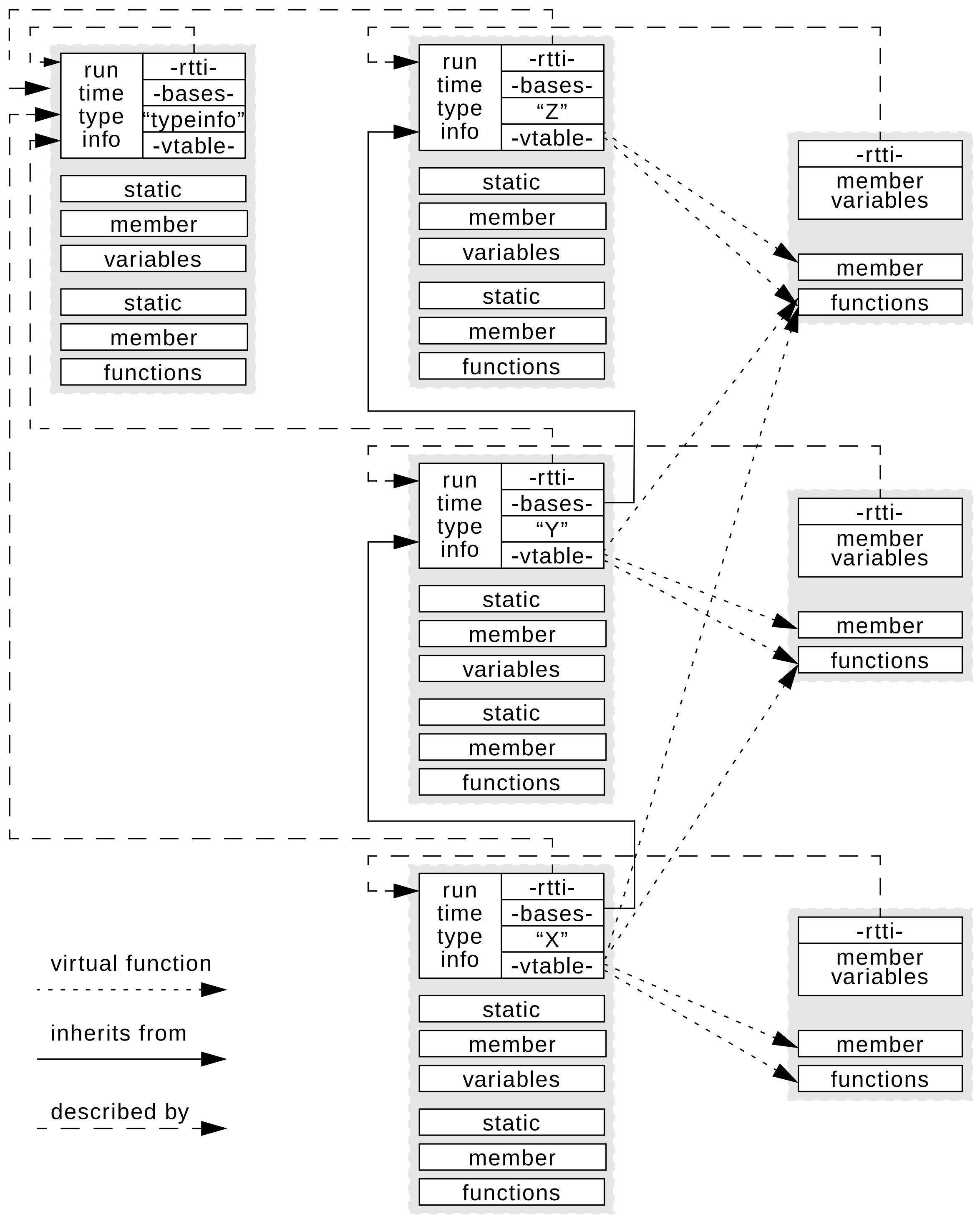

Figure 3.8 C++ Object Model (Memory Perspective)

performed at compile-time; those few conversions that cannot be completely resolved at compile-time are partially resolved as fixed indexes that index the `-vtable-` at run-time. The `-vtable-` is all that remains of the more general name to address conversion table required by less compiled languages. Seen from this perspective, class variables and instance variables differ in their access policy, but share the same name-space. Each class variable exists with a 1:1 relationship to a class, and is accessible by name from that class, and derived classes. However,

5. "class description" misleadingly refers to the wrong meta-level.

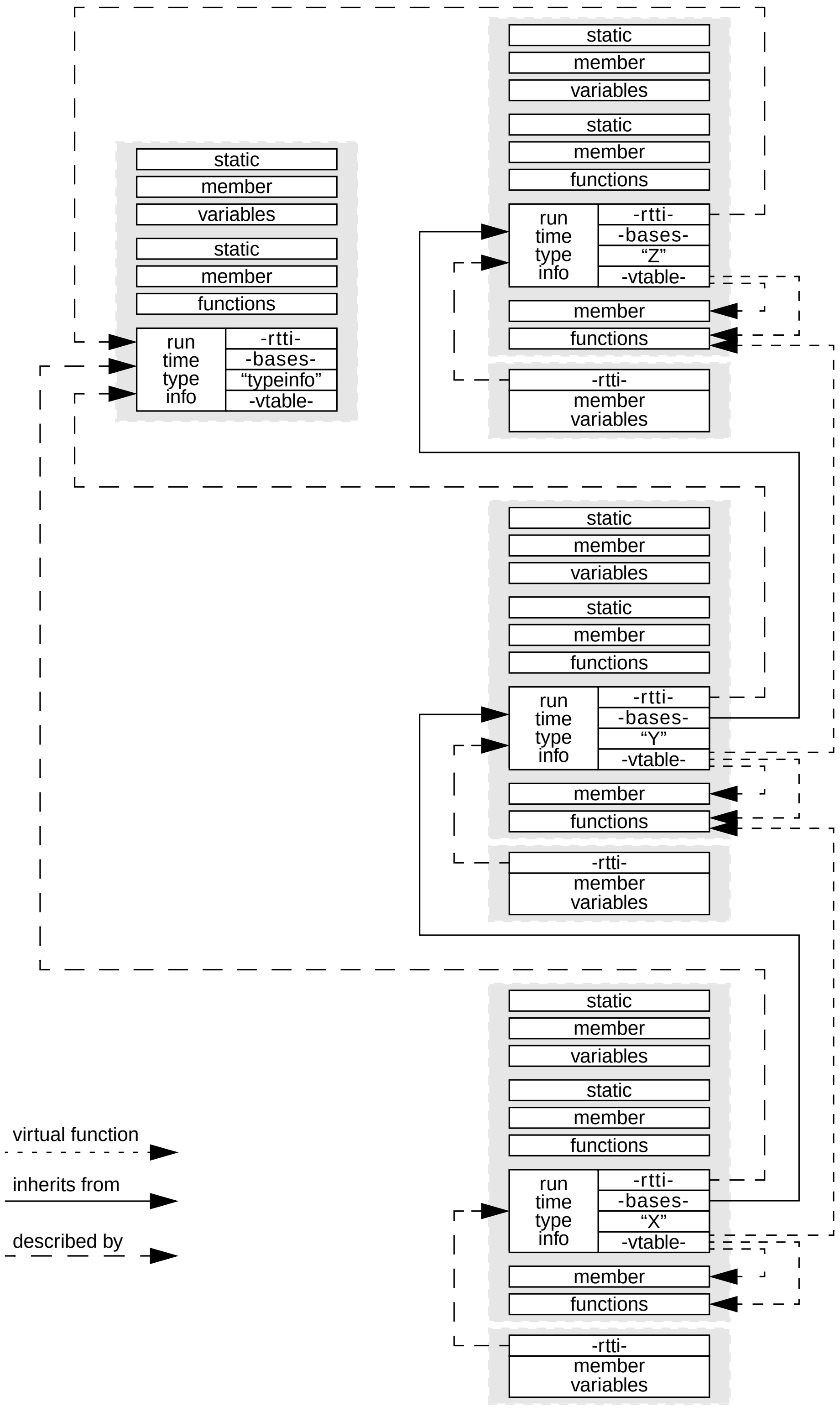

Figure 3.9 C++ Object Model (Naming Perspective)

there is no reason for class variables to be grouped as an object, and in C++ they are not. It is in fact impossible to group class variables as a contiguous object, since shared class variables cannot be adjacent for more than two distinct derived classes. When class variables are accessible with respect to instances, there is no reason for them to be accessible with respect to their classes as well, and again in C++ they are not. However, the symmetrical interpretation, in which class variables are the instance variables of their meta-classes, requires that they are accessible with respect to instances and classes and so blurs the distinction between meta-levels [Maes87].

Therefore names of class variables (and functions) are visible in the instance description (the conventional name-space for access with respect to instances), but are omitted from the class description (the meta-namespace for access relative to instance meta-objects (classes)).

Compile-time Object Model

The run-time Object Model of Figure 3.9 is redrawn without the clutter and for a single level of inheritance in Figure 3.10.

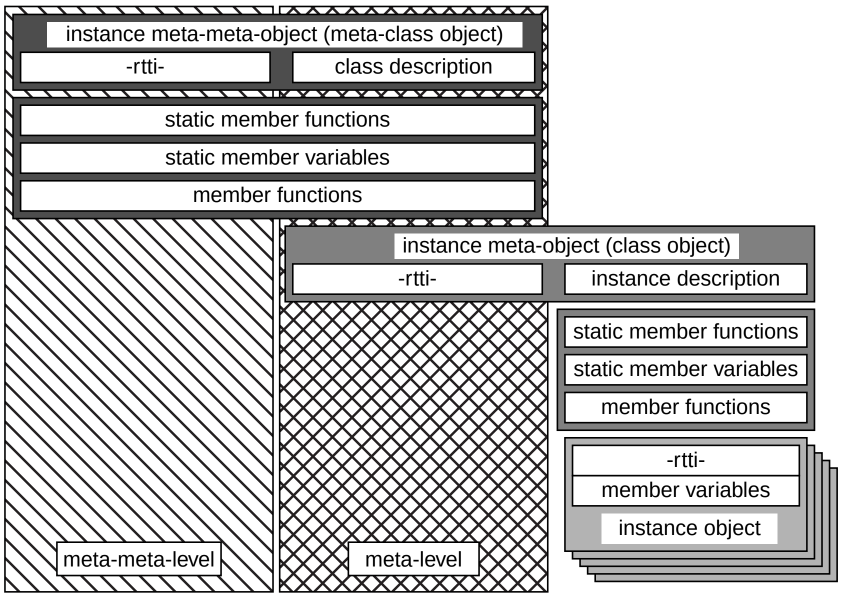

Figure 3.10 C++ Run-time Object Model

Programming normally operates using the multiple instance objects of each class shown in the right-hand column. Static and non-static member functions and variables share the same name-space. The hidden `-rtti-` pointer identifies the run-time type information description of the instance. Invocation of `typeid()` shifts the programming perspective to the meta-level, where only the members of the `typeinfo` instance describing the class are visible. A further invocation of `typeid()` shifts to the meta-meta-level and since `typeinfo` is its own meta-class, only its members remain visible.

The corresponding compile-time programming model for FOG is shown in Figure 3.11.

A class meta-object has instance variables and functions, as well as class variables and functions in just the same way as a run-time instance meta-object (class) has instance (member) variables and functions and class (static member) variables and functions. The class meta-object members are referred to as (non-static) meta-members and static meta-members by direct analogy with (non-static) members and static members.

Each class declaration forms part of a class meta-object that also comprises meta-variables and a description of the run-time instance objects. The very right-hand ‘level’ corresponds to the normal run-time perspective, in which a small amount of the instance description maintained by the class meta-object is provided as the run-time-type-information of the class object. This level does not exist at compile-time.

Meta-programming occurs at the meta-level where the description of objects (member names and types) is available as well as the meta-members of the meta-class. The `-ctti-` counterpart of `-rtti-` identifies the describing meta-object.

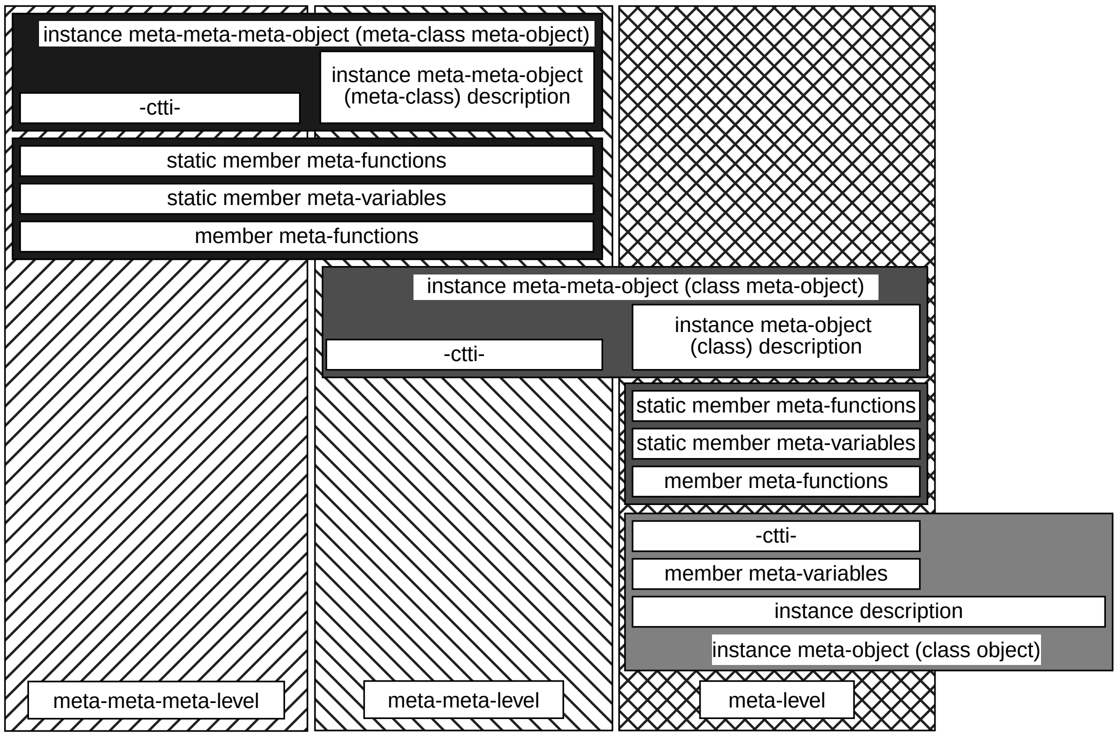
instance meta-meta-meta-object (meta-class meta-object)
instance meta-meta-object (meta-class) description
-ctti-
static member meta-functions
static member meta-variables
member meta-functions
instance meta-meta-object (class meta-object)
instance meta-object (class) description
-ctti-
static member meta-functions
static member meta-variables
member meta-functions
-ctti-
member meta-variables
instance description
instance meta-object (class object)
meta-meta-meta-level
meta-meta-level
meta-level

Figure 3.11 FOG Compile-time Object Model

The class description describing an instance meta-object contains the full mapping from names to built-in and user-defined meta-functions and meta-variables, rather than the highly optimised `-vtable-` in the run-time type information.

Instance meta-objects are instances of `class` or `class_specifier` meta-types. Class and meta-class meta-objects are instances of the `meta_type` meta-type.

In the same way that `typeid()` shifts the perspective to the left in the run-time diagram, the `meta_type()` built-in member function shifts the compile-time perspective to the left. Whereas the `typeid` of a run-time class is `type_info`, the meta-type of a compile-time meta-object is `meta_type`.

The meta-class

The compile-time representation of a class comprises two meta-objects, one class meta-object to describe the class, and one instance meta-object to describe each instance. Since these two meta-objects have matching inheritance and always exist as a pair, it is convenient to regard the pair of meta-objects as a single object, which may be safely but loosely referred to as the meta-class.

Other meta-types

Figure 3.11 is drawn for classes and their instances. The same diagram applies for all meta-objects, replacing ‘instance’ by the meta-object category. Only variables, functions, meta-variables and meta-functions have their own user-defined meta-functions and meta-variables.

A function meta-object, of meta-type `function` or `function_specifier`, contains a function description (the parameters). The corresponding function meta-meta-object contains the mapping of all meta-names applicable to the function including any meta-variables defined within an *object-statements-clause*. The function body is indirectly defined via the `value` meta-variable.

Similarly, but taking care to use distinct terminology, a meta-function meta-object, of meta-type `meta_function` or `meta_function_specifier`, contains a meta-function description (the parameters). The corresponding meta-function meta-meta-object contains the mapping of all meta-names applicable to the meta-function including any meta-variables defined within an *object-statement-clause*. The meta-function body is indirectly defined via the `value` meta-variable.

The distinction between meta-functions and function meta-objects is shown in Figure 3.6 on page 79.

Meta-inheritance

C++ supports instance objects and, to a limited extent, class objects at run-time. FOG extends C++ to support class objects and, to a limited extent, meta-class objects at compile time. FOG provides a meta-class for every class and built-in type. The inheritance of meta-classes mirrors that of the class hierarchy, so that the meta-base-classes of every meta-class are the meta-classes of the base classes of the corresponding class. Every meta-class without any other meta-base-classes automatically inherits (virtually) from the built-in meta-class `::auto`. Additional meta-inheritance may be specified by using the `auto` keyword as an access-specifier. The inheritance and meta-inheritance for

```
class Base {};
class Derived : public Base, auto char {};
```

is shown in Figure 3.12. With the exception of meta-inheritance from `::auto`

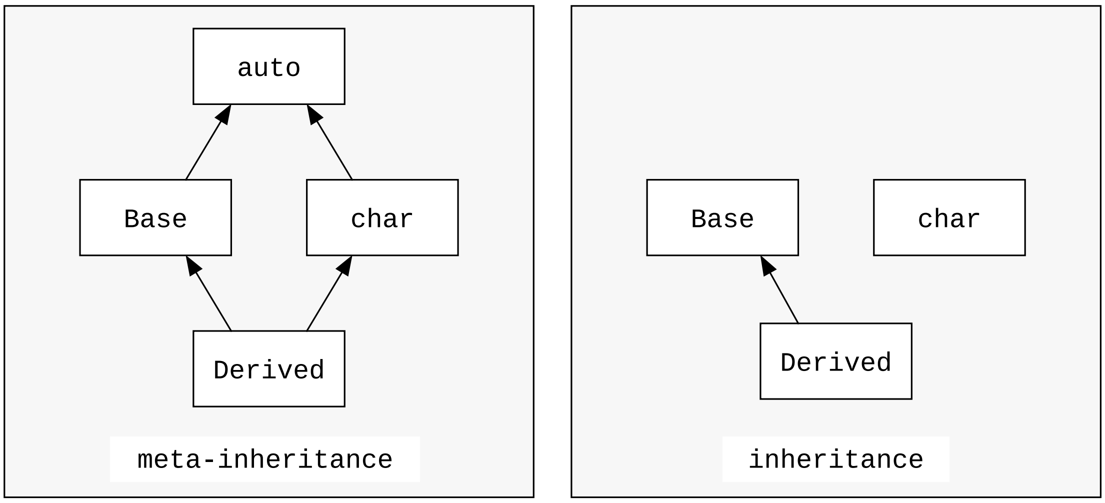

Figure 3.12 Inheritance and Meta-inheritance

which is always virtual, meta-inheritance is defined by the `virtual` keyword. A virtual meta-class appearing more than once in the meta-inheritance hierarchy only contributes one set of meta-variables.

3.6 Meta-programming

The meta-construction, meta-main and meta-destruction compilation stages (shown in Figure 3.1 on page 60) consist exclusively of application functionality. There is no built-in functionality.

During the meta-construction stage, the meta-constructor of each class (including built-in types) is invoked once in a least-derived first order. Programs may supply code for execution during this stage by declaring a meta-constructor:

```
auto MyClass::MyClass()
{
    /* meta-program */
}
```

This code is composed with any other definitions of the same meta-constructor. Such composition includes inherited contributions, which are executed first in accordance with the normal principles of constructors. Since everything inherits from the `auto` class, definition of a meta-constructor for `auto` provides a mechanism for executing a meta-program in all classes.

The meta-main stage consists of execution of all definitions of:

```
auto nil main()
```

The meta-destruction stage mirrors the meta-construction stage. The meta-destructor for each class is executed once, again in least-derived first order, but with inherited contributions executed after local contributions in accordance with destructor principles.

It has been recognised that the operation of two apparently independent meta-programs on the same program can lead to poorly defined behaviour. For instance, consider two meta programs which

- add a diagnostic print-out to every function to create a call-trace
- add a `check_invariant()` and invocation from each non-const function

Whether a diagnostic print-out is added to the `check_invariant()` routine depends upon the application order. Whether such a print-out should be added is a subtle user preference.

The distinction is fairly trivial in this example. The distinction is critical for applications that involve synchronization or persistence.

The problem is largely ignored in practical reflective systems. It is assumed that the meta-programmer will coordinate multiple meta-programs. The theoretical problem is addressed by the reflective tower [Smith84], in which each level of reflection defines a new language for the level above that hides the language of the level below. The rather impractical need for a distinct representation of each object at each level is described by [Chiba96].

The three stages offered by FOG perhaps represent a pragmatic compromise and symmetry with run-time concepts of static-construction, main-program and static-destruction. During the meta-construction stage, actual declarations are in a highly unstable state, since further meta-programming may provide additional declarations. It is therefore unwise to place any code in a meta-constructor that browses child declarations. Meta-construction code should consist only of definitions. Browsing meta-programming should be implemented in meta-destructors, and meta-programs should avoid creating new declarations during meta-destruction in order to support consistent behaviour by other meta-destructors. Meta-destructors should only elaborate and compose with existing declarations. The meta-main stage is not strictly necessary, however it avoids the need for relatively arbitrary meta-programs to be constrained by the invocation mechanisms of meta-construction or meta-destruction.

The non-trivial examples in Chapter 6 make extensive use of the meta-construction and meta-destruction phases to realise each example. It would appear that the two tree traversals are insufficient to support multiple meta-programs. It is essential for all of one meta-program to execute before any of the next, since use of just meta-construction and meta-destruction phases requires interleaved meta-program execution as the inheritance hierarchy is descended. Additional tree traversals can be realised by performing a hierarchical traversal starting with an iteration over `::all_classes()`, with each meta-program extending the meta-main program to perform traversals. Sequencing of these meta-programs could be achieved by maintaining a list of passes in a meta-variable maintained by user code, or by some programming tricks exploiting meta-inheritance to establish an ordering. These approaches are either cumbersome or cute. A more direct mechanism for specifying multiple meta-programs and their sequencing dependencies seems desirable. Eliminating the existing use of meta-constructors and meta-destructors also avoids a syntax conflict if FOG is extended to support user-defined meta-types, since such types should obviously have constructors and destructors.

3.7 Syntax macros

The superset parsing approach described in Chapter 4 isolates the syntactic and semantic analysis stages. This supports an implementation in which all syntactic processing completes before any semantic processing starts. Enforcing this isolation prohibits any syntax dependency on semantics, and unfortunately prevents the definition of syntax macros.

A syntax macro supports (or rather gives the illusion of supporting) user-defined language extensions[6]. Such an extension could in its general form support arbitrary additional syntax so that:

```
with counter from 1 to 100 step 5 in { static int $counter; }
```

could be recognised as an alternate way of writing

```
for (int counter = 1; counter < 100; counter += 5)
    { static int $counter; }
```

More practically syntax macros should support what appear to be extra *decl-specifier*s

```
synchronized class MyClass
{
    persistent int _count;
};
```

so that programmers appear to use an extended language, although the *synchronized* and *persistent* extensions are realised by meta-programming in a standard language. The example is clearly more readable than the functionally equivalent:

```
$synchronized(class MyClass
                {
                    $persistent(int _count);
                };)
```

In both cases application meta-functions are invoked to support the concepts of multi-process synchronisation or data-base persistence. The syntax macro approach has the advantage of offering a much more acceptable programming interface.

Implementation of a syntax macro requires the trigger word (`persistent`) to be recognised in order to perform the appropriate syntactical parse.

Recognition of the trigger word in an unconstrained context requires the trigger word to become a new reserved word, introducing the problems of conflicting usage and unwanted replacement associated with the C preprocessor.

Alternatively, recognition of the trigger word within a restricted syntactical context imposes the implementation problem of executing yacc at compile-time to generate an updated syntax analyser, and the practical problem of enabling the application programmer to understand the shift-reduce conflicts associated with a proposed macro. Resolution of these conflicts is unlikely to be portable. The complexity of generating and diagnosing an updated analyser at compile-time are inappropriate for a language with as difficult a syntax as C++.

A potential solution to the problem of conflicting name capture lies in the use of the C++ name hierarchy. A syntax macro could be defined within a namespace or

6. The extensions are shown by italicizing the type-writer font.

`class` and would only be a reserved word within that namespace or `class`. However this approach has two problems:

A reserved word does not necessarily occur lexically within its class:

```
    persistent int MyClass::_status;
```

would have to be written:

```
    class MyClass
    {
        persistent int _status;
    };
```

This is inelegant but could perhaps be tolerated.

The class may be indeterminate:

```
    $do_something(class MyClass
                {
                    persistent int _status;
                };
               );)
```

Since the context of `MyClass` is undetermined, it cannot be known whether *`persistent`* is a scope-dependent reserved word and so syntactic analysis is not possible.

This is also inelegant and could perhaps also be tolerated, but the first example indicates that scope-specific syntax macros do not extend C++ comfortably. The second example shows an incompatibility with other FOG concepts.

A syntax macro must therefore be scope-independent, and so have the same status as any other reserved word. Syntax macros should be restricted to applications where the benefit of the cleaner invocation far outweighs the hazards of the introduction of a global name.

Definition of a syntax macro should integrate with the rest of the language, and necessarily occurs at global scope. However, usage of a syntax macro requires semantic analysis of its definition to have been completed. Therefore preserving the implementation option of completing syntactical analysis before starting semantic analysis requires semantic analysis of *syntax-macro-definition*s to occur during syntax analysis. Syntax macro definitions should therefore have a distinctive syntax to facilitate this special treatment.

syntax-macro-definition:
    `explicit auto` *meta-type identifier* `(` *syntax-macro-parameter-list*$_{opt}$ `)` *exposed-tree*$_{opt}$
        compound-tree-statement

syntax-macro-parameter-list:
    syntax-macro-parameter
    syntax-macro-parameter-list `,` *syntax-macro-parameter*

syntax-macro-parameter:
    meta-type identifier exposed-tree$_{opt}$
    identifier
    reserved-word
    punctuation

The further overloading of `explicit` and `auto` is unpleasant and only slightly mnemonic. Perhaps using a linguistically consistent but different meaning for the `register` reserved word would be better. Introducing a new reserved word such as `syntax` would be better still.

Declaration of a syntax-macro declares the *identifier* to be a reserved trigger word for a sentence that should satisfy the *meta-type* syntax. The syntax to be accepted by the parser comprises this trigger word followed by the sequence of *syntax-macro-parameter*s, which comprise expected syntax elements and further words or punctuation which are temporarily reserved between recognition of the trigger word and detection of the end of the syntax.

It is a slightly surprising but fortunate accident that this syntax supports specification of any combination of intervening punctuation including , or). Thus the syntax-macro to intercept and pack a fractional coordinate such as

```
    pt(0.5, -0.7)
```

could be specified as

```
explicit auto expression pt( (, assignment_expression[7] x, ,,
                                                 expression y, ))
    { (int(32768 * $x) << 16) | (int(32768 * $y) & 0xFFFF); }
```

using an overline to distinguish tokens defining the structure of the definition from those parameterising the definition.

Overloading is permitted, subject to the constraint that the set of temporarily reserved words is the union of all syntaxes triggered by the overloaded trigger word.

```
explicit auto variable_specifier persistent(variable_specifier v)
{
    $v;
    /* additional meta-programming using $v */
}
explicit auto class_specifier persistent(class_specifier c)
{
    $c;
    /* additional meta-programming using $c */
}
```

When the trigger word is recognised, the additional reserved words are temporarily treated as tokens distinct from *identifier*s ensuring that they serve to terminate preceding syntax. Syntax-macro parameters are then identified one at a time. For each parameter, a one token lookahead is used to see whether the explicit *identifier*, *reserved-word* or *punctuation* requirement of a parameter can be satisfied. If satisfied, the lookahead is discarded, overload alternatives without explicit requirements are discarded and the scan continues looking for the next parameter. When no explicit requirement can be satisfied and a meta-typed parameter is required, a recursive syntactical analysis is invoked to locate a generic syntax element. Overload alternatives that the generic element satisfies are retained. Eventually:

- no alternatives remain:
  a syntax error has been detected
- one alternative remains and it requires no further parameters:
  the syntax-macro arguments have been successfully identified
- more than one alternative remains:
  an ambiguous invocation has been detected

The temporarily reserved words are restored to their previous status after processing of the syntax macro. This may still be a reserved status since syntax macros can be invoked recursively.

Care should be exercised in the use of `{` `}` and `;` as specific punctuation since FOG uses these for recovery from syntax errors, which may be hampered by unconventional usage of these tokens.

A syntax-macro is functionally the same as a meta-function. It differs only in its invocation syntax. A meta-function invocation has a trigger operator (`$` or `@`) and an expression identifying the meta-function name followed by parenthesised comma-separated generic syntax elements. A syntax macro has a trigger word

7. `assignment_expression` rather than `expression` is necessary to ensure that `pt(0,0)` is not treated as `pt((0,0)` **missing**), since `0,0` is an expression.

identifying the macro followed by a sequence of user-defined punctuation, (temporarily) reserved words and generic syntax elements.

4 Parsing

This chapter deals with practical problems encountered and the reasons for the solutions adopted while developing the parser for FOG. Most of the problems are related to C++ and so the new superset grammar parsing approach is described primarily from a C++ perspective. The FOG enhancements extend the grammar but do not add to the fundamental parsing difficulties.

The traditional technology, structure and terminology of a compiler are briefly outlined in order to provide some grounding for readers not well-versed in compiler fundamentals. The dragon book [Aho86] is the recognised authority.

Typical approaches to parsing C++ are discussed, the choice of parsers available as a basis for the FOG parser is reviewed and then the evolution of FOG from a very simple to fully fledged parser is described.

The potential and actual ambiguities encountered by a C++ parser are analysed, by way of demonstrating an extended form of regular expression notation that supports reasoning about grammars.

The syntax generalisations of the superset grammar are described and the new notation is then used to justify the soundness of the superset parsing approach. The approach eliminates many of the context-dependent problems from C++.

A few details of the code structure are provided to demonstrate the high degree of isolation between the parsing stages.

Finally some size metrics are produced to compare the C++ parsing approaches and estimate the extra cost of the FOG enhancements.

Readers particularly concerned about the language details may care to browse the on-line penultimate working draft [C++96] of [C++98] at

`http://www.maths.warwick.ac.uk/c++/pub/wp/html/cd2`

or print off a copy of Annex A from

`http://www.maths.warwick.ac.uk/c++/pub/dl/cd2/CD2-{PDF,PS}.tar.Z`

4.1 Terminology

Figure 4.1 is based on Figure 1.9 of the dragon book [Aho86]. It shows the typical components of an application that translates a source program into a target program.

Successive stages of analysis extract the meaning of the source program, enabling an intermediate representation to be built, optimised and then converted to the required target program. A symbol table maintains information to be shared between stages. The error handler supports generation of error messages in as helpful a fashion as possible.

The analysis is broken into three stages. The lexical analysis identifies and validates individual lexemes (words), the syntax analysis identifies and validates grammatical constructs (sentences), and the semantic analysis validates the meaning of each construct within a wider context.

Lexeme is the normal term for the product of lexical analysis, the same concept is more commonly referred to as token in the context of syntax analysis. However syntax presents the greater technical challenge and so the term token is preferred in this thesis.

In C and C++, the preprocessor provides additional translation. It may be implemented as a separate program, or as an additional stage between lexical analysis and syntax analysis. The C++ standard treats the preprocessor as an extra stage and uses the term preprocessor token to describe a lexeme that passes from lexical analysis to preprocessor, and the term token to describe a lexeme passed from preprocessor to syntax analyser.

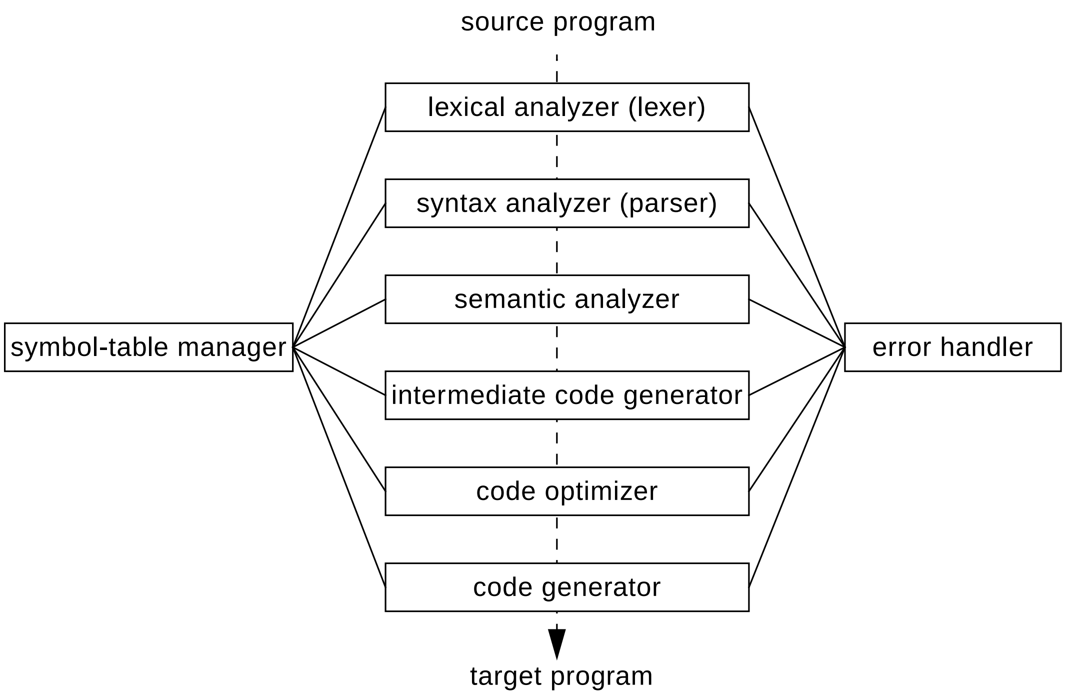

Figure 4.1 Compiler Translation Stages

Lexical and syntactical analysis were once the hardest parts of a compiler to write, however with the advent of standard lexer and parser generator tools, these stages are now relatively easily automated.

The lex program converts a grammar specification in the form of a number of regular expressions into a state machine or DFA (Deterministic Finite Automaton) that accepts characters one by one from a source file and emits a lexeme for each analysed word. lex [Lesk75] is the standard tool. flex [Levine90] is a more polished version distributed as part of the GNU tool set.

The yacc program converts a grammar specification in the form of BNF rules into a LALR(1)[1]shift-reduce parser that demands lexemes one by one from the lexer and invokes action routines as each rule is reduced. yacc [Johnson75] is the standard tool. bison [Levine90] is a more polished version distributed as part of the GNU tool set.

The flex++ [Coëtmeur93b] and bison++ [Coëtmeur93a] variants were used for FOG. The variants encapsulate the generated lexer or parser as a C++ class, and so readily support multiple and re-entrant lexers and parsers. FOG uses one lexer grammar, four (tiny) parser grammars for ANSI C preprocessing and one (huge) parser grammar for extended C++ parsing. A re-entrant lexer and parser may be created by `std::parse` to support deferred character-level substitution.

The operation of the state machine generated by an LALR parser is extremely simple comprising just four actions for each possible next token.

- accept the token as the termination of a sentence in the grammar
- reject the token as inconsistent with any sentence of the grammar
- shift to another state, deferring any decision
- reduce following recognition of a rule

Parser generators allow application code to be supplied for execution when a rule is recognised. This code will typically create a data structure that describes the

1. 1 token Look-Ahead, Left-to-right scanning of input, Right-most derivation in reverse.

information that has just been parsed. In compiler applications, these data structures are highly recursive and well represented by a tree structure. The structure is called an Abstract Syntax Tree.

In the following very simple yacc grammar, there are 7 rules leading to 4 productions. Each (production or reduction) rule has a non-terminal at its left-hand side and may use terminals or non-terminals on its right-hand-side. `Identifier` and `'*'` are terminals.

```
term:        Identifier                { $$ = create_identifier_node($1); }
    |        Number                    { $$ = create_number_node($1); }

product:     term                      { $$ = $1; }
    |        product '*' term          { $$ = create_multiply_node($1, $3); }

expression:  product                   { $$ = $1; }
    |        expression '+' product    { $$ = create_sum_node($1, $3); }

grammar:     expression ';'            { $$ = $1; }
```

There are two rules for the production of a `term`. One from an `Identifier` and one from a `Number`. A `create_`*`xxx`* routine is associated with each rule using the special $*n* variables to access inputs, and $$ to propagate a result.

The source sentence

```
    a + b * 5 ;
```

is parsed to create the AST

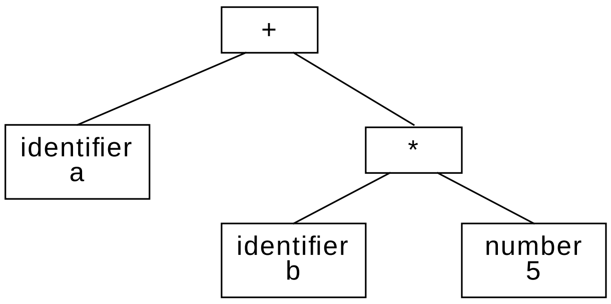

Figure 4.2 Abstract Syntax Tree

Shifts occur to advance to states that reflect a partial parse following each of +, b and *. Reductions occur as each rule is recognised

- one reduction after the a
- one reduction after the b
- three reductions after 5, for each of the rules
  - `Number` to `term`
  - `product * term` to `product`
  - `expression + product` to `expression`.

Each of these reductions activates the application code that creates the AST nodes. Once the external textual representation has been converted to an internal AST form, the program can be manipulated by compilation code to perform whatever checking, correction, rearrangement or optimisation is necessary to perform the translation.

An unambiguous grammar provides for only one possible parse tree for a given input, although a potentially infinite amount of lookahead may be required to distinguish between alternative partial trees for partial inputs. Conflicts arise from an attempt to distinguish alternatives prematurely. A reduce-reduce conflict arises if there are two alternate simplifications available. A shift-reduce conflict arises between an elaboration and a simplification of context.

A parser generator has a policy by which conflicts are resolved to produce a deterministic, although not necessarily useful parser. The default resolution of conflicts may be controlled by the programmer using the concept of precedence. The name reflects its original use to resolve precedence problems with arithmetic operators. The concept is of more general use. A special `%prec` non-terminal may be used to specify resolution of a conflict.

Translation programs are frequently presented with source files containing errors. As many of these errors as possible should be detected and diagnosed in a helpful fashion. It is rarely acceptable for a translator to just stop and report that a “parse error” has been encountered somewhere. The analysis must therefore continue after an error has occurred. In support of this philosophy, yacc suspends analysis and generates a special `error` token when an error is encountered. A carefully written grammar can make use of `error` to control resumption of the analysis.

4.2 Approaches to C++ Parsing

An LALR(1) parser generates a table driven parser for an unambiguous context-free grammar, subject to the requirement for detection of the right-most edge of a grammar production with 1 token of lookahead. The C++ grammar is ambiguous, context-dependent, and potentially requires infinite lookahead to resolve some ambiguities. Other ambiguities are resolved by the language definition.

An LALR parser is not an obvious match to these requirements. However the alternatives are worse. A (bottom-up) LALR parser is faster and more compact than an LR parser, and able to handle all grammars that could be handled by a simpler (top-down) LL parser, and so the most widely used parsers are based on LALR(1).

In order to satisfy the constraints of an LALR(1) parser, the ambiguities, context dependence, and lookahead problems of C++ must be resolved.

The dragon book [Aho86] recognises that the boundaries between lexical, syntactic and semantic analysis are not clear cut.

Traditional C++ approaches seek a correct high resolution parse. As a result, the boundary between syntactic and semantic analysis has to be shifted to exploit semantic information during syntactic analysis by the parser and to leak semantic information through to the lexer. Use of semantic information during syntactic analysis requires very tight coupling to ensure that scope context is honoured and that changes of name visibility in mid-statement are correct. [Roskind91] provides a particularly unpleasant example where a change of classification midway through an apparent declaration leads to a contradiction, that is only resolvable as an expression.

The two variants of the new parsing approach described in this chapter do not move the boundary. The parser proceeds without full semantic knowledge and produces a result that is syntactically consistent, but sometimes semantically incorrect. An additional pass is therefore added to the semantic analysis to correct the inaccuracies of the syntactic analysis.

The advantages of this approach

- elimination of type (and template) tagging
- much simpler grammar that more closely follows the standard
- clear separation of syntactic and semantic processing
- conversion of syntactic ambiguities to semantic ambiguities
  - disentangles resolution from grammar implementation
- conversion of syntactic errors to semantic errors
  - avoids some losses of synchronization
  - provides more opportunities for helpful error diagnostics
- probably very slightly smaller code size

The disadvantages of this approach

- introduction of new ambiguities
- an additional semantic correction pass
- additional semantic validity checking
- probably very slightly slower

Two variants of the approach are described. The previous implementation (available on the net) uses multiple passes to resolve ambiguities. The current implementation uses a superset grammar approach enabling operation without type or template information, as is necessary to support meta-programming consistently (Section 3.2.3).

The context-dependencies of C++ are described in detail in Appendix F.2.1.

4.3 Alternatives

Before developing the FOG parser, a brief review of the available alternatives was made. These alternatives are described in this section, along with two others that were not known at the time. Unfortunately, the developers of commercial C++ compilers do not make their parsers freely available in the public domain, and so the many proprietary implementations cannot be considered. However, comparison of the public domain approaches is sufficient to shed useful light on the difficulties.

4.3.1 Roskind grammar

A yaccable C++ 2.1 grammar was made available by Jim Roskind [Roskind91]. This grammar dates from 1991 and has not been updated to handle C++ facilities such as templates or exceptions foreshadowed by the ARM [Ellis90], or to incorporate concepts such as `bool` and namespace added during standardisation. The paper accompanying the grammar provides a very insightful discussion into the source of the parsing problems and some rather pathological examples, whose correct interpretation is debatable.

The grammar resolves context dependence by a “lex hack”, so that the lexer classifies identifiers as either IDENTIFIER or TYPEDEFname. The paper notes the need for another such hack to resolve template names.

The grammar is no more than a grammar. There is no action code to react to successfully analysed constructs, and only dummy hooks at the places where symbol table maintenance must be performed. The grammar code has no error recovery.

Ambiguities are resolved but are not removed from the grammar code. Some ambiguities are eliminated by rewriting parts of the grammar. Others are carefully analysed to ensure that the default ambiguity resolution policy of the parser generator chooses the required alternative.

Some potential lookahead problems are resolved by structuring of the grammar code to recurse on the right-hand side, or by flattening out, in each case deferring reductions until more context has been seen. Other lookahead problems are resolved using the disambiguation policies of the previous paragraph. Further problems could have been resolved by more flattening, but were perceived to not merit resolution while there was a possibility that the C++ grammar could change.

The grammar has a total of 24 shift-reduce and 18 reduce-reduce conflicts, originating from 11 ambiguities. (An ambiguity can result in more than one conflict, since conflicts occur between states; more than one state transition may be affected by an ambiguity.)

4.3.2 gcc

The GNU C compiler [Stallman98] has evolved to handle Objective C and C++. The compiler continues to improve, and is close to the C++ standard, but currently (version 2.8.0) experiences significant problems with template instantiation, because of the lack of a compilation database. These problems would not affect the use of gcc as a foundation for FOG.

The gcc compiler is portable to a very large number of operating systems, on which a build process normally involves compiling gcc through the local compiler, then recompiling gcc using the potentially better optimisations of gcc. The need to bootstrap through the local compiler requires extensive conditionalisation so that the gcc sources avoid the defects of all known compilers. gcc source code is therefore harder to read than it might be. gcc source code is necessarily written in C, and so lacks the modularization and polymorphism that can be achieved using classes and Object Orientation in a language such as C++. The internal data structures of gcc comprise a tree node that is a union of all possible expressions, operators, names, declarations, statements, files and etc.

The gcc compiler is a complete compiler including good error recovery and diagnosis, and full symbol table maintenance. gcc is recognised to be of production quality.

The lexer is hand coded and makes a seven way categorisation of identifiers to disambiguate the subsequent parser. However the subsequent parser requires 109 `%prec` directives to resolve 704 conflicts explicitly, leaving 5 shift-reduce and 38 reduce-reduce conflicts to be resolved automatically.

4.3.3 CPPP

A C++ parser was developed at Brown University, as a general purpose tool for which a variety of applications were foreseen [Reiss95]. This parser has steadily evolved, however the most recent version available on the net is version 1.82 from 1996. It would appear that development stopped before facilities such as `namespace` or `bool` were implemented.

CPPP achieves a higher degree of decoupling between lexer, parser and database than gcc, and has a grammar that closely resembles the published C++ grammar. CPPP comprises three stages: a lexer, a lookahead parser, and a main parser. The lookahead parser recognises potentially ambiguous constructs and invokes custom parsing routines to look sufficiently far ahead to resolve the ambiguity. Additional tokens are inserted into the token stream so that the subsequent parser proceeds without ambiguity.

The P++ developers [Singhal96] report extending CPPP successfully, but only after resolving a fair number of bugs. CPPP was also used for Iguana [Gowing96].

The grammar has only one unresolved shift-reduce conflict, but uses precedence extensively (22 `%prec`) to suppress a further 410 conflicts. The high number of conflicts is misleading. Most are the result of flattening the expression syntax and using grammar precedence to implement arithmetic precedence. This results in a faster parser since an expression term is reduced just once, rather than once at each of the ten binary operator precedence levels of the C++ grammar.

4.3.4 PCCTS

The Purdue Compiler Construction Tool Set provides alternative versions of lex and yacc called DLG and ANTLR. ANTLR is an LL(k) parser generator. Use of LL principles provides the freedom and (for practical grammars) the necessity of incorporating semantic resolution within the parse. A C++ grammar is available for use with PCCTS [Lilley97], but is heavily disclaimed as initial and experimental. A significantly customised version of ANTLR is required by the grammar.

4.3.5 C++ to F-code translator

The work described in this thesis concerns meta-compilation and the FOG implementation. The inspiration for FOG arose from research aimed at the development of a compiler implementing optimisations appropriate to DSP processors. This compiler was to use F-code [Muchnick93] as its intermediate representation. Some work was performed on a C++ to F-code translator as part of this earlier research. At that time, there was no knowledge of the CPPP or PCCTS grammar and so there was an implementation choice between

- the out-of-date Roskind grammar (42 conflicts)
- the tightly coupled gcc grammar (747 conflicts)
- a custom solution

The author had previously extended gcc to perform automated documentation generation for C++ code. This had merely required a late read-only traversal of the internal data structures. The difficulties of debugging with union nodes, the lack of clear documentation on the semantics of each node, and the enormous 250,000 line size of the code indicated that wholesale extension of gcc could lead to considerable problems.

A simpler alternative of just reusing the gcc parser grammar was examined. Examination of the parser showed that the parser, lexer and program data base were too closely coupled, making separate re-use of the grammar alone impractical.

The seemingly large number of unresolved conflicts, out-of-date character and obscure coding of declarators in the Roskind grammar discouraged its use. Recognition that the introduction of tree-literals would impact the heart of the grammar indicated that a clean grammar should be the starting point. A custom solution seemed the only alternative.

4.4 FOG parsing

The original aims of FOG concerned facilities for

- automatic insertion of repetitious code into class declarations
- elimination of redundant source text
- support for algorithm-centric modularization

The first aim is satisfied by derivation rules.

The others correspond to what is now known as weaving in the Aspect Oriented Programming world.

FOG/1 - superficial guided parse

It was perceived that these goals could be satisfied by a very simple parser assisted by extra keywords (guides). The first implementation therefore explicitly annotated C++ source adding new reserved words such as `constructor`, `function`, `variable` and `type` for the guides. Lines that needed special treatment could be easily identified and other lines copied from input to output without interpretation.

```
constructor Class::Class(int aSize) : _size(aSize) {};
```

This approach imposed a language incompatibility. For a legal C++ source file to be acceptable to FOG/1, it was necessary to manually add the guides to every declaration.

A second problem arose as to what syntax to use following the guide. The C++ declaration syntax is complicated and highly recursive. There are better syntaxes than the C++ syntax [Werther96], but they look very out of place in a C++ program.

The C++ declaration syntax was preserved, and the problem of recursion of function signatures was partially solved by treating the function signature as unparsed text to be copied though to the output. This hid the problem for most functions but could not cope with pointers to functions.

```
variable int (*v)();
function int (*f())();
```

FOG/2 - pragmatic guided parse

Resolution of the function signature problem mandated an accurate parse of the signature. This problem had already been solved as part of the C++ to F-code translator and so the relevant part of the grammar was re-used. C++ function signatures comprise parameter types, names and optional default values. The default value was initially left unparsed, with the text copied unchanged to the output.

```
function void f(int a = unparsed_text);
```

This usually works, but requires recognition of the `)` or `,` that terminates the initializer. The initializer is an expression and so is subject to the use of templates. A simple parse of the initializer in

```
template <bool T1, int T2> class B;
function void f(int a = B < c, 5>);
```

may identify the comma-terminated `B < c` as the first initializer, before misinterpreting the residue. It requires the knowledge that `B` in the unparsed text is a template to correctly resolve the instantiation. Reliable parsing of declarations requires reliable parsing of expressions too.

FOG/3 - pragmatic full parse

At this point it was becoming clear that many of the complexities of C++ parsing could not be avoided in FOG, and it seemed likely that every attempt to avoid a complexity would introduce a deficiency. This is a far from unique discovery. There are a number of commercial development tools that have taken short-cuts to parsing C++, with the result that interesting C++ programs are misinterpreted. The class browser of Microsoft Visual C++ is just one example.

Another large segment of the C++ to F-code grammar was therefore added to FOG, so that FOG contained most of the C++ grammar. The traditional C++ declaration / expression ambiguities did not (yet) arise, because expressions occurred only in the limited context of initializers.

The need for guides was a barrier to porting existing C++ code to exploit FOG. Some of the first code to be ported from C++ to run through FOG was some compiler code. This code made use of the guides as function names. Requiring that the member function `type()` be renamed did not seem sensible.

With the increasingly accurate C++ parsing in FOG, it was no longer essential to have the extra guides. Parsing was no longer made easy by their use.

Once the extra guide keywords had been eliminated, there were few fundamental differences between FOG and C++ syntax. Each of these was challenged and eliminated, with the result that the FOG grammar is an almost pure[2] superset of C++.

$-parsing

Tree-literals were originally envisaged as being resolved by a preprocessor before the real code emerged. A tree-literal was therefore recognised very early and

2. Some very minor exceptions are listed in Section 2.3.

replaced. Replacement could occur anywhere, including within strings. The semantics of the replacement were purely lexical, and so the replaced text could contribute partial strings and partial reserved words. This was flexible, powerful, undisciplined and awkward to implement. The ability to handle partial tokens required an unpleasant ability to recurse earlier lexing stages. The implementation resulted in a very complicated lexer, that had to maintain a stack of states according to how far through a string / character / number it was when a replacement started. It was far from clear that implementation would behave correctly under perverse usage.

Recognition that the ANSI C string concatenation could be generalised to concatenation of adjacent textual tokens revealed that there was no need for substitution within strings:

```
        "built at $time on $date"           // Not FOG
```

It could happen almost as easily between strings:

```
        "built at "$time" on "$date         // FOG
```

This then enabled the lexer to be more disciplined; preprocessor tokens could be identified first, then substitution could occur token by token, with only a minor complexity in retokenizing the result of a concatenation. Retokenisation was eventually discarded as unnecessary (Section 2.1.1.4).

Ambiguity resolution

Removal of the guides requires the syntaxes for typedefs, variables and functions to coexist. Ambiguities arise:

```
    T ( A );
```

could be a constructor for T with an unnamed parameter of type A. Or it could be a variable of type T with the redundantly parenthesised name A. This problem is traditionally resolved by ensuring that the lexer has the semantic information available to classify T and A as *class-name* or *type-name* or *identifier*. In C++, this classification requires accurate scope context and symbol table maintenance to ensure that an *identifier* is classified as a *type-name* at the declaration point.

Type information was not being used in the FOG parse and it seemed desirable to continue not using type information. The few ambiguities that appeared as a result were resolved by back-tracking, which is described in Section 4.8.

Development of the C++ to F-code translator had revealed how difficult resolving the expression / declaration ambiguity was. The grammar grew unpleasantly large as productions are elaborated to create sub-productions without ambiguities. Back-tracking was introduced to support sequential rather than concurrent consideration of alternatives.

Since FOG was then only analysing declarations and meta-statements, the major C++ ambiguities did not arise, only minor problems and implementation inconveniences. It was not necessary to perform semantic correction of syntactic errors, the parse-declarations-first policy disambiguated adequately.

This summarises the evolution leading up to the previous implementation of FOG that uses back-tracking to resolve all ambiguities. Semantic leakage is limited to template names. Type information is not used.

Superset

Consideration of the performance overheads associated with marking and back-tracking for every meta-statement, and of the validity of not using type information led to the more efficient superset parsing approach described in this chapter. The grammar for this approach has been implemented, cross-checked for completeness against the C++ standard, and processed by both bison and yacc to show lack of fundamental ambiguity, and successfully used to parse C++

programs. The bison report file has been used to verify correct resolution of the 24 conflicts that result from the 4 residual C++ ambiguities, 1 introduced ambiguity, and 5 implementation artefacts. These ambiguities are summarised in the comment header of Appendix B and described in Appendix F.2.

The ambiguities and semantic corrections resulting from the lack of type information apply to this superset parsing approach, and to a lesser extent to the multi-pass parsing approach.

4.5 Analysis of the C++ Grammar

The parsing approach described in this thesis deviates from accepted practice. It is therefore necessary to justify that the approach is sound. In order to do this we must first understand the standard problems and then identify any new problems before showing how the new approach resolves them. This requires a fairly detailed examination of some aspects of the C++ grammar and the introduction of a notation that supports reasoning about that grammar.

4.5.1 Notation

The dragon book [Aho86] describes two different notations for defining languages.

Regular Expressions

The simple regular expression notation supports description of classes of character sequences and is the basis for the lex lexer generator. An identifier can be defined by

```
    [A-Z_a-z][0-9A-Z_a-z]*
```

The expression starts with a character from the class containing the alphabetic characters or underscore and continues with an arbitrary number of repetitions of the second class that adds the numeric characters. This is a compact notation but unable to express recursion and so cannot describe the language of matched nested braces (`{}`, `{{}}`, `{{{}}}` etc.). The inability to specify matching delimiters prevents the use of regular expressions to define most (if not all) programming languages; the use of matched parentheses to enforce arithmetic precedence is almost universal.

Context-Free Grammars

A (Context-Free) Grammar (CFG) is used to describe a more complicated language and is the basis of the yacc parser generator. A grammar is defined by rules that operate on the terminals and non-terminals of the language. Terminals correspond to the input tokens, non-terminals appear as the left-hand side of production rules involving terminals and/or non-terminals on their right-hand side. The brace language may be defined using { and } as terminals and *braces* and *grammar* as non-terminals. *grammar* is the distinguished non-terminal that defines the language.

grammar: *braces*
| { *grammar* }

braces: { }

The grammar is specified using a Backus-Naur Form. Alternative rules producing the same non-terminals are separated by `|`. When a clear multi-line formatting policy is used as in the C++ standard, the `|` may be omitted. Rules are variously referred to as production rules, or reduction rules.

The availability of intermediate non-terminals gives Context-Free Grammars much greater power than regular expressions. However the requirement to use multiple rules and the transformation of repetition into recursion makes it difficult to reason about the grammar.

Hybrid notation

C++ comprises two relatively independent subgrammars, one to define expressions and another to define declarations. It is well known that there are sentences such as.

```
    int (var);
```

that are ambiguous. It could be a functional cast of the variable var to an integer value or a declaration of the redundantly parenthesised var as an integer. Identifying these ambiguities in the grammar is difficult because a few hundred inter-related rules are not a convenient representation for logical reasoning.

We need to be able to substitute one rule in another in order to derive the rules that identify each non-terminal with respect to terminals, or relatively fundamental non-terminals. We will therefore extend regular expressions to support C++ syntax. We can then represent an expression as one extended regular expression, a declaration as another, and identify the ambiguities by comparing their terms.

Both regular expressions and context-free grammars can describe the concatenation of lexically adjacent elements, but only context-free grammars support a complex ordering through nesting of non-terminals. We therefore introduce •, a functional operator to support more arbitrary ordering in regular expressions.

The • operator is always applied to a specific argument and so associates from right to left. This of course differs from the composition operation in functional languages, since we are interested in successive application, not in function composition.

The • operator has higher precedence than lexical concatenation. Thus:

$\alpha\ Z \bullet \gamma\,;$

represents α concatenated with the application of Z to γ. If Z denotes application of braces, and α and γ are the identifiers a and g. The above expression denotes the sentence

```
        a { g } ;
```

The nested braces language, using ε as the empty set of sentences, is

$Z\bullet\varepsilon$	{ }
$Z\bullet Z\bullet\varepsilon$	{ { } }
$Z\bullet Z\bullet Z\bullet\varepsilon$	{ { { } } }
$Z\bullet Z\bullet Z\bullet Z\bullet\varepsilon$	{ { { { } } } }
etc.	

which we abbreviate to

$Z^{+}\bullet\varepsilon$

More practically, we represent the pointer prefix of a declarator by $P_d(\zeta)$ and the array suffix of a declarator by $A_d(\eta^{?})$ so that

$P_d($`X::y::*const`$)\bullet A_d(5)\bullet v$

denotes

```
        X::y::*const v [5]
```

The subscripts in these names form part of a compact naming policy: P for pointer, A for array, P_d for a pointer valid for use in a declarator, A_e for an array valid for use in an expression.

Since the parenthesised parameterization is not usually significant in ambiguity reasoning, we can refer more simply to

$P_d\bullet A_d\bullet\theta_d$

where θ_d is any name valid in a *declarator*.

The full notation is summarised in the following tables

(Non-)Terminal	Notation	Description or example
Free (untyped) variables	α	
Built-in type (one word)	β	`int`
Declaration	δ	*simple-declaration*
The empty set	ε	
Pointer Type	ζ	*ptr-operator*
Assignment-expression	η	*assignment-expression*
Name in declaration	θ_d	*declarator-id*
Name in expression	θ_e	*id-expression* plus a bit
Name other than local destructor	θ	
Constant-expression	κ	*constant-expression*
Character, Number or String	λ	*literal*
Parameter-Declaration	π	*parameter-declaration*
Parameter-Declaration and Assignment-Expression	ρ	$\rho = \eta \cap \pi$
Type-name in declaration	σ_d	`size_t`
Simple-type-specifier	σ_e	*simple-type-specifier*
Type	τ	*type-id*
Generalised Assignment-Expression	χ	$\chi \supseteq \eta$
Generalised Parameter-Declaration or Assignment-Expression	ω	$\omega \supseteq \eta \cup \pi$

Table 4.1 Terminals and Non-Terminals

Operator	Notation	Description
Independent instances	$\alpha_1, \alpha_2, \alpha_3, \alpha_4$	
Zero or one	$Z^?{\bullet}\alpha$	α or $Z{\bullet}\alpha$
Zero or more	$Z^*{\bullet}\alpha$	α or $Z{\bullet}\alpha$ or $Z{\bullet}Z{\bullet}\alpha$ or $Z{\bullet}Z{\bullet}Z{\bullet}\alpha$ or etc..
One or more	$Z^+{\bullet}\alpha$	$Z{\bullet}\alpha$ or $Z{\bullet}Z{\bullet}\alpha$ or $Z{\bullet}Z{\bullet}Z{\bullet}\alpha$ or $Z{\bullet}Z{\bullet}Z{\bullet}Z{\bullet}\alpha$ or etc..
Exactly one of	$\begin{Bmatrix}\alpha_1\\ \alpha_2\end{Bmatrix}$	α_1 or α_2
Comma separated list of	$L{\bullet}\alpha$	${}^0L{\bullet}\alpha$ is ε ${}^1L{\bullet}\alpha$ is α ${}^2L{\bullet}\alpha$ is α_1 , α_2 ${}^3L{\bullet}\alpha$ is $\alpha_1 , \alpha_2 , \alpha_3$ etc.
Covers/Contains	$\alpha_1 \subseteq \alpha_2$	Every sentence of α_1 is contained in α_2
Strictly Covers / Contains	$\alpha_1 \subset \alpha_2$	Every sentence of α_1 is contained in α_2, and some sentence of α_2 is not contained in α_1
Intersection	$\alpha_1 \cap \alpha_2$	Sentences common to α_1 and α_2
Union	$\alpha_1 \cup \alpha_2$	Sentences of α_1 or α_2

Table 4.2 Mathematical Operators

There are no operators that apply to more than one argument, although operators may take additional parenthesised parameters. The list operator uses a pre-superscript rather than a post-superscript to highlight the distinction between the repetition of the argument many times and multiple application of the operator. The argument is repeated with no constraint between the arguments. Thus ${}^{2}L \bullet Z^{?} \bullet \alpha$ covers the 4 possibilities of Z present or absent independently for each argument whereas $Z^{?} \bullet {}^{2}L \bullet \alpha$ covers only the 2 possibilities of Z jointly present or absent.

Operator	Notation	Lexical Example
Array declarator suffix	$A_d(\kappa^?) \bullet \alpha$	α [ ]
Array expression suffix	$A_e({}^{+}L \bullet \eta) \bullet \alpha$	α [η]
Array for declarator and expression	$A_{de}(\kappa) \bullet \alpha$	α [κ]
Array for declarator or expression	$A({}^{*}L \bullet \eta) \bullet \alpha$	α [η]
Parenthesis Brackets	$B \bullet \alpha$	(α)
Cast expression	$C(\tau) \bullet \alpha$	(τ) α
decl-specifier prefix	$D \bullet \alpha$	extern α
exception-specification suffix	$E \bullet \alpha$	α throw ( )
Function declarator suffix	$E^? \bullet V^? \bullet F_d({}^{*}L \bullet \pi) \bullet \alpha$	α (π_1 , π_2) const throw ( )
Function call suffix	$F_e({}^{*}L \bullet \eta) \bullet \alpha$	α (η_1 , η_2)
Function call and declarator	$F_{de}({}^{*}L \bullet \rho) \bullet \alpha$	α (ρ_1 , ρ_2)
Function call or declarator	$E^? \bullet V^? \bullet F({}^{*}L \bullet \omega) \bullet \alpha$	α (ω_1 , ω_2) const throw ( )
Assigned initializer	$I(\eta) \bullet \alpha$	$\alpha = \eta$ α = `{ 5 }`
Constructed initializer	$J({}^{+}L \bullet \eta) \bullet \alpha$	α (η_1 , η_2)
(Non-pointer) Prefix operator	$O_p \bullet \alpha$	++ α
Infix operator[a]	$O_i \bullet \alpha$	α_1 / α_2 α = `{ 5 }`
Suffix operator	$O_s \bullet \alpha$	α ++

ptr-operator	$P_d(\zeta) \bullet \alpha$	`* volatile`
		`&`
		`Class::* const` α

Pointer for expression	$P_e(\zeta) \bullet \alpha$	* α & α
Unified Pointer operator	$P(\zeta) \bullet \alpha$	**cv-qualifier-seq*$_{opt}$ α & α $D^{+} \bullet \varepsilon$ `::`**cv-qualifier-seq*$_{opt}$ α
type-specifier prefix	$T \bullet \alpha$	`int` α
cv-qualifier-seq suffix	$V \bullet \alpha$	α const

Table 4.3 Grammatical Operators

a. The ={} form of *initializer* is incorporated directly into a generalised O_i to slightly reduce the number of distinct declarator / expression operators.

Overloading of lexical tokens and other simple lexical properties lead to the following identities

Notation	Description
$\sigma_d \subset \sigma_e \subset T\bullet\varepsilon \subset D\bullet\varepsilon$	Application type-names
$\beta \subset \sigma_e \subset T\bullet\varepsilon \subset D\bullet\varepsilon$	Built-in type-names
$\sigma_d \subset \theta_d$	Declarator type-names
$\theta_e \subset \theta_d$	Non-type-names
$V\bullet\varepsilon \subset T\bullet\varepsilon$	
$A_d\bullet\alpha \supset A_{de}\bullet\alpha \subset A_e\bullet\alpha$	
$A_d\bullet\alpha \subset A\bullet\alpha \supset A_e\bullet\alpha$	
$F_d\bullet\alpha \supset F_{de}\bullet\alpha \subset F_e\bullet\alpha$	
$E^?\bullet V^?\bullet F_d\bullet\alpha \subset E^?\bullet V^?\bullet F\bullet\alpha \supset F_e\bullet\alpha$	
$I\bullet\alpha \subset O_i\bullet\alpha$	
$P_d\bullet\alpha \supset P_{de}\bullet\alpha \subset P_e\bullet\alpha$	
$P_d\bullet\alpha \subset P\bullet\alpha \supset P_e\bullet\alpha$	
$B\bullet\tau = C(\tau)\bullet\varepsilon = F(\tau)\bullet\varepsilon$	Parenthesis / cast null / abstract function
$B\bullet{}^*L\bullet\pi = F_d({}^*L\bullet\pi)\bullet\varepsilon$	Parenthesis / abstract function
$B\bullet{}^*L\bullet\eta = F_e({}^*L\bullet\eta)\bullet\varepsilon$	Parenthesis / null-call
$B\bullet{}^*L\bullet\omega = F({}^*L\bullet\omega)\bullet\varepsilon$	Parenthesis / null-call or function
$\left\{\begin{matrix} O_p \\ P \\ D \\ T \end{matrix}\right\}\bullet\left\{\begin{matrix} O_s \\ A \\ F \\ E \\ V \end{matrix}\right\}\bullet\alpha = \left\{\begin{matrix} O_s \\ A \\ F \\ E \\ V \end{matrix}\right\}\bullet\left\{\begin{matrix} O_p \\ P \\ D \\ T \end{matrix}\right\}\bullet\alpha$	Prefix and suffix operators commute

Table 4.4 Properties

4.5.2 C++ Grammar Properties

This notation will now be applied to analyze the declaration and expression syntaxes. The analysis is a little lengthy and ignores a number of peripheral syntaxes that do not contribute to ambiguities. Thus `...`, which is unique to a *parameter-declaration-clause*, and `sizeof()`, which is unique to an expression are excluded from the analysis.

4.5.2.1 Names

Before we can analyse the grammar, we must first understand how names are used in the grammar. Figure 4.3 shows the coverage of the relevant productions in a graphical form. The diagram is a kind of Venn diagram in which different forms of name are arranged in four columns and thirteen rows. Shaded areas indicate the coverage of each production.

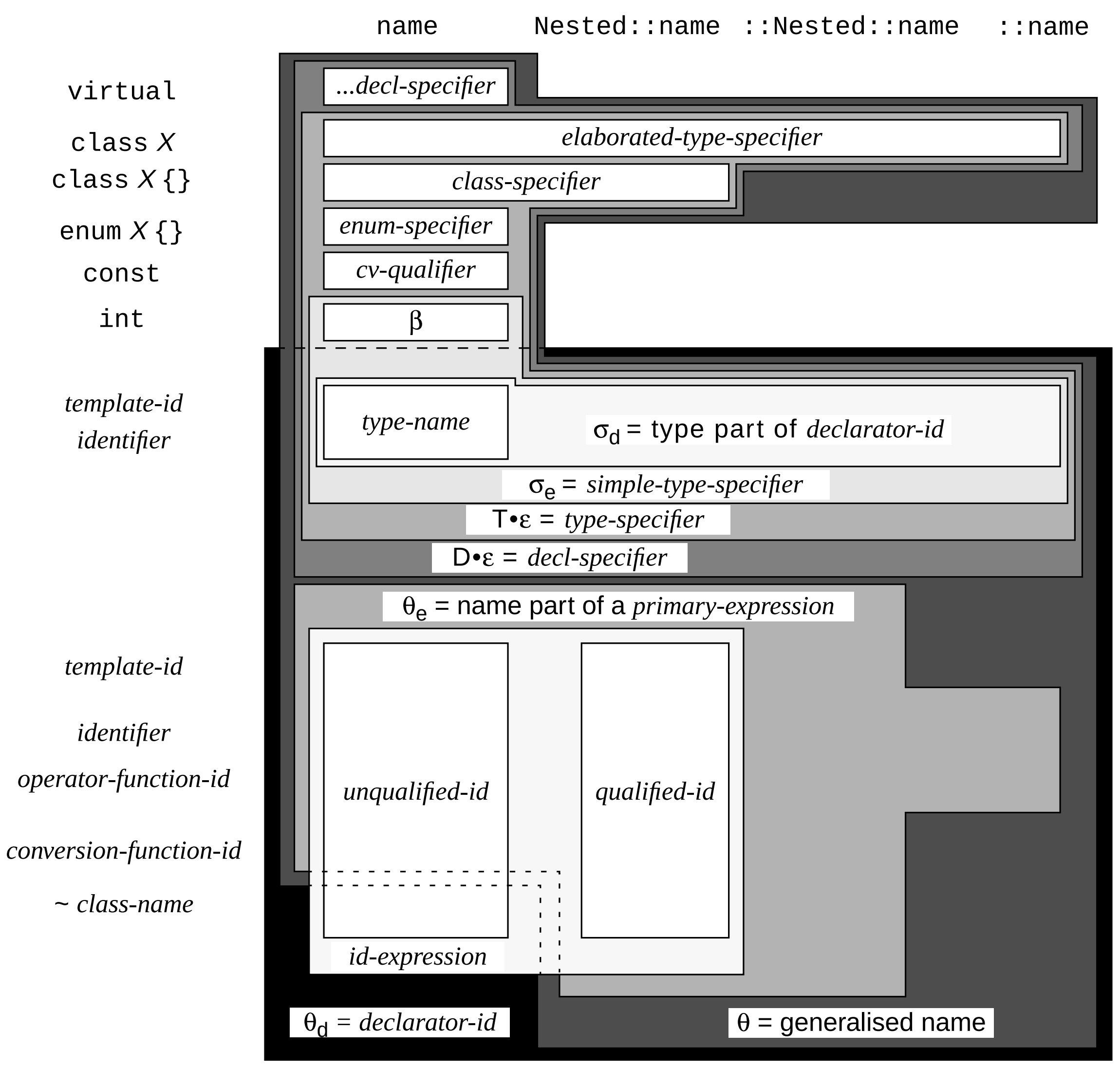

Figure 4.3 C++ Names

Four columns represent the possible scope nestings of a name:

- unscoped name (e.g. name)
- nested name (e.g. `Nested::name`)
- global scope nested name (e.g. `::Nested::name`)
- global name (e.g. `::name`)

Thirteen rows represent each of the different categories of name. The top 8 rows correspond to type-names:

- *...decl-specifier*, keywords such as `virtual`, `static` and `friend`
- *elaborated-type-specifier*, an enum or class reference
- *class-specifier*, a class definition
- *enum-specifier*, an enum definition
- *cv-qualifier*, `const` or `volatile`
- β, a single word built-in type such as `int`
- *template-id* (e.g. `FixedSizeArray < 4 >`)
- *identifier* (e.g. `MyType`)

The bottom five rows denote non-type names:

- *template-id* (e.g. `sort < int >`)
- *identifier* (e.g. `my_variable`)
- *operator-function-id* (e.g. `operator+=`)
- *conversion-function-id* (e.g. `operator MyClass *const *`)
- destructor name (e.g. `~MyClass`)

The *qualified-id* grammar production can be seen to cover all categories of nested non-type-name. A *declarator-id* covers all possible non-type-names and user-definable type-names.

A name is used in two distinct contexts.

A *declarator-id* forms the name in a declaration such as

```
int aDeclaratorId();
```

The name part of a *primary-expression* may be used in an expression such as

```
p = &aPrimaryExpressionName;
```

The irregular shape of the θ_e contributions to a *primary-expression* is the source of many difficulties in implementing the parser grammar. Reduce-reduce conflicts arise from a need to commit to a *declarator-id* or *primary-expression* before sufficient lookahead context has been examined. Part of this is just a trap for the unwary implementor. Since $\theta_e \subset \theta_d$, no conflict need arise. It is just the unhelpful way the grammar is written that is a problem.

Name differences between declarator and expression

A type-name is not generally valid in an expression. However, a specific variant of function call in the syntax for *postfix-expression*, supports use of a type-name as the function name, and serves to invoke a constructor or functional cast.

Omission of an unscoped destructor name resolves the ambiguity between the one's complement operator and a destructor for

```
~non_class_name & 7;
```

and gives the correct interpretation (§5.3.1-9) of

```
~ClassName();
```

Omission of a global *conversion-function-id* is semantically correct, but represents a needless syntactic complexity, since an ambiguity resolution of

something-ending-in-:: *conversion-function-id*

exploiting the syntactic exclusion requires something-ending-in-:: to be meaningful. The only construct ending in `::` is a *nested-name-specifier*, whose presence contradicts the presence of a global name.

Omission of a global destructor name is similarly semantically correct, but syntactically redundant.

The same argument applies to a global *template-id*, however in this case, it would appear that the omission is an error. Given:

```
template <class T> void sort(T *anArray, size_t arraySize);
```

Refusal to permit the hopefully redundant `::` in

```
p = &::sort<int>;
```

seems unreasonable.

Conclusion

The global name exclusions from a *primary-expression* can be ignored syntactically. A subsequent semantic check may yield a helpful diagnostic. Only the omission of the local destructor need be honoured.

Once the distinction between *type-name* and *identifier* is removed, the local destructor exclusion is the sole difference between a *declarator-id* and a name in a *primary-expression*. The generalised name coverage shown as θ is used for both purposes in Section 4.7. This covers some syntactically meaningless names in an expression, but misses out the local destructor from a *declarator-id*. The omission will be covered by a complement expression and must be repaired semantically.

4.5.2.2 Declarators, Declarations and Type Identifiers

The C++ grammar defines

declarator:
    direct-declarator
    ptr-operator declarator

direct-declarator:
    declarator-id
    direct-declarator (*parameter-declaration-clause*)
                        cv-qualifier-seq$_{opt}$ *exception-specification*$_{opt}$
    direct-declarator [*constant-expression*$_{opt}$]
    (*declarator*)

It is the two level recursion between these productions that makes them difficult to understand. Two levels are required because a CFG cannot express both prefix and suffix elaboration in the same production unambiguously. Considering:

affixed-production:
    `terminal`
    `prefix` *affixed-production*
    affixed-production `suffix`

It is unclear whether the prefix or suffix production rule is reduced first in:

```
prefix terminal suffix
```

In the extended regular expression notation outlined above, functional operators are used for lexical prefixes, suffixes and their combinations. The prefix-suffix ordering ambiguity is therefore removed and all forms of lexical decoration can be expressed uniformly. We may express the *declarator* syntax as

$$declarator = \left\{\begin{matrix} direct\text{-}declarator \\ P_d \cdot declarator \end{matrix}\right\}$$

$$direct\text{-}declarator = \left\{\begin{matrix} \theta_d \\ E^? \bullet V^? \bullet F_d \cdot direct\text{-}declarator \\ A_d \cdot direct\text{-}declarator \\ B \cdot declarator \end{matrix}\right\}$$

The recursion in the first choice can be simplified:

$$declarator = P_d^* \bullet direct\text{-}declarator$$

and then substituted in the second to give

$$\textit{direct-declarator} = \left\{\begin{matrix} E^? \bullet V^? \bullet F_d \\ A_d \\ B \bullet P_d^{\,*} \end{matrix}\right\}^* \bullet \theta_d$$

$$\textit{declarator} = P_d^{\,*} \bullet \left\{\begin{matrix} E^? \bullet V^? \bullet F_d \\ A_d \\ B \bullet P_d^{\,*} \end{matrix}\right\}^* \bullet \theta_d$$

Applying the same analysis to other parts of the C++ grammar we find that

$$\textit{abstract-declarator} = P_d^{\,*} \bullet \left\{\begin{matrix} \left\{\begin{matrix} E^? \bullet V^? \bullet F_d \\ A_d \\ B \bullet P_d^{\,*} \end{matrix}\right\}^* \bullet \left\{\begin{matrix} E^? \bullet V^? \bullet F_d \\ A_d \\ B \bullet P_d^{\,+} \end{matrix}\right\} \\ P_d \end{matrix}\right\} \bullet \varepsilon$$

$$\textit{type-id} = \tau = T^+ \bullet P_d^{\,*} \bullet \left\{ \left\{\begin{matrix} E^? \bullet V^? \bullet F_d \\ A_d \\ B \bullet P_d^{\,*} \end{matrix}\right\}^* \bullet \left\{\begin{matrix} E^? \bullet V^? \bullet F_d \\ A_d \\ B \bullet P_d^{\,+} \end{matrix}\right\} \right\}^? \bullet \varepsilon$$

$$\textit{new-type-id} = T^+ \bullet P_d^{\,*} \bullet A_d^{\,*} \bullet \varepsilon$$

$$\textit{init-declarator} = \left\{\begin{matrix} I \\ J \end{matrix}\right\}^? \bullet P_d^{\,*} \bullet \left\{\begin{matrix} E^? \bullet V^? \bullet F_d \\ A_d \\ B \bullet P_d^{\,*} \end{matrix}\right\}^* \bullet \theta_d$$

$$\textit{init-declarator-list} = {}^+L \bullet \left\{\begin{matrix} I \\ J \end{matrix}\right\}^? \bullet P_d^{\,*} \bullet \left\{\begin{matrix} E^? \bullet V^? \bullet F_d \\ A_d \\ B \bullet P_d^{\,*} \end{matrix}\right\}^* \bullet \theta_d$$

parameter-declaration = π where

$$\pi = D^+ \bullet I^? \bullet P_d^{\,*} \bullet \left\{\begin{matrix} \left\{\begin{matrix} E^? \bullet V^? \bullet F_d \\ A_d \\ B \bullet P_d^{\,*} \end{matrix}\right\}^* \bullet \left\{\begin{matrix} \theta_d \\ \left\{\begin{matrix} E^? \bullet V^? \bullet F_d \\ A_d \\ B \bullet P_d^{\,+} \end{matrix}\right\} \bullet \varepsilon \end{matrix}\right\} \\ \varepsilon \end{matrix}\right\}$$

(The usage of θ_d in a *parameter-declaration* is syntactically correct although semantic constraints allow only an *identifier*.)

Ignoring the optional ellipsis which is not a source ambiguity:

$$\textit{parameter-declaration-clause} = B \bullet {}^{*}L \bullet \pi$$

Generalising slightly by ignoring the constraint that a *decl-specifier-seq*$_{opt}$ applies only to the first element of an *init-declarator-list*:

$$\textit{simple-declaration} = \delta = D^{*} \bullet {}^{*}L \bullet \begin{Bmatrix} I \\ J \end{Bmatrix}^{?} \bullet P_d{}^{*} \bullet \begin{Bmatrix} E^{?} \bullet V^{?} \bullet F_d \\ A_d \\ B \bullet P_d{}^{*} \end{Bmatrix}^{*} \bullet \theta_d$$

4.5.2.3 Expressions

Analysis of an expression omitting uniquely prefixed terms such as new or const_cast that may be conveniently considered to be part of λ leads to

$$\textit{primary-expression} = \begin{Bmatrix} \theta_e \\ \lambda \\ B \bullet \textit{expression} \end{Bmatrix}$$

$$\textit{postfix-expression} = \begin{Bmatrix} \textit{primary-expression} \\ A_e \bullet \textit{postfix-expression} \\ F_e \bullet \textit{postfix-expression} \\ F_e \bullet \sigma_e \\ O_s \bullet \textit{postfix-expression} \end{Bmatrix} = \begin{Bmatrix} A_e \\ F_e \\ O_s \end{Bmatrix}^{*} \bullet \begin{Bmatrix} \theta_e \\ \lambda \\ F_e \bullet \sigma_e \\ B \bullet \textit{expression} \end{Bmatrix}$$

While analysing prefix operations, we lump the more mundane operators such as ++ under O_p, but keep the potentially ambiguous pointer operators * and & as P_e.

$$\textit{unary-expression} = \begin{Bmatrix} \textit{postfix-expression} \\ \begin{Bmatrix} O_p \\ P_e \end{Bmatrix} \bullet \textit{cast-expression} \end{Bmatrix}$$

$$\textit{cast-expression} = \begin{Bmatrix} \textit{unary-expression} \\ C \bullet \textit{cast-expression} \end{Bmatrix} = C^{*} \bullet \textit{unary-expression}$$

$$= C^{*} \bullet \begin{Bmatrix} \begin{Bmatrix} O_p \\ P_e \end{Bmatrix} \bullet C^{*} \end{Bmatrix}^{*} \bullet \textit{postfix-expression}$$

Simplifying the nested arbitrary choice, and substituting the *postfix-expression*

$$\textit{cast-expression} = \begin{Bmatrix} O_p \\ P_e \\ C \end{Bmatrix}^{*} \bullet \begin{Bmatrix} A_e \\ F_e \\ O_s \end{Bmatrix}^{*} \bullet \begin{Bmatrix} \theta_e \\ \lambda \\ F_e \bullet \sigma_e \\ B \bullet \textit{expression} \end{Bmatrix}$$

The ten levels of precedence for binary operators and assignment are not significant to this analysis. All binary and the tertiary infix operators are represented by O_i and we may write

$$\textit{assignment-expression} = \eta = O_i^* \bullet \left\{\begin{matrix} O_p \\ P_e \\ C \end{matrix}\right\}^* \bullet \left\{\begin{matrix} A_e \\ F_e \\ O_s \end{matrix}\right\}^* \bullet \left\{\begin{matrix} \theta_e \\ \lambda \\ F_e \bullet \sigma_e \\ B \bullet \textit{expression} \end{matrix}\right\}$$

$$\left\{\begin{matrix} \textit{expression} \\ \textit{expression-list} \end{matrix}\right\} = {}^+L \bullet O_i^* \bullet \left\{\begin{matrix} O_p \\ P_e \\ C \end{matrix}\right\}^* \bullet \left\{\begin{matrix} A_e \\ F_e \\ O_s \end{matrix}\right\}^* \bullet \left\{\begin{matrix} \theta_e \\ \lambda \\ F_e \bullet \sigma_e \\ B \bullet \textit{expression} \end{matrix}\right\}$$

$$= {}^+L \bullet O_i^* \bullet \left\{\begin{matrix} O_p \\ P_e \\ C \end{matrix}\right\}^* \bullet \left\{\begin{matrix} A_e \\ F_e \\ O_s \end{matrix}\right\}^* \bullet \left\{ B \bullet {}^+L \bullet O_i^* \bullet \left\{\begin{matrix} O_p \\ P_e \\ C \end{matrix}\right\}^* \bullet \left\{\begin{matrix} A_e \\ F_e \\ O_s \end{matrix}\right\}^* \right\}^* \bullet \left\{\begin{matrix} \theta_e \\ \lambda \\ F_e \bullet \sigma_e \end{matrix}\right\}$$

The doubly nested one or more choices permit arbitrary ordering

$$\left\{\begin{matrix} \textit{expression} \\ \textit{expression-list} \end{matrix}\right\} = {}^+L \bullet O_i^* \bullet \left\{\begin{matrix} O_p \\ P_e \\ C \end{matrix}\right\}^* \bullet \left\{\begin{matrix} A_e \\ F_e \\ O_s \end{matrix}\right\}^* \bullet \left\{\begin{matrix} A_e \\ F_e \\ O_s \\ O_p \\ O_i \\ P_e \\ C \\ B \bullet {}^+L \end{matrix}\right\}^* \bullet \left\{\begin{matrix} \theta_e \\ \lambda \\ F_e \bullet \sigma_e \end{matrix}\right\}$$

and the arbitrary ordering subsumes the multiplicity of prefixes

$$\left\{\begin{matrix} \textit{expression} \\ \textit{expression-list} \end{matrix}\right\} = {}^+L \bullet \left\{\begin{matrix} A_e \\ F_e \\ O_s \\ O_p \\ O_i \\ P_e \\ C \\ B \bullet {}^+L \end{matrix}\right\}^* \bullet \left\{\begin{matrix} \theta_e \\ \lambda \\ F_e \bullet \sigma_e \end{matrix}\right\}$$

4.5.3 C++ Ambiguities (using type information)

We have derived relatively simple formulae for the major declaration and expression constructs. The utility of this representation will be shown by deriving formulae that describe the conventional C++ ambiguities, when type information is available. This derivation can then be revisited to assess the consequences of parsing without type information.

The major ambiguities occur between and within declarations and expressions, since these syntaxes lack unique keywords. Ambiguities in statement syntax are isolated, since a leading `if` isolates the syntax from all other syntax (but not from the dangling `else` problem).

We concentrate on declarations and expressions since these are the sources of problems

- as many other implementors have already discovered
- as any attempt to implement a C++ grammar with yacc detects

4.5.3.1 Declaration / Declaration Ambiguity

A declaration / declaration ambiguity arises when

$$\textit{simple-declaration} = \delta = D^* \bullet {}^*L \bullet \begin{Bmatrix} I \\ J \end{Bmatrix}^? \bullet P_d{}^* \bullet \begin{Bmatrix} E^? \bullet V^? \bullet F_d \\ A_d \\ B \bullet P_d{}^* \end{Bmatrix}^* \bullet \theta_d$$

is open to more than one interpretation. Multiple interpretations are possible because there is lexical overlap between the different operators.

D^*•θ_d and D^*•P_d•α

A *declarator-id* (θ_d) or *ptr-operator* (P_d) may start with a `::` which may be ambiguous with respect to the last name in a *decl-specifier* (D).

```
Class ::Scope::p
Class ::Scope::* p
```

This ambiguity is not explicitly resolved in the standard, but a resolution may be inferred from the requirement to maximise the length of a *decl-specifier-seq* (§7.1-2). The resolution guarantees a semantic error.

A_d•θ_d or θ_d

An array declarator may add a `[]` suffix to a name.

The names `operator new[]` and `operator delete[]` end in `[]`, and the names `operator new` and `operator delete` exist. It is therefore unclear whether

```
int operator new[];
```

declares an array or a scalar. Since neither alternative is semantically valid, the syntactic problem is academic.

F_d•θ_d or θ_d

An array declarator may add a `()` suffix to a name.

The name `operator()` ends in `()`, but there is no name `operator` so there is no ambiguity.

$F_d({}^*L \bullet \pi) \bullet \theta_d$ or $D \bullet B \bullet {}^*L \bullet \pi$

The overloaded usage of parentheses leads to an ambiguity between

- $F_d({}^*L \bullet \pi) \bullet \theta_d$ - a constructor declaration
- $D \bullet B \bullet {}^*L \bullet \pi$ - a redundantly parenthesised variable declaration

for:

```
T(a)
```

Trimming δ to match these two forms

$$\delta_c = \begin{Bmatrix} I \\ J \end{Bmatrix}^? \bullet \left\{ \begin{matrix} E^? \bullet V^? \bullet F_d \\ A_d \end{matrix} \right\}^* \bullet F_d({}^*L \bullet \pi_c) \bullet \theta_d$$

$$\delta_v = D \bullet \begin{Bmatrix} I \\ J \end{Bmatrix}^? \bullet \left\{ \begin{matrix} E^? \bullet V^? \bullet F_d \\ A_d \end{matrix} \right\}^* \bullet B \bullet P_d^{\ *} \bullet \left\{ \begin{matrix} E^? \bullet V^? \bullet F_d \\ A_d \\ B \bullet P_d^{\ *} \end{matrix} \right\}^* \bullet \theta_d$$

The initializers, array and function suffixes can be dismissed using the semantic constraints on a constructor, leaving

$$\delta_c = F_d({}^*L \bullet \pi_c) \bullet \theta_d$$

$$\delta_v = D \bullet B \bullet P_d^{\ *} \bullet \left\{ \begin{matrix} E^? \bullet V^? \bullet F_d \\ A_d \\ B \bullet P_d^{\ *} \end{matrix} \right\}^* \bullet \theta_d$$

The ambiguity arises for $\delta_c \cap \delta_v$, which is non-trivial when the name preceding the parentheses is $\sigma_d = \theta_d \cap (D \bullet \varepsilon)$. The inside of the parenthesis is ambiguous when the parameter

$$\pi = D^+ \bullet I^? \bullet P_d^{\ *} \bullet \left\{ \begin{matrix} \left\{ \begin{matrix} E^? \bullet V^? \bullet F_d \\ A_d \\ B \bullet P_d^{\ *} \end{matrix} \right\}^* \bullet \left\{ \begin{matrix} \theta_d \\ \left\{ \begin{matrix} E^? \bullet V^? \bullet F_d \\ A_d \\ B \bullet P_d^{\ +} \end{matrix} \right\} \bullet \varepsilon \end{matrix} \right\} \\ \varepsilon \end{matrix} \right\}$$

is ambiguous with respect to

$$P_d^{\ *} \bullet \left\{ \begin{matrix} E^? \bullet V^? \bullet F_d \\ A_d \\ B \bullet P_d^{\ *} \end{matrix} \right\}^* \bullet \theta_d$$

Trimming impossible terms gives the two constraints

$$\pi_c = \left\{\begin{array}{c} D^+ \bullet \left\{\begin{array}{c} E^? \bullet V^? \bullet F_d \\ A_d \\ B \bullet P_d{}^* \end{array}\right\}^* \bullet \left\{\begin{array}{c} \theta_d \\ \left\{\begin{array}{c} E^? \bullet V^? \bullet F_d \\ A_d \\ B \bullet P_d{}^+ \end{array}\right\} \bullet \varepsilon \end{array}\right\} \\ \\ D^+ \bullet \varepsilon \end{array}\right\}$$

$$\pi_v = \left\{\begin{array}{c} E^? \bullet V^? \bullet F_d \\ A_d \\ B \bullet P_d{}^* \end{array}\right\}^* \bullet \theta_d$$

$\sigma_d = \theta_d \cap (D^+ \bullet \varepsilon)$ is a simple solution to $\pi_{cv} = \pi_c \cap \pi_v$.

A more complicated solution arises through the recursive ambiguity between precedence and function argument parentheses.

$$\left\{\begin{array}{c} E^? \bullet V^? \bullet F_d \\ A_d \end{array}\right\}^* \bullet F_d\left(\left\{\begin{array}{c} E^? \bullet V^? \bullet F_d \\ A_d \end{array}\right\}^* \bullet \sigma_d\right) \bullet \sigma_d$$

leading to the complete form of the constructor / parenthesised variable ambiguity

$$\delta_{cv} = F_d\left(\left\{\left\{\begin{array}{c} E^? \bullet V^? \bullet F_d \\ A_d \end{array}\right\}^* \bullet F_d\left(\left\{\begin{array}{c} E^? \bullet V^? \bullet F_d \\ A_d \end{array}\right\}^* \bullet \sigma_d\right)\right\}^? \bullet \sigma_d\right) \bullet \sigma_d$$

$J({}^*L\bullet\eta)\bullet\theta_d$ or $D\bullet B\bullet{}^*L\bullet\pi$

Parenthesis overloading would also appear to lead to an ambiguity between construction of an object and a redundantly parenthesised variable. However, construction of an object requires an explicit type and so the object construction must at least be of the form $D\bullet J({}^*L\bullet\eta)\bullet\theta_d$. The ambiguity of this with respect to a function declaration is considered next.

$D^*\bullet J({}^*L\bullet\eta)\bullet\theta_d$ or $D^*\bullet F_d({}^*L\bullet\pi)\bullet\theta_d$

There is a lexical ambiguity between

- $D^*\bullet J({}^*L\bullet\eta)\bullet\theta_d$ - a constructed object declaration
- $D^*\bullet F_d({}^*L\bullet\pi)\bullet\theta_d$ - a function declaration

for:

```
T a(b), c(d), e(f);
```

Semantic constraints permit and require ambiguous *decl-specifier-seq* prefixes for the first element of an *init-declarator-list*, but require no *decl-specifier-seq* for subsequent elements. The D and L terms are therefore eliminated to give the following forms for each alternative *simple-declaration*.

$$\delta_o = J({}^{*}L \bullet \eta) \bullet P_d{}^{*} \bullet \begin{Bmatrix} E^{?} \bullet V^{?} \bullet F_d \\ A_d \\ B \bullet P_d{}^{*} \end{Bmatrix}^{*} \bullet \theta_d$$

$$\delta_f = P_d{}^{*} \bullet F_d({}^{*}L \bullet \pi) \bullet \begin{Bmatrix} E^{?} \bullet V^{?} \bullet F_d \\ A_d \\ B \bullet P_d{}^{*} \end{Bmatrix}^{*} \bullet \theta_d$$

These two terms are a direct match syntactically, subject to the recursive ambiguity to satisfy the parameter lists. This is $F_{de}({}^{*}L \bullet \rho)$ and is analysed in the next section. The ambiguity is therefore:

$$\delta_{of} = P_d{}^{*} \bullet F_{de}({}^{*}L \bullet \rho) \bullet \begin{Bmatrix} E^{?} \bullet V^{?} \bullet F_d \\ A_d \\ B \bullet P_d{}^{*} \end{Bmatrix}^{*} \bullet \theta_d$$

4.5.3.2 Declaration / Expression ambiguity

The declaration / expression ambiguity arises when a

$$\textit{simple-declaration} = \delta = D^{*} \bullet {}^{*}L \bullet \begin{Bmatrix} I \\ J \end{Bmatrix}^{?} \bullet P_d{}^{*} \bullet \begin{Bmatrix} E^{?} \bullet V^{?} \bullet F_d \\ A_d \\ B \bullet P_d{}^{*} \end{Bmatrix}^{*} \bullet \theta_d$$

followed by a semi-colon, forming part of a *statement*, is ambiguous with respect to

$$\textit{expression} = {}^{+}L \bullet \eta = {}^{+}L \bullet \begin{Bmatrix} A_e \\ F_e \\ O_s \\ O_p \\ O_i \\ P_e \\ C \\ B \bullet {}^{+}L \end{Bmatrix}^{*} \bullet \begin{Bmatrix} \theta_e \\ \lambda \\ F_e \bullet \sigma_e \end{Bmatrix}$$

which when followed by a semi-colon is an *expression-statement*.

The expression / declaration ambiguity may be identified by comparing terms to give

$$\delta \cap {}^{+}L \bullet \eta = D^{0} \bullet {}^{+}L \bullet \begin{Bmatrix} I \\ J \end{Bmatrix}^{?} \bullet P_e{}^{*} \bullet \begin{Bmatrix} F_{de} \\ A_{de} \\ B \bullet {}^{1}L \bullet P_e{}^{*} \end{Bmatrix}^{*} \bullet \begin{Bmatrix} \theta_e \\ F_{de} \bullet \sigma_e \end{Bmatrix}$$

D^0 matches the absence of D in an expression.

^+L enforces the expression requirement for at least one element.

Commutativity of prefix and suffix operators permits the intervening pointer to be traversed by the initializers so that: I resolves the ambiguity between I and O_i since $I \subset O_i$. J is covered by an F_e in an expression.

P_e unifies P_e and P_d acquiring the P_e from the expression multiplier term.

F_{de} is the recursive ambiguity between F_d and F_e analysed below.

A_{de} resolves the ambiguity between A_d and A_e restricting the array argument to exactly one *constant-expression*.

^1L is the identity operator necessary to cover $B \bullet P_d^*$ by successive $B \bullet {}^+L$ and P_e terms from the expression multiplier term.

θ_e resolves the ambiguity between θ_d and θ_e since $\theta_e \subset \theta_d$.

The $F_{de} \bullet \sigma_e$ terms arises because $\sigma_e \subset \theta_d$.

A quick test of this formula makes the prediction that

```
    i = 0;
```

should be ambiguous, although such a simple ambiguity is not mentioned elsewhere, and resolution as a declaration would reject most C++ programs. The reason is that the analysis above is purely syntactic. In the production

simple-declaration: *decl-specifier-seq*$_{opt}$ *init-declarator-list*$_{opt}$;

the strict syntactic interpretation permits the *decl-specifier-seq* (the type) to be omitted. `i = 0` is a valid form of *init-declarator-list*. An untyped name alone is therefore syntactically ambiguous. Since §6.8-3 prohibits the use of more than semantic type information to disambiguate, there appears to be a problem. The problem disappears if the constraint in §7-7 is interpreted as a syntactic rather than semantic constraint. The constraint specifies that a *decl-specifier-seq* may only be omitted for function-like declarations. Incorporating this constraint, together with constraints that implicit `int` functions are not allowed, and functions are not constructed, gives

$$\delta \cap {}^+L \bullet \eta = {}^+L \bullet I^? \bullet \begin{Bmatrix} F_{de} \\ A_{de} \end{Bmatrix}^* \bullet F_{de} \bullet \begin{Bmatrix} \theta_e \\ \sigma_e \end{Bmatrix}$$

The $I^?$ covers only the `= 0` of a *pure-specifier* and can be eliminated since there is no D prefix to supply a `virtual` keyword.

θ_e covers only destructor names and *conversion-function-id*s, which cannot be declared where expression statements are valid.

Functions returning arrays or functions are invalid.

Application of semantic constraints therefore reduces the ambiguity to the more familiar:

$$\delta \cap {}^+L \bullet \eta = {}^+L \bullet F_{de} \bullet \begin{Bmatrix} \theta_e \\ \sigma_e \end{Bmatrix}$$

Continuing the analysis to determine the recursive ambiguity. The F_{de} ambiguity arises when a *parameter-declaration-clause* is ambiguous with respect to a parenthesised *expression-list*. This occurs when each *parameter-declaration*

$$\pi = D^+ \bullet I^? \bullet P_d{}^* \bullet \left\{\begin{bmatrix} \left\{\begin{bmatrix} E^? \bullet V^? \bullet F_d \\ A_d \\ B \bullet P_d{}^* \end{bmatrix}\right\}^* \bullet \left\{\begin{bmatrix} \theta_d \\ \left\{\begin{bmatrix} E^? \bullet V^? \bullet F_d \\ A_d \\ B \bullet P_d{}^+ \end{bmatrix}\right\} \bullet \varepsilon \end{bmatrix}\right\} \\ \varepsilon \end{bmatrix}\right\}$$

is ambiguous with respect to the corresponding

$$\textit{assignment-expression} \;=\; \eta = \left\{\begin{bmatrix} A_e \\ F_e \\ O_s \\ O_p \\ O_i \\ P_e \\ C \\ B \bullet {}^+L \end{bmatrix}\right\}^* \bullet \left\{\begin{bmatrix} \theta_e \\ \lambda \\ F_e \bullet \sigma_e \end{bmatrix}\right\}$$

The presence of D^+ in the *parameter-declaration* would appear to preclude an ambiguity, however $\sigma_e \subset D \bullet \varepsilon$. Trimming completely unsuitable terms and arranging to exploit the lexical commutation $D \bullet F \bullet \varepsilon = F \bullet D \bullet \varepsilon$ gives the solution as

$$\rho \;=\; \pi \cap \eta \;=\; D \bullet I^? \bullet \left\{\begin{bmatrix} F_d \\ A_d \end{bmatrix}\right\}^* \bullet F_d \bullet \varepsilon \cap \left\{\begin{bmatrix} A_e \\ F_e \\ O_i \end{bmatrix}\right\}^* \bullet F_e \bullet \sigma_e$$

This simplifies to give

$$\rho \;=\; \pi \cap \eta \;=\; I^? \bullet \left\{\begin{bmatrix} F_{de} \\ A_{de} \end{bmatrix}\right\}^* \bullet F_{de} \bullet \sigma_e$$

as the form of an argument of F_{de} that is ambiguously either a *parameter-declaration* or an *expression*. This ambiguity is not only recursive, but also exhibits multiplicity in its recursion.

Performing a sanity check: the simplest form is

```
    TypeName()
```

which is recognisable as an *expression* involving construction of a `TypeName`. That it is also a *parameter-declaration* requires understanding of a very dark corner of C++ (§8.3.5-3). The declaration interpretation is of an abstract (unnamed) function taking no arguments and returning `TypeName`. This has no meaning in C++, since functions are not first class entities. Function names are interpreted as pointers to functions and so the example is equivalent to:

```
    TypeName (*)()
```

4.5.3.3 Expression / Expression ambiguities

The foregoing analysis has taken little account of ambiguity between operators. Operators are excessively overloaded: parentheses variously denote a cast, a function-call or arithmetic grouping. We therefore analyse an assignment expression in terms of its lexical layout eliminating the application operator and using only lexical adjacency. For this purpose we introduce $\tilde{}$ so that $\tilde{\alpha}$ denotes a comma separated list of zero or more elements of α. Punctuation such as (,), [, and] represent the lexical character.

The production rule for an assignment expression is:

$$\eta \rightarrow \left\{ \begin{array}{cc} \eta\ [\ \eta\ ] & \textit{array} \\ \eta \left(\tilde{\eta} \right) & \textit{call} \\ \eta\ O_s & \textit{suffix} \\ O_p\ \eta & \textit{prefix} \\ (\ \tau\ )\ \eta & \textit{cast} \\ \eta\ O_i\ \eta & \textit{infix} \\ P\ \eta & \textit{pointer} \\ \left(\tilde{\eta} \right) & \textit{parenthesis} \\ \theta_e & \textit{name} \\ \lambda & \textit{value} \\ \sigma_e \left(\tilde{\eta} \right) & \textit{functional-cast} \end{array} \right]$$

The [η] of an array is nearly unique and so creates no expression ambiguities. The sole other use occurs in `operator new []` and `operator delete []` where the absence of η disambiguates.

The `()` of calls, casts, parentheses and function-casts may create

- parenthesised-call / cast-parenthesis
- parenthesised-binary / cast-unary
- call / functional-cast

ambiguities.

The parenthesised-call / cast-parenthesis ambiguity is

$$\left(\tilde{\eta}_1 \right) \left(\tilde{\eta}_2 \right) \cap (\tau)\ (\eta_3)$$

and the parenthesised-binary / cast-unary is

$$\left(\tilde{\eta}_1 \right) O_i\ \eta_2 \cap (\tau) \left\{ \begin{array}{c} O_p \\ P_e \end{array} \right\} \eta_3$$

There are ambiguities when $\eta_2 = \eta_3$ and $\eta_1^{\sim} = \tau$ and O_i is $+$, $-$, $*$ or &. Further ambiguities exist when O_i is ++ or `--` and η_2 and η_3 are parenthesised.

The ambiguities arising when $\eta = \tau$ are analysed in Section 4.5.3.4 as the *type-id* / *assignment-expression* ambiguity.

The call / functional-cast ambiguity analysed in Section 4.5.3.5 does not arise when type-name information is available.

Some suffix operators such as ++ are also prefix operators. No ambiguity can arise because suffix operators cannot precede a prefix operator.

Some infix operators such as + are also prefix operators, while others such as * are also pointer operators. No ambiguity arises since, in an expression such as

```
    a * * * b
```

the absence of a further suffix ambiguity ensures that the first * must be infix and subsequent *'s prefix. However, a *conversion-function-id* is covered by θ_e and may end in a * creating this further suffix ambiguity.

```
    &Class::operator int* * *pointer
```

This is resolved by language definition (§12.3.2-4) to maximise the length of the *conversion-function-id*: a resolution that can never avoid a subsequent semantic error. An equivalent ambiguity within a *new-expression* is similarly resolved (§5.3.4-2).

θ_e and λ are independent and cause no ambiguity, beyond those already discussed.

4.5.3.4 *type-id* / *expression-list* ambiguity

A parenthesised-call / cast-parenthesis or parenthesised-binary / cast-unary ambiguity exists when the

$$\textit{type-id} = \tau = T^{+} \bullet P_d^{*} \bullet \left\{ \begin{bmatrix} E^{?} \bullet V^{?} \bullet F_d \\ A_d \\ B \bullet P_d^{*} \end{bmatrix}^{*} \bullet \begin{Bmatrix} E^{?} \bullet V^{?} \bullet F_d \\ A_d \\ B \bullet P_d^{+} \end{Bmatrix} \right\}^{?} \bullet \varepsilon$$

of the cast may be confused with the

$$\begin{Bmatrix} \textit{expression} \\ \textit{expression-list} \end{Bmatrix} = {}^{+}L \bullet \begin{Bmatrix} A_e \\ F_e \\ O_s \\ O_p \\ O_i \\ P_e \\ C \\ B \bullet {}^{+}L \end{Bmatrix}^{*} \bullet \begin{Bmatrix} \theta_e \\ \lambda \\ F_e \bullet \sigma_e \end{Bmatrix}$$

of the parenthesised call as (a) in

```
    (a)(b)
    (a)-b
    (a)++(b)
```

The T^{+} term can only match the σ_e, since $\sigma_e \subset T \bullet \varepsilon$, leading to the solution

$$\tau \cap ({}^{+}L \bullet \eta) = \begin{Bmatrix} A_{de} \\ F_{de} \end{Bmatrix}^{*} \bullet F_{de} \bullet \sigma_{e}$$

which again makes use of the strange equivalence of a function and pointer to function to determine that type information alone is insufficient to disambiguate

```
    (Class())(x)
```

which could be

```
    Class().operator()(x)
```

or

```
    (Class (*)())(x)
```

A cast to function is not one of the recognised forms of cast enumerated in §5.4, and so the ambiguity has a well-defined semantic resolution, which may be used to avoid the syntactic ambiguity.

4.5.3.5 call / functional-cast ambiguity

The call / functional-cast ambiguity arises when the type-name σ_e in a functional-call can be mistaken for an *assignment-expression*. A type-name is not valid as an *assignment-expression* so there is no call / functional-cast ambiguity when type-name information is available.

4.6 Parsing the ambiguities

We have used an extended regular expression notation to derive the declaration and expression ambiguities in C++ syntax. We now examine how ambiguities may be resolved in a practical parser.

If the BNF provided as part of the C++ standard is converted directly into a yacc grammar, yacc reports many ambiguities, mostly relating to declarations and expressions. The expression / declaration ambiguities are by far the most serious and much the hardest to resolve.

We now consider the traditional approach to resolving an ambiguity, a multi-pass approach formerly adopted by FOG, and a more efficient superset grammar approach with similar characteristics but whose implementation is not yet complete.

4.6.1 Parsing against an ambiguity (the traditional approach)

An ambiguity may be resolved within the grammar, by rewriting the grammar to remove the ambiguity, or by providing assistance in the form of disambiguation tokens from a lookahead parser. The lookahead approach is straightforward, but just redefines the problem as one to be solved elsewhere, potentially using ad hoc code that may be prone to incorrect programming assumptions in regard to the complexity of a recursive ambiguity.

Removing the ambiguities within the grammar is very hard. Given two mutually ambiguous subgrammars A and B, it is easy to see that the ambiguity is removed by identifying the ambiguity AandB comprising all sentences that could form part of A and could form part of B. The ambiguous case may then be removed from each of the original subgrammars to leave OnlyA and OnlyB. The revised grammars comprising OnlyA, AandB and OnlyB are free from the A B ambiguity.

$$\begin{Bmatrix} A \\ B \end{Bmatrix} = \begin{Bmatrix} \begin{Bmatrix} OnlyA \\ AandB \end{Bmatrix} \\ \begin{Bmatrix} AandB \\ OnlyB \end{Bmatrix} \end{Bmatrix} = \begin{Bmatrix} OnlyA \\ AandB \\ OnlyB \end{Bmatrix} = \begin{Bmatrix} A \\ OnlyB \end{Bmatrix}$$

The analyses in the previous section show how AandB can be identified from A and B, and show that in the case of the expression / declaration ambiguity, the ambiguity has to be expressed recursively. Converting AandB back into BNF rules is relatively straightforward. However OnlyA and OnlyB require a subtraction of formulae and consequently result in very complicated expressions for OnlyA and OnlyB. Determining these in the non-recursive context is somewhat daunting. An accurate recursive resolution of expression-that-is-not-a-declaration is a major undertaking. For the declaration / expression case, the other subtraction leading to declaration-that-is-not-an-expression does not need to be evaluated, because the disambiguation rule mandates that the ambiguity be resolved as a declaration. It is sufficient to parse the declaration unchanged as A and the expression-that-is-not-a-declaration as OnlyB.

Once unambiguous formulae have been identified, they then need to be converted to BNF in a way that does not require more than one token of lookahead. It is not sufficient to convert the formulae independently. They must be converted together so that no shift-reduce or reduce-reduce conflicts are introduced when the parser has seen a partial input that could prefix more than one alternative.

In practice, the use of type information resolves nearly all ambiguities, so that the shared prefix constraint is only a hard problem.

The relatively arbitrary nature of the gcc, CPPP and Roskind grammars suggest that an approximate solution to the above problems was discovered empirically. The problem is too complex to be amenable to an empirical approach. Both the gcc and CPPP grammars are reported to fail to correctly resolve more complicated declaration / expression ambiguities.

In summary, parsing against an ambiguity requires an accurate implementation of the grammar. This is hard to achieve for a deep and recursive ambiguity. [Roskind91] describes his solution as “A LOT of work” and notes that some ambiguities are resolved prematurely. The gcc implementation was also not easily reached.

4.6.2 Parsing without an ambiguity (the multi-pass approach)

Expressions and declarations are disambiguated by preferring a declaration to an expression whenever there is an ambiguity (§6.8). A two-pass parser can therefore be designed that first parses for a declaration, and if that parse fails, then parses for an expression.

$$\begin{Bmatrix} A \\ B \end{Bmatrix} = A \text{ or else } B$$

This makes for a much simpler grammar implementation since no grammar revision is required to remove the ambiguity. We just need to support the ability to perform multiple passes.

Back-tracking in the context of a parser involves examining the input token stream to see whether the stream satisfies a candidate syntax, and if not backing up again to try another candidate.

This practice is common in hand-written parsers, which normally use a top-down left-most reduction at the left policy (LL). Examining the left of a production tends to make premature decisions that then need to be undone.

Use of derivation at the right in an LR parser avoids premature decisions and can make back-tracking unnecessary. Generally, back-tracking is undesirable, since work performed upon each backed-up path is wasted. Well-structured grammar code does not need to back-track.

The standard parser tools do not support back-tracking, and although the author is not aware of any other implementation that back-tracks, tools such as yacc are in such widespread use that it is unlikely that the approach is totally novel. The implementation of back-tracking is presented in Section 4.8, following the description of parsing approaches.

Cost

Accurate determination of the cost of back-tracking requires instrumentation of a parser that adopts both approaches. Such a parser has not been developed, so we can only estimate the likely costs.

Back-tracking incurs three costs:

- marking and unmarking a restart position (always)
- restarting at the mark (only when a back-track necessary)
- wasted analysis effort (only when a back-track necessary)

Maintenance of the marked position need not be particularly costly, if each token is already represented by a polymorphic object, but may be more noticeable if the tokens are acquired directly as a binary stream from a preprocessor.

FOG maintains a garbage collector context at each mark, so re-establishing the mark not only back-tracks on the input context, but also destroys any unwanted objects created to support the failed analysis. This cost will be low, since a failed analysis will normally fail before creating many objects.

The wasted analysis effort will also be small, since an illegal syntax will usually fail after only a few tokens.

Back-tracking incurs no costs for syntax such as *selection-statement*s that can be identified from their first token. Costs remain small for syntax that corresponds to the first analysis alternative. The costs only become significant for *expression-statement*s that closely resemble *declaration-statement*s and so cause significant wasted effort.

The parsing cost is estimated to increase by 20%, which is undesirable for a production compiler, but justifiable in terms of the improved modularity for a research tool.

The approach recommended in the next section reduces this 20% estimate to a negligible level.

4.6.3 Parsing with the ambiguity (the superset approach)

A design philosophy of C is the principle that declarations imitate the style of their usage. The declaration syntax is therefore deliberately rather than accidentally and inconveniently similar to the expression syntax. This property can be exploited to develop a superset syntax that encompasses both declarations and expressions.

$$\left\{\begin{matrix} A \\ B \end{matrix}\right\} \subseteq AorB$$

AorB covers all sentences that satisfy A and all sentences that satisfy B. Deriving AorB involves adding the relevant formulae and pruning any duplication. In order to simplify the grammar we may choose to add further terms to AorB provided we do not introduce any new ambiguities, or at least provided we can resolve any new ambiguities that we do introduce. We require the grammar for AorB to provide cover for at least A and at least B. We do not require precise equivalence.

The consequence of choosing to make AorB larger is that some sentences that were formerly syntax errors, are now accepted by the superset grammar. These sentences should be diagnosed in a later semantic analysis. This is actually beneficial, since the extra sentences that are accepted have a close similarity to legal sentences and so cover likely programming errors. Accepting such errors syntactically improves the likelihood that a diagnostic will report an error that is relevant to the programmers intent. For instance given

```
    typedef type a virtual;
```

many compilers may succeed in diagnosing an illegal typedef. The greater syntactic coverage may allow the compiler to accept the typedef initially but report that `virtual` is not a legal qualifier for the name of a typedef.

With such a superset grammar there is then no ambiguity, merely a loss of resolution. The loss of resolution can be recovered by semantic processing following the superset syntactic parse.

4.7 The Superset Grammar Approach

The superset grammar approach described in this section comprises two innovations, each of which could in principle be used independently.

Unification of declaration and expression syntaxes provides a solution to the major C++ parsing problem: the declaration / expression ambiguity. The problem ceases to be syntactic. It is deferred to the semantic level where it belongs and is relatively easily resolved.

Elimination of the use of type information avoids the need for potentially infinite lookahead to perform type disambiguation, since incorporation of type requires substantial grammar elaboration to handle the undecided lookahead. Removal of type simplifies the grammar, allowing type related ambiguities to be removed from the grammar and deferred for semantic resolution, where they too belong.

We first revisit the ambiguity analysis of Section 4.5.3 to see what problems a lack of type information causes. We then present relevant parts of the superset grammar to show how the superset is implemented. The full superset C++ grammar may be found in Appendix B, and the extended FOG superset in Appendix C.

4.7.1 C++ Ambiguities (without type information)

With type information, σ_e is a type-name and is distinct from θ_e which is a non-type-name. Without type information there is no distinction, and we just use the superset name θ. This was shown graphically in Figure 4.3 on page 127.

$$\theta \supset D \bullet \varepsilon \cup \theta_e$$

The $F_e \bullet \sigma_e$ term in expressions may be subsumed by the replicator.

$$\begin{Bmatrix} \textit{expression} \\ \textit{expression-list} \end{Bmatrix} = {}^{+}L \bullet \begin{Bmatrix} A_e \\ F_e \\ O_s \\ O_p \\ O_i \\ P_e \\ C \\ B \bullet {}^{+}L \end{Bmatrix}^{*} \bullet \begin{Bmatrix} \theta \\ \lambda \end{Bmatrix}$$

4.7.1.1 Declaration / Expression ambiguity

The strict ambiguity for a declaration / expression ambiguity (from Section 4.5.3.2) simplifies initially to

$$\delta \cap {}^{+}L \bullet \eta = D^0 \bullet {}^{+}L \bullet \begin{Bmatrix} I \\ J \end{Bmatrix}^{?} \bullet P_e^{*} \bullet \begin{Bmatrix} F_{de} \\ A_{de} \\ B \bullet {}^{1}L \bullet P_e^{*} \end{Bmatrix}^{*} \bullet \theta$$

and after application of semantic constraints to:

$$\delta \cap {}^{+}L \bullet \eta = {}^{+}L \bullet F_{de} \bullet \theta$$

with the corresponding recursive parameter ambiguity ultimately simplifying to

$$\pi \cap \eta = I^{?} \bullet \begin{Bmatrix} F_{de} \\ A_{de} \end{Bmatrix}^{*} \bullet F_{de} \bullet \theta$$

These are greater ambiguities but they have the same structure as before. The former ambiguities involving just σ_e now involve θ. This loss of precision can be recovered as soon as type information is available. Type information is not necessary to identify the syntactical structure in which θ is used. The ambiguity may therefore be safely deferred for semantic rather than syntactic resolution.

4.7.1.2 Expression / Expression ambiguity

Removal of the distinction between σ_e and θ_e requires reassessment of ambiguities related to σ_e and τ.

The former declaration / declaration ambiguity involving `operator new[]` now becomes an expression / expression ambiguity as well.

4.7.1.3 *type-id* / *expression-list* ambiguity

A parenthesised-call / cast-parenthesis or parenthesised-binary / cast-unary ambiguity exists when the

$$\textit{type-id} = \tau = T^{+} \bullet P_d^{*} \bullet \left\{ \begin{Bmatrix} E^{?} \bullet V^{?} \bullet F_d \\ A_d \\ B \bullet P_d^{*} \end{Bmatrix}^{*} \bullet \begin{Bmatrix} E^{?} \bullet V^{?} \bullet F_d \\ A_d \\ B \bullet P_d^{+} \end{Bmatrix} \right\}^{?} \bullet \varepsilon$$

of the cast may be confused with

$$assignment\text{-}expression\ =\ \eta = \left\{\begin{matrix} A_e \\ F_e \\ O_s \\ O_p \\ O_i \\ P_e \\ C \\ B \bullet {}^{+}L \end{matrix}\right\}^{*} \bullet \left\{\begin{matrix} \theta \\ \lambda \end{matrix}\right\}$$

The T$^+$ term can now match part of θ, since without type information $T \bullet \varepsilon \cap \theta \neq \varepsilon$, leading to the solution (for the potential type names)

$$\tau \cap ({}^{+}L \bullet \eta)\ =\ \left\{\begin{matrix} A_{de} \\ F_{de} \end{matrix}\right\}^{*} \bullet (\theta \cap T \bullet \varepsilon)$$

indicating that

```
(T)+5
```

is now ambiguous: is it a cast of +5 to type T, or the sum of T and 5?

This decision cannot be made without type information. However it can be deferred until type information is available, since an AST node that misleadingly describes the addition of a type to a value can be detected and corrected to describe the corresponding cast.

4.7.1.4 Call / functional-cast ambiguity

The call / functional-cast ambiguity arises when the type-name σ_e in a functional-call can be mistaken for an *assignment-expression*. A type-name is not distinct from θ which is a valid *assignment-expression*. All forms of functional-cast become ambiguous. The functional-cast is excised from the grammar, with detection of functional-casts deferred until type information is available to determine whether the function-name associated with a call is a type-name or not.

4.7.2 A naive Assignment-Expression / Parameter-Declaration superset

assignment-expression and *parameter-declaration* occur as part of the recursive ambiguity between an *expression-list* as a function call argument list and the *parameter-declaration-clause* of a *function-definition*. We must find a superset that covers both

$$\pi = D^{+} \bullet I^{?} \bullet P_d^{*} \bullet \left\{\begin{matrix} \left\{\begin{matrix} E^{?} \bullet V^{?} \bullet F_d \\ A_d \\ B \bullet P_d^{*} \end{matrix}\right\}^{*} \bullet \left\{\begin{matrix} \theta_d \\ \left\{\begin{matrix} E^{?} \bullet V^{?} \bullet F_d \\ A_d \\ B \bullet P_d^{+} \end{matrix}\right\} \bullet \varepsilon \end{matrix}\right\} \\ \varepsilon \end{matrix}\right\}$$

and

$$\eta = \left\{ \begin{bmatrix} A_e \\ F_e \\ O_s \\ O_p \\ O_i \\ P_e \\ C \\ B \bullet {}^{+}L \end{bmatrix}^{*} \right\} \bullet \left\{ \begin{bmatrix} \theta \\ \lambda \end{bmatrix} \right\}.$$

A very naive common superset for $\omega \supseteq \left\{ \begin{bmatrix} \pi \\ \eta \end{bmatrix} \right\}$ is

$$D^{*} \bullet \left\{ \begin{bmatrix} A \\ E^{?} \bullet V^{?} \bullet F \\ O_s \\ O_p \\ O_i \\ P \\ E^{?} \bullet V^{?} \bullet C \\ E^{?} \bullet V^{?} \bullet B \bullet {}^{+}L \end{bmatrix}^{*} \right\} \bullet \left\{ \begin{bmatrix} \theta \\ \lambda \\ \varepsilon \end{bmatrix} \right\}$$

where

- A is the superset of A_d and A_e
- $F({}^{*}L\bullet\omega)$ is the superset of $F_d({}^{*}L\bullet\pi)$ and $F_e({}^{*}L\bullet\eta)$
- P is the superset of P_d and P_e

This superset covers ε and so introduces numerous ambiguities. For instance a name could be either θ or D•ε, and an infix operator with ε as its left operand is indistinguishable from the equivalent prefix operator. A much tighter superset is required.

4.7.3 The Superset

Definition of a superset that covers a *parameter-declaration* and *assignment-expression* requires that their component terms also represent corresponding supersets. We therefore present the component supersets in increasing order of complexity before finally reaching the generalised *parameter-declaration* and showing that it provides superset grammar coverage.

4.7.3.1 Generalised Name

The ambiguity in

```
    prefix ( b )
```

causes extreme parsing difficulties because the C++ grammar prepends the `prefix` to the parenthesis for two different reasons, only one of which is guaranteed to have parentheses present:

- a prefix name is associated with parentheses for a function call
- a prefix name is associated with an optionally parenthesised declarator for a *simple-declaration*

One of these alternatives must be eliminated to resolve the parsing difficulty. Eliminating a function call is undesirable since parsing of function arguments occurs in the midst of productions that enforce arithmetic precedences. Eliminating the name from the *decl-specifier-seq* prefix of declarations is possible.

The solution is to maximise the parsed length of any word-like sequence before associating punctuation. Therefore

```
extern int f()
```

is parsed so that `extern int f` is parsed in its entirety before the parentheses are applied. This avoids the problem of whether a prefixed name may be followed by a parenthesis or not, since a name is only prepended in one place. The cascade of names is parsed first. As a result of this, the parse trees for

```
int (var)
Class(arg)
Class(int)
```

are all the same. Semantic processing must use type information to separate the possibilities and recognise the equivalence of

```
int (var);
int var;
```

The `template` keyword affects the way in which a name is used. It must therefore bind to that name. `template` is therefore always parsed close to the name. This potentially causes a conflict with an *explicit-instantiation* which provides an external binding of the `template` keyword. The conflict is resolved by ensuring that every valid declaration that can participate in an *explicit-instantiation* incorporates the `template` prefix. In practice this means that every prefixing rule that could form part of a declaration in the generalised expression must apply a `template` prefix if it applied any other prefix.

In the grammar analysis, a maximised name is denoted by D$^+$, which is convenient for analysis but does not perform very much of a syntactic breakdown. Classifying the components of D as

u	a user-defined name (e.g. `::name` or `operator const char *` or `class X {...}`) already including an optional `template` prefix
b	a built-in type name (e.g. `short`)
t	the `template` keyword
q	anything else (e.g. `const` or `extern` or `virtual`)

D$^+$ is parsed (using regular expression syntax) as

```
(t*q)* ( (t*b(b|q)*)? (uq*(b(b|q)*)?)* | t*b(b|q)*)
```

rather than

```
(u|b|q|t)+
```

All possibilities are parsed and the requirement to gather `template` prefixes is observed. The `b(b|q)*` grouping captures a complete (multi-word) built-in type name together with all interspersed and trailing qualifiers. The `uq*` similarly captures a single word user-defined name together with all trailing qualifiers. An arbitrary mix of names is permitted recognising that two built-in type-names cannot be adjacent. This grouping ensures that only one AST node need be created for each name and that trailing qualifiers attach to their preceding name. The `(t*q)*` prefix associates any prefix qualifiers with the first name.

The leading `(t*q)*` is separated and not implemented as part of a D^+ parse. This exploits the semantic constraint that a name must contain at least a user-defined or built-in type name, and avoids ambiguities when a generalised name:

- follows a cast generalised with a trailing *cv-qualifier-seq*$_{opt}$ to cover an abstract function declarator

  ```
  (cast) const p
  ```
- follows a cv-qualified pointer

  ```
  int * const p
  ```
- is used with the FOG negated qualifier extension

  ```
  !static int p;
  ```

The omitted prefix `(t*q)*` term is only valid as part of a *decl-specifier-seq* which occurs at the start of certain declarations. The missing specifiers are therefore parsed as a prefix once all other ambiguities have been removed:

```
simple_declaration:    ';'
    |                  init_declaration_list ';'
    |                  decl_specifier_prefix simple_declaration
```

A related complexity arises with pointers to members

```
Class :: *
```

since the following is syntactically valid

```
Type Class::* p = 0
```

In order to pursue the same policy of maximising the name to avoid shift-reduce conflicts, the parse for the pointer scope absorbs all the preceding name components. This must of course eventually be sorted out by the semantic processing, but differs little from the problem of resolving

```
int * * p = 0
```

which the superset parse identifies as an assignment of zero to the product of the name `int` and the dereference of p. This might appear to be a severe misparse, but actually corresponds to an economy of AST nodes. The pointer-to declarator node does not need to exist. Its functionality is folded into the superset multiplier node, which now has one behaviour for types and another behaviour for values.

4.7.3.2 Generalised Array

The two forms of array suffix: $A_d(\kappa^?)$ and $A_e(^+L\bullet\eta)$ are generalised to $A(^*L\bullet\eta)$.

4.7.3.3 Generalised Parentheses

The two forms of function suffix: $E^?\bullet V^?\bullet F_d(^*L\bullet\pi)$ and $F_e(^+L\bullet\eta)$ are generalised to $E^?\bullet V^?\bullet F(^*L\bullet\omega)$ where $\omega \supseteq \pi \cup \eta$

The precedence enforcing parentheses are generalised from $B\bullet{}^*L$ to $E^?\bullet V^?\bullet B\bullet{}^*L$.

Although not necessary to unify declarations and expressions, generalisation of a cast is necessary to avoid shift-reduce conflicts. C is therefore replaced by $E^?\bullet V^?\bullet C$.

In order to parse `delete[]` followed by an unparenthesised expression. $C(\tau)\bullet\alpha$ is further generalised to cover $[\ \eta^?\ ]\ \alpha$ as well as $(\ \tau\ )\ \alpha$.

4.7.3.4 Generalised pointers

ptr-operator is not generalised, however the usage of * as a binary operator is replaced by *star-ptr-operator* in order to accept a *cv-qualifier-seq*$_{opt}$ following a * in a *multiplicative-expression*. Additionally any *decl-specifier* preceding the scope of a pointer to member is associated with the scope, thereby avoiding conflicts and the need to introduce another infix expression operator.

```
unary_expression:
                postfix_expression
     |          "++" cast_expression
     |          "--" cast_expression
     |          ptr_operator cast_expression
     |          suffix_decl_specified_scope star_ptr_operator cast_expression
     ...
multiplicative_expression:
                pm_expression
     |          multiplicative_expression star_ptr_operator pm_expression
     |          multiplicative_expression '/' pm_expression
     |          multiplicative_expression '%' pm_expression
star_ptr_operator:
                '*'
     |          star_ptr_operator cv_qualifier
```

4.7.3.5 Generalised *primary-expression*

In order to cover some forms of *abstract-declarator* as expressions and thereby avoid shift-reduce conflicts, a *primary-expression* is extended to cover an abstract array A•ε in addition to an abstract function covered by the parenthesis generalisation to $E^?$•$V^?$•B•*L•ω. The same

$$\left\{\begin{matrix} A \\ E^? \bullet V^? \bullet B \bullet {}^*L \end{matrix}\right\}$$

term is therefore used as an expression (to support *abstract-declarator*), as an expression prefix (to support a cast) and as an expression suffix (to support function parameters). Use of precisely the same syntax avoids shift-reduce conflicts.

```
primary_expression:
                literal
     |          "this"
     ...
     |          abstract_expression
abstract_expression:
                parenthesis_clause
     |          '[' expression.opt ']'
     |          "template" abstract_expression
parenthesis_clause:
                parameters_clause cv_qualifier_seq.opt exception_specification.opt
parameters_clause:
                '(' parameter_declaration_clause ')'
```

4.7.3.6 Generalised *assignment-expression*

In order to provide complete coverage of a declarator *initializer*, the right-hand expression of an *assignment-expression* using = is extended to accept {} forms.

```
assignment_expression:
                conditional_expression
     |          logical_or_expression assignment_operator assignment_expression
     |          logical_or_expression '=' braced_initializer
     |          throw_expression
braced_initializer:
                '{' initializer_list '}'
     |          '{' initializer_list ',' '}'
     |          '{' '}'
```

The overall form of the generalised *assignment-expression* is

$$\chi = \begin{Bmatrix} A \\ E^? \bullet V^? \bullet F \\ O_s \\ O_p \\ O_i \\ P \\ E^? \bullet V^? \bullet C \\ E^? \bullet V^? \bullet B \bullet {}^*L \end{Bmatrix}^* \bullet \begin{Bmatrix} D^+ \bullet \varepsilon \\ \lambda \\ \begin{Bmatrix} A \\ E^? \bullet V^? \bullet B \bullet {}^*L \bullet D^* \bullet I^? \bullet P_d^+ \end{Bmatrix} \bullet \varepsilon \end{Bmatrix}$$

$\chi \supset \eta$ since $A \supset A_e$, $F \supset F_e$, $P \supset P_e$, $D^+ \bullet \varepsilon \supset \theta$.

4.7.3.7 Generalised *parameter-declaration*

Adding one extra term to χ provides coverage for π, and ensures that $\omega \supset \chi \supset \eta$. The additional term defines ω as a generalised *parameter-declaration*

$$\omega = \begin{Bmatrix} \chi \\ D^* \bullet I^? \bullet P_d{}^+ \bullet \varepsilon \end{Bmatrix}$$

The extra $D^+ \bullet P_d{}^* \bullet I^? \bullet \varepsilon$ exhibits significant prefix ambiguity with a binary expression. It is only the ε at the right-hand side that disambiguates

```
a **** ,
```

from

```
a **** b,
```

Implementation of this term therefore re-uses intermediate expression productions to avoid shift-reduce conflicts, and consequently covers many meaningless sentences.

```
abstract_pointer_declaration:
            ptr_operator_seq
    |       multiplicative_expression star_ptr_operator ptr_operator_seq.opt
abstract_parameter_declaration:
            abstract_pointer_declaration
    |       and_expression '&'
    |       and_expression '&' abstract_pointer_declaration
parameter_declaration:
            assignment_expression
    |       abstract_parameter_declaration
    |       abstract_parameter_declaration '=' assignment_expression
    |       decl_specifier_affix parameter_declaration
```

4.7.3.8 Coverage of the generalised *parameter-declaration*

It must be shown that $\pi \subseteq \omega$ where

$$\omega = \left\{ \begin{array}{c} \left\{ \begin{array}{c} A \\ E^? \bullet V^? \bullet F \\ O_s \\ O_p \\ O_i \\ P \\ E^? \bullet V^? \bullet C \\ E^? \bullet V^? \bullet B \bullet {}^*L \end{array} \right\}^* \bullet \left\{ \begin{array}{c} D^+ \bullet \varepsilon \\ \lambda \\ A_d \bullet \varepsilon \\ E^? \bullet V^? \bullet B \bullet {}^*L \bullet D^* \bullet I^? \bullet {P_d}^+ \bullet \varepsilon \end{array} \right\} \\ D^* \bullet I^? \bullet {P_d}^+ \bullet \varepsilon \end{array} \right\}$$

$$\pi = D^+ \bullet I^? \bullet {P_d}^* \bullet \left\{ \begin{array}{c} \left\{ \begin{array}{c} E^? \bullet V^? \bullet F_d \\ A_d \\ B \bullet {P_d}^* \end{array} \right\}^* \bullet \left\{ \begin{array}{c} \theta_d \\ \left\{ \begin{array}{c} E^? \bullet V^? \bullet F_d \\ A_d \\ B \bullet {P_d}^+ \end{array} \right\} \bullet \varepsilon \end{array} \right\} \\ \varepsilon \end{array} \right\}$$

To keep the equations more manageable:

$$\xi = \left\{ \begin{array}{c} E^? \bullet V^? \bullet F_d \\ A_d \\ B \bullet {P_d}^* \end{array} \right\}^* \bullet \left\{ \begin{array}{c} \theta_d \\ \left\{ \begin{array}{c} E^? \bullet V^? \bullet F_d \\ A_d \\ B \bullet {P_d}^+ \end{array} \right\} \bullet \varepsilon \end{array} \right\}$$

π may be split to create 4 sub-problems:

$$\pi = \left\{ \begin{array}{c} D^+ \bullet {P_d}^+ \bullet I^? \bullet \xi \\ D^+ \bullet I^? \bullet \xi \\ D^+ \bullet I^? \bullet \varepsilon \\ D^+ \bullet {P_d}^+ \bullet I^? \bullet \varepsilon \end{array} \right\}$$

each of which will be shown after first showing that pointers to ξ are covered.

$P_d^* \bullet I^? \bullet \xi \subset \omega$

Of the four alternatives offered by the right hand term of ξ, the top one and bottom two are clearly covered, since $I \subset O_i$, $P_d \subset P$, $\theta_d \subset D^+ \bullet \varepsilon$. The third may be written more fully as

$$P_d^* \bullet I^? \bullet \left\{\begin{matrix} E^? \bullet V^? \bullet F_d \\ A_d \\ B \bullet P_d^* \end{matrix}\right\}^* \bullet E^? \bullet V^? \bullet F_d({}^*L \bullet \omega) \bullet \varepsilon$$

and then rewritten

$$P_d^* \bullet I^? \bullet \left\{\begin{matrix} E^? \bullet V^? \bullet F_d \\ A_d \\ B \bullet P_d^* \end{matrix}\right\}^* \bullet E^? \bullet V^? \bullet B \bullet {}^*L \bullet \omega$$

and then expanding ω gives

$$P_d^* \bullet I^? \bullet \left\{\begin{matrix} E^? \bullet V^? \bullet F_d \\ A_d \\ B \bullet P_d^* \end{matrix}\right\}^* \bullet \left\{\begin{matrix} E^? \bullet V^? \bullet B \bullet {}^*L \bullet \chi \\ E^? \bullet V^? \bullet B \bullet {}^*L \bullet D^* \bullet I^? \bullet P_d^+ \bullet \varepsilon \end{matrix}\right\}$$

Examining top and bottom right-hand terms shows them to be covered by ω.

$D^+ \bullet I^? \bullet \varepsilon \subset \omega$

This term is covered by $O_i^? \bullet D^+ \bullet \varepsilon$ since prefix and suffix operators commute and we generalised the right-hand side of an assignment to support the {} form of declaration *initializer*.

$D^+ \bullet P_d^+ \bullet I^? \bullet \varepsilon \subset \omega$

This is covered by the extra term added for use of a generalised *parameter-declaration* rather than a generalised *assignment-expression*.

$D^+ \bullet P_d^+ \bullet I^? \bullet \xi \subset \omega$

Taking the lexical perspective for ordinary pointers and references: the first P_d in a term which could be a multiplier in a declaration of the form

```
static long int * *a = 0;
```

The problem

$$D^+ \bullet P_d \bullet P_d^* \bullet I^? \bullet \xi$$

can be written, after introducing lexical separation around the first P_d, as

$$D^+ \bullet \varepsilon \quad P_d \bullet \varepsilon \quad P_d^* \bullet I^? \bullet \xi$$

which is covered by

$$D^+ \bullet \varepsilon \quad O_i \bullet \varepsilon \quad P_d^* \bullet I^? \bullet \xi$$

since P_d is covered by O_i, and the possibility of a trailing *cv-qualifier-seq* on P_d is covered by the generalisations to *unary-expression* and *multiplicative-expression*. Introduction of the dyadic O_i partitions the problem into the two smaller problems: $D^+ \bullet \varepsilon \subset \omega$ and $P_d^* \bullet I^? \bullet \xi \subset \omega$.

It is clear that $D^+ \bullet \varepsilon \subset \omega$, and we have already shown that $P_d^* \bullet I^? \bullet \xi \subset \omega$.

For pointers to members such as

```
extern int Class::* p = 5
```

the generalised form of pointer P subsumes the D^+ prefix leaving the problem $P_d{}^*\bullet I^?\bullet\xi \subset \omega$ which has already shown.

$D^+\bullet I^?\bullet\xi \subset \omega$

This may be shown by considering the three alternative locations for the left-most occurrence of a B in:

$$D^+ \bullet I^? \bullet \begin{Bmatrix} E^? \bullet V^? \bullet F_d \\ A_d \\ B \bullet P_d{}^* \end{Bmatrix}^* \bullet \begin{Bmatrix} \theta_d \\ \begin{Bmatrix} E^? \bullet V^? \bullet F_d \\ A_d \\ B \bullet P_d{}^+ \end{Bmatrix} \bullet \varepsilon \end{Bmatrix} \subset \omega$$

There may be no occurrence of a B:

$$D^+ \bullet I^? \bullet \begin{Bmatrix} E^? \bullet V^? \bullet F_d \\ A_d \end{Bmatrix}^* \bullet \begin{Bmatrix} \theta_d \\ \begin{Bmatrix} E^? \bullet V^? \bullet F_d \\ A_d \end{Bmatrix} \bullet \varepsilon \end{Bmatrix} \subset \omega$$

This is covered since the absence of prefix operators allows the D prefix to be combined with the θ_d or ε.

There may only be a B in the final replicator.

$$D^+ \bullet I^? \bullet \begin{Bmatrix} E^? \bullet V^? \bullet F_d \\ A_d \end{Bmatrix}^* \bullet B \bullet P_d{}^+ \bullet \varepsilon \subset \omega$$

which may be rewritten as

$$I^? \bullet \begin{Bmatrix} E^? \bullet V^? \bullet F_d \\ A_d \end{Bmatrix}^* \bullet F(P_d{}^+ \bullet \varepsilon) \bullet D^+ \bullet \varepsilon \subset \omega$$

which is satisfied since the function argument $P_d{}^+\bullet\varepsilon \subset \omega \subset {}^*L\bullet\omega$.

There may be a B in the first multiplier

$$D^+ \bullet I^? \bullet \begin{Bmatrix} E^? \bullet V^? \bullet F_d \\ A_d \end{Bmatrix}^* \bullet B \bullet P_d{}^* \bullet \xi \subset \omega$$

which may be rewritten as

$$I^? \bullet \begin{Bmatrix} E^? \bullet V^? \bullet F_d \\ A_d \end{Bmatrix}^* \bullet F(P_d{}^* \bullet \xi) \bullet D^+ \bullet \varepsilon \subset \omega$$

which is covered since the recursive problem $P_d{}^*\bullet\xi \subset \omega \subset {}^*L\bullet\omega$ has already been shown.

4.7.4 Ambiguities in the superset grammar

The complete C++ grammar implemented using the superset approach is provided in Appendix B. Processing that grammar through bison and yacc provides a rigorous check of the ambiguities, that are described in Appendix F.2. The overall accuracy of the grammar depends on transcription errors and the number of changes required to resolve ambiguities. The grammar is similar to the standard and so relatively easily checked. The more substantial changes to resolve ambiguities are based on establishing a superset of the declaration and expression syntax. The validity of the superset was shown in the previous section. Much simpler proofs are used to show that the same or a slightly tailored superset solves other ambiguity problems. Restructuring of a few productions is necessary to remove or at least resolve shift-reduce conflicts.

Provided that the syntactic parse discards no information, ambiguities that are not resolved syntactically can be resolved semantically. This is easily achieved by ensuring that each reduction fully parameterises each AST node that is created.

A discussion of the resolution of the practical difficulties associated with the detailed C++ syntax may be found in Appendix F.2 including details of the new ambiguities resulting from the superset approach in Appendix F.2.6. The semantic processing required to recover from the reduced syntactic resolution is described in Appendix F.3.

4.8 Back-tracking

Introduction of back-tracking into an LALR parser grammar in order to support a reparse against an alternate syntax proves to be remarkably easy.

4.8.1 Linear search in yacc

The multi-pass FOG productions to perform a linear search through the declaration and expression ambiguities of a `statement` resemble

```
statement: mark declaration_statement unmark
    |      mark remark expression_statement unmark
    |      mark remark error ';' unmark
mark:      /* empty */    { push_input_context_marker(); }
remark:    error          { rewind_input_context_to_marker(); }
unmark:    /* empty */    { pop_input_context_marker(); }
```

in which

- `/* empty */` is a comment to highlight use of the ε terminal
- `{}` surrounds the code invoked when a rule is reduced.
- `error` is a special error recovery token generated by the parser when an illegal token is encountered.

The `mark` rule places a marker on the input token sequence. `remark` back-tracks and starts another attempt to parse the same sequence. `unmark` removes the marker but does not affect the token stream. In practice, statements nest, and this approach is used to resolve other parsing problems too, so `mark` pushes onto and `unmark` pops from a stack of marked positions.

The operation of the parser requires that all three alternatives start by shifting and then reducing the `mark` to ensure that the parser generator has a common prefix, and that the context stack is consistently maintained. The first alternative then attempts to parse the input token sequence as a *declaration-statement* and unmarks if that parse succeeds. The other two alternatives require an error to occur, as will be the case if the *declaration-statement* parse fails. Reduction of the `remark` rule that handles the error rewinds the input context back to the mark so that an attempt at an *expression-statement* parse occurs. If this also fails, the further error of the third alternative is satisfied and error recovery proceeds by discarding tokens until a semicolon is encountered.

This approach directly implements the disambiguation rule (§6.8-1): if it could be a declaration, it is (first alternative), otherwise it’s an expression (second alternative). However the parse is a syntactic one. If it looks like a declaration, it is parsed as a declaration, otherwise it is parsed as an expression. This is not a problem when accurate type information is available, however we are advocating parsing without type information and so those ambiguities that require type information are resolved in favour of declarations (because declarations are parsed in the first pass). Expressions that are misparsed because they look like, but cannot be, declarations must be corrected once type information is introduced in subsequent semantic processing. The nature of these expressions is discussed in Appendix F.2.5 and their resolution in Appendix F.3.1.

Code to detect the misparse has the full construct available as AST nodes, avoiding the difficult partial context and lookahead problems that occur while trying to disambiguate in the grammar. Code to correct the misparse performs a localised tree rearrangement with many of the misparsed declaration AST nodes re-usable as nodes in the corrected expression tree.

The approach is relatively easy to implement in the parser, but requires the input token source to support marking, and back-tracking to specific positions. In practice there is a potential for errors in the grammar. Failure to pair `mark` and `unmark` leads to stack drift that materialises as an unhelpful parsing failure with a generally confusing diagnostic. Parsing control actions are transparent to the parse table generator, and so unnecessary `mark`s may be omitted:

```
statement:    selection_statement
    |         mark declaration_statement unmark
    |         mark remark expression_statement unmark
    |         mark remark error ';' unmark
```

selection-statement:
    `if` (*condition*) *statement*
    `if` (*condition*) *statement* `else` *statement*
    `switch` (*condition*) *statement*

A statement starting with an `if` keyword can only be a *selection-statement*, since `if` occurs nowhere else in the grammar. A *selection-statement* can therefore be parsed without marking the context for backtracking. This works because the parser generator uses one token of lookahead. It may therefore examine the token logically following the mark before deciding which parsing alternatives to retain. This avoids incurring the cost of back-tracking. The corollary is that the marked position in the input token sequence must be adjusted to ensure that the lookahead token is made available following a back-track. A more unpleasant corollary occurs if the omission of a mark is a programming error. In the above example, if a *declaration-statement* could start with an `if`, the ambiguity would be diagnosed by the parse generator as a shift-reduce conflict. However if the *expression-statement* could start with an `if`, no diagnostic results, the generated parser parses the *selection-statement*, but does not perform a back-track for the *expression-statement*. This is one of the harder forms of parsing error to cure: the grammar code looks correct.

The back-tracking approach is able to resolve the type-name or identifier ambiguity adequately because there is very little syntactic difference: both are names. Most ambiguities are correctly resolved in favour of the declaration.

Back-tracking for templates

Back-tracking is less well-suited to resolving the template-name ambiguity, since in one case the syntax involves what should be paired brackets

```
template_name < a , b > - 5      // template_name<a,b> - 5
```

and in the other case an infix operator

```
non_template_name < a , b > - 5 // non_template_name<a , b>-5
```

The ASTs for the two alternatives are rather different, the error in the tree structure is poorly localised, and grows as the number of ambiguous <’s increase.

However, it is resolvable, but not by back-tracking to try an alternative restarting at the <, because the template name may be parsed successfully

```
template_name < a > b [ 5 ]                 // Ok
template_name < a > b [ 5 ] + 7             // Syntax error
non_template_name < a > b [ 5 ] + 7         // Ok
```

and consequently reduced and removed from the parser stack before a syntactic contradiction is detected. For back-tracking to succeed, the entire expression and more generally the entire declaration or statement, must be retried to test each template / non-template hypothesis. This requires a search of the binary tree.

Back-tracking for template-names is perhaps not appropriate for a C++ parser that performs semantic interpretation. Relatively loose coupling between syntactic and semantic processing will suffice to provide the information needed for syntactic analysis, since template-names cannot be introduced in mid-statement.

Back-tracking for template-names may be appropriate for C++ parsers that do not need full program comprehension. For instance, the relatively rare errors from a template misparse need not render a pretty printing program unusable.

However back-tracking for templates is unavoidable in FOG, since semantic information cannot be provided.

4.8.2 Binary tree search in yacc

The lack of context during meta-programming described in Section 3.2.3 mandates parsing without template information, and so a search of the binary tree of possible parses is required. A search may be orchestrated by marking the input token context at the start of each declaration or statement, and associating a binary tree search context with that mark. The grammar

compound-statement:
    { *statement-seq*$_{opt}$ }

statement:
    control-statement
    ...

is therefore implemented as

```
compound_statement:    '{' statement_seq.opt '}'
     |                 '{' statement_seq.opt looping_statement '#' bang error '}'
                                           { UNBANG("Bad statement-seq."); }
statement_seq.opt:     /* empty */
     |                 statement_seq.opt looping_statement
     |                 statement_seq.opt looping_statement '#' bang error ';'
                                           { UNBANG("Bad statement."); }
looping_statement:     start_search looped_statement   { end_search(); }
looped_statement:      statement
     |                 advance_search '+' looped_statement
     |                 advance_search '-'
statement:             control_statement
                       ...
advance_search:        error               { yyerrok; yyclearin; advance_search(); }
bang:                  /* empty */         { BANG(); }
start_search:          /* empty */         { start_search(); }
```

Usages of *statement* are served by `looping_statement` which organises the binary tree search around `statement`. The search iteration is managed by the three functions `start_search()`, `advance_search()` and `end_search()` invoked from the action routines. `start_search()` and `end_search()` occur as the pre and post actions that `looping_statement` imposes upon `looped_statement`, which makes use of the `error` token to repeatedly attempt a parse of `statement`, so long as `advance_search()` keeps injecting a + token into the rewound input stream. `looped_statement` succeeds when a successful and therefore consistent parse is found. Eventually, when all search possibilities are exhausted, `advance_search()` rewinds and injects a - and a # token into the input stream, and the looped_statement again succeeds. A looped_statement never fails therefore looping_statement always pairs calls to start_search() and end_search().

It is necessary for looped_statement to succeed on failure in order to terminate the loop, since an error initiates each re-iteration. However, it is wrong for a parse failure to be turned into a success. The extra # injected into the input stream enables the spurious success to be diagnosed. In the absence of additional rules, the # is illegal C++ (after preprocessing) and so regenerates the suppressed error. Extra rules in compound_statement and statement_seq.opt catch the #, reduce the bang production in order suppress the default unhelpful “parser error” diagnostic, and then require an error token. The error can only arise after an error which therefore occurs and initiates an error recovery search for the next ; or } at which to resynchronize and generate a slightly better diagnostic. Generation of a good quality diagnostic requires the diagnostics associated with the most-nearly-successful loop iteration to be cached during the loop for emission at the end of the loop. The most-nearly-successful metric is conveniently determined as the largest number of tokens parsed prior to the error.

The domain of the search cannot be directly known by the loop management productions. It is maintained as a side effect of identifier parsing.

```
id:             identifier                      %prec SHIFT_THERE
    |           identifier template_test '+' template_argument_list '>'
    |           identifier template_test '-'
    |           template_id
template_test:  '<' { template_test(); }
template_id:    "template" identifier '<' template_argument_list '>'
```

A template ambiguity arises, whenever an identifier is followed by a < without a template prefix. The ambiguity is resolved, in the absence of template information, by the prevailing state of the binary tree search. This state is determined by the template_test() routine which injects + to select a template interpretation or - and < for an arithmetic interpretation.

The template_test rule introduces a shift-reduce conflict between the usage in the expression syntax and the usage presented above. This conflict is resolved by the %prec to force all occurrences of identifier < to take the test. When the test injects a +, parsing continues by analysing template arguments. When the test injects a - and <, the - satisfies the parse for an identifier and the extra < restores the token required to proceed with the arithmetic interpretation.

On the first traversal of source tokens, template_test() increases the binary tree search depth and selects the template hypothesis. On subsequent traversals, template_test() signals the prevailing binary tree search hypothesis.

It is rarely necessary to perform the full binary tree search, since a branch and bound can exploit early failure and avoid searching other hypotheses that share the same failing prefix.

There is unfortunately no guarantee that the first accepted hypothesis is correct, and so the subsequent semantic analysis must be prepared to reorganise the AST to accommodate errors in either direction.

It should be noted that there is no error handling for template arguments, so that a template argument parse failure propagates onwards enabling the statement level binary tree search to poll syntax alternatives.

Cost

This algorithm clearly has exponential complexity with respect to the number of ambiguous <’s in a statement, although this complexity does not arise in practice as is demonstrated by applying the grammar of Appendix C to three large bodies of code to determine its cost. The results are summarised in Table 4.5.

Code body	Preprocessed Lines	Statements	*identifier* < statements	% ambiguous	Back-tracks							
					0	1	2	3	4	5	6	7
fog	88133	38470	290	0.75	166	124	0	0	0	0	0	0
product	119383	44353	1652	3.7	784	851	9	8	0	0	0	0
gcc	426797	127366	2099	1.7	29	2047	7	21	1	2	2	1

Table 4.5 Back-tracking costs

The *fog* code is the entire C++ source code for FOG. Fewer than 1% (290 out of 38470) of statements contained an identifier followed by a <, and a consistent syntax for these was usually (166 out of 290) found without back-tracking by assuming the template form. A consistent syntax was always found after one back-track.

The *product* code is the entire C++ source code for a proprietary product. A higher proportion but still fewer than 4% of statements contained an identifier followed by an <, and a consistent syntax for these was almost always found after one back-track. A consistent syntax was always found after three back-tracks.

The *gcc* code comprises almost the entire source code of the gcc compiler. This is C code and so gives worst case performance under the back-tracking presumption to try template syntax first. Nearly 2% of statements were ambiguous. 29 statements were incorrectly resolved as templates without back-tracking. 2047 statements were correctly resolved after the back-track necessary to change the template presumption. The higher number of back-tracks do not indicate the correctness of their parsing conclusion, however since the maximum depth of the binary tree search was 6, it is clear that the use of branch and bound in the search ensured that the 64th solution was found in at most 8 tries.

These results confirm the intuitively expected behaviour. Arithmetic expressions tend to have at most one relational operator, and so favouring template usage when syntactically consistent results in few expressions that need semantic correction.

The exponential search complexity does not arise in practice. From the C++ results, we may infer that back-tracking may occur for 50% of ambiguous statements, which appear to be about 2% of all statements. From the gcc results we may infer that a wrong parse is found 1% of the time. The syntactic cost is therefore an extra 1% of syntactic statement analysis, which is perhaps half of the total syntactic analysis. The semantic cost involves the correction of a template misparse for 0.01% of statements.

4.9 FOG grammar

The full FOG grammar in Appendix C uses similar techniques to those for the simpler C++ grammar in Appendix B.

In addition to the use of back-tracking in the superset grammar to solve the C++ parsing problems

- template name (Appendix F.2.1.3 and Appendix F.2.1.4)
- bit field or inheritance (Appendix F.2.4.1)

the back-tracking approach is also used to solve the following parser problems

- replacement of formal meta-parameters (Section 3.3.10)
- “old-style” C (type I) function declarations(Appendix F.2.4.2)

The preliminary scan of the body of a meta-function (Section 3.3.11) marks the position of the start of any tree-literal and attempts to parse a formal parameter.

If a formal parameter is referenced, its value is used. Otherwise the parser back-tracks and copies the tree-literal unchanged into the meta-function body.

Multi-pass implementation

The multi-pass implementation of FOG parsed first for declarations and then back-tracked for expressions. Further nested back-tracking was used to resolve

- cast ambiguities (Appendix F.2.5.2)
- new placement / initializer ambiguity (Appendix F.2.5.5)
- `sizeof` ambiguity (Appendix F.2.5.6)
- `typeid` ambiguity (Appendix F.2.5.7)
- type / value template arguments (Appendix F.2.5.8)

The multi-pass implementation did not parse statements within function bodies further than the replacement of $-expressions. It therefore incurred very few of the severe ambiguities that arise when expressions and declarations coexist.

4.10 Code Structure

The file input processing acquires tokens from source files and performs ANSI C preprocessing. The implementation closely follows the translation phases described in §2.1. It is shown in pictorial form in Figure 4.4.

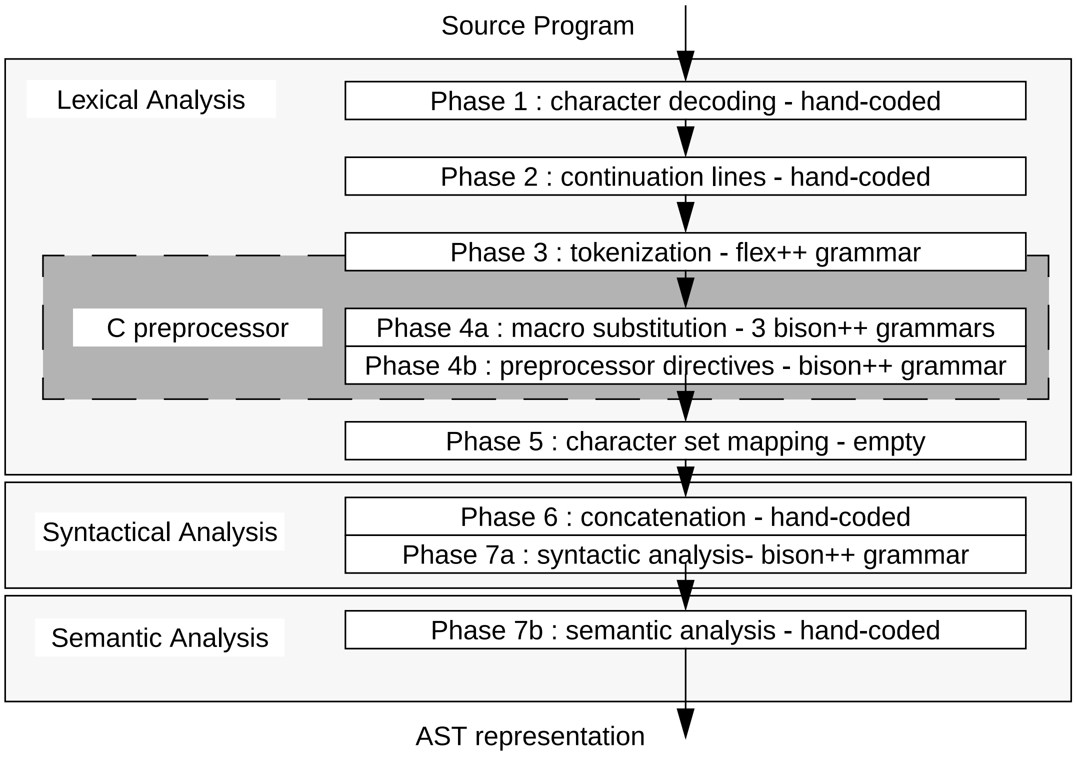
Source Program
Lexical Analysis
Phase 1 : character decoding - hand-coded
Phase 2 : continuation lines - hand-coded
Phase 3 : tokenization - flex++ grammar
C preprocessor
Phase 4a : macro substitution - 3 bison++ grammars
Phase 4b : preprocessor directives - bison++ grammar
Phase 5 : character set mapping - empty
Syntactical Analysis
Phase 6 : concatenation - hand-coded
Phase 7a : syntactic analysis- bison++ grammar
Semantic Analysis
Phase 7b : semantic analysis - hand-coded
AST representation

Figure 4.4 FOG analysis phases

Phase 1 (character mapping, trigraph replacement and universal character replacement) and phase 2 (backslash line continuation) are performed by nested subroutines in the input routine to the lexer.

Phase 2 operates in a demand-driven fashion, returning one line at a time to the lexer.

The lexer, automatically generated by flex++, performs phase 3. The lexer is very simple (150 lines of rules), just identifying each of the different lexemes, and creating an appropriate derived instance of the polymorphic FogToken.

Construction costs are reduced by making substantial use of the Flyweight pattern [Gamma95], with all tokens stored in a tokenization symbol table which is initially populated by special objects for all the reserved words and punctuation sequences. Whenever a text string is tokenized, the appropriate Flyweight is chosen, so that a string such as and_eq is encoded as an object that retains its spelling for the purposes of stringizing, but which otherwise behaves the same as the object created for the text string &=. Reserved words such as `bool` or `else` are similarly associated with objects that have relevant behaviour.

Minor complexities are resolved by using four alternate lexer states to handle

- normal tokenization
- waiting for `*/` in a multi-line C-style comment
- recognising preprocessor keywords between the # and subsequent directive
- non-standard tokenization after a `#include`.

Phase 3 operates in a demand driven fashion, returning one lexeme at a time. Each lexeme has a dual identity, as a preprocessor token and as a token.

The phase 4 preprocessing directives and macro substitution are not performed by the lexer. Phase 4 processing is performed by two further layers of processing, each operating in a demand-driven fashion to return one lexeme at a time to its caller. The inner layer performs macro substitution. The outer layer detects and implements preprocessor directives. Macro substitution is performed by three very simple parser grammars that

- locate any macro arguments following a function-like macro
- substitute any identifiers in the macro replacement
- retokenize around ## and stringize # identifier

Substitution (scan 1) and replacement (scan 2) of identifiers may involve a recursive substitution, but not where scan 1 detects an identifier matching the current macro (§16.3.4-2). The replacement for the detected macro and arguments is determined as soon as the macro is detected, with the resultant token sequence buffered and gradually drained by successive demands for more lexemes.

Preprocessor # directive lines are parsed using a further grammar, with identifier substitution bypassed except where permitted by the syntax. Conditional directives maintain a state stack, the top of which determines whether lexemes from non-preprocessor lines should be discarded. `#define` and `#undef` maintain the dictionary of macro definitions referenced by the inner preprocessor layer. `#include` pushes a new source context on to the lexer.

The lexemes returned from the outer preprocessor layer comprise the token stream after all ANSI C preprocessing has been completed, and before any FOG extensions have been realised. The token stream includes whitespace and new lines. To improve error diagnostics, each token is returned with its source file, source line and source column context.

There is no phase 5 in FOG, the execution and source character sets of a source to source translator are necessarily the same. There is however a distinct internal character set in which escape sequences, universal characters and other characters are uniformly represented by binary values within characters and strings. The uniform internal representation forms the basis for concatenation, and is created (lazily) when the string or character lexeme is created during phase 3. Each number, string or character has three spellings: the original ASCII source spelling to be used by preprocessor stringizing, the uniform internal binary representation, and a regenerated ASCII spelling for output purposes. The output may need to differ from the input to preserve the meaning of concatenated escape sequences (§2.13.4-3):

`"\xA" "B"` concatenates to `"\012B"` not `"\xAB"`

The concatenation phase 6 and the analysis phase 7 are described in Section 2.1.1.

Early FOG implementations performed $-substitution, Cpp substitution and #directive resolution within the lexer, permitted $-substitution within characters and strings and identified reserved words explicitly. The implementation was large, complicated, vulnerable to obscure programs and lacked clear semantics. The current implementation is simple, with a clear responsibility for each processing layer. The leakage between layers is very constrained:

- macro definitions are accessible via the `std::get_cpp()` meta-function
- the `using` form of include currently shares the same low-level implementation mechanism as `#include`
- the source file identification describing the name and origin of each source file contains a mixture of pre-processor and non-preprocessor attributes

4.11 Summary

The complexity of the FOG grammar and the success of the multi-pass and superset approaches can be judged by comparing the parser generator statistics for three alternative C++ grammars. Table 4.6 provides a summary of the statistics variously extracted from the source grammar, from the generated output or from the report file of the modified version of bison++ used to build FOG.

	Roskind	gcc	CPPP	C++ super-set	C++ multi-pass	FOG multi-pass	FOG super-set
Unresolved Shift-Reduce conflicts	24	5	1	0	0	0	0
Unresolved Reduce-Reduce conflicts	18	38	0	0	0	0	0
Resolved conflicts	0	704	410	24	14	16(+1)	32
`%prec` usage	0	109	22	8	6	6(+1)	15
Tokens with precedence	0	61	40	14	4	4	17
Rules	664	779	431	558	661	1456	939
States	1257	1399	702	894	1119	2217	1550
Non-ASCII terminals	77	82	104	95	82	241	110
YYLAST	13954	9534	3724	5507	6501	30119	12893

Table 4.6 Grammar Statistics

The figures should be treated with some caution, because the goals and completeness of each grammar are not comparable. (A smaller number is better for all entries.)

The Roskind grammar is accurate, but lacks templates and other modern facilities.

The gcc grammar is an almost complete C++ grammar, and contains a few extensions. The grammar code also contains error recovery and performance optimisations.

The CPPP grammar is preliminary, and lacks some modern facilities.

The multi-pass and superset grammars aim for syntactic consistency, whereas the other grammars aim for semantic accuracy.

The C++ superset grammar (Appendix B) is a complete working C++ grammar with error recovery. The grammar unifies the declaration and expression syntax to resolve ambiguities.

The FOG multi-pass grammar is that of the earlier FOG implementation. It is fairly complete, adds syntax for meta-level processing, has some error recovery, but does not provide parsing within function bodies.

The C++ multi-pass grammar is a version of the FOG multi-pass grammar trimmed to handle just C++ but adapted to parse all of C++. The grammar contains some error recovery, and a structure that retains some unnecessary generality from the full grammar. The grammar has never been used. The results serve to compare the parsing approach adopted in FOG with other C++ approaches, and to determine the extra complexity introduced by FOG extensions.

The FOG superset grammar (Appendix C) is the working FOG grammar, using unified parsing and incorporating error recovery.

Conflicts

Each conflict provides an opportunity for a programming error to yield an incorrect grammar implementation. The low number of conflicts in the Roskind and even smaller number in the new grammars ensure that the residual conflicts can be analysed by hand. Roskind makes a point of avoiding %prec, relying on line ordering in the grammar instead. This is an understandable policy with yacc, since yacc does not report conflicts resolved by using %prec. The use of %prec in FOG, and validation by bison, allows lines to be reordered and explicitly identifies each conflict where that conflict is intended to be resolved. The extensive use of precedence by gcc and CPPP and the very large number of tokens which can participate in precedence resolution offer ample scope for a conflict to be silently and erroneously resolved.

The FOG grammars use only two active precedence levels, reducing hazards of fortuitous resolution between levels. Only one token has a distinct precedence level. Two bordering precedence levels have the dummy tokens REDUCE_HERE and SHIFT_THERE so that precedence rules are coded as in Appendix F.2.3.1.

The superset and multi-pass approaches clearly have fewer conflicts. The multi-pass approach has fewest, since grammar conflicts are resolved in independent passes. The (+1) on the FOG grammar accounts for the dangling else resolved by the token acquisition routine. Substantially larger numbers should be added to gcc and CPPP metrics to account for their lookahead code.

Size

The number of rules in a grammar corresponds to the line count in a more conventional program, and so measures the programming effort. The number of states in the generated state machine is a measure of the complexity of the problem that the program solves. YYLAST is the dimension of the two largest tables generated for the table-driven parser. YYLAST therefore provides a good indication of the executable size after all compactions have been performed. The size of the FOG multi-pass grammar is close to the 32767 implementation limit for the standard bison parser. During development FOG exceeded that limit, which is why a modified version of bison++ is used to support 99999 table entries.

The number of rules and states in the Roskind and gcc grammars are surprisingly similar, more so after allowing for the greater completeness and error recovery in the gcc grammar. The CPPP grammar is noticeably smaller, which might be an indication of a better grammar, or confirmation of its inaccurate status.

The C++ superset grammar is noticeably smaller, the benefit of merging declarator rules into expressions rather than flattening out to try to create more lookahead.

The trimmed C++ multi-pass grammar is similar in size to the gcc and Roskind grammars. Each of these grammars is over-size through the use of grammar flattening to extract or resolve extra semantic context at the syntactic level.

The full multi-pass FOG grammar is at least twice as large, the result of some extra grammar for the meta-level syntax and the extra rules to support syntax-driven interaction with meta-functions and meta-variables. The extra rules are mostly simple. It would be wrong to infer that FOG doubles the grammar complexity.

A fairer comparison of the parsing cost of FOG can be drawn from the more effective superset grammar which is rather less than twice the complexity of its C++ counterpart and pretty similar to the gcc and Roskind grammars.

Parsing Assistance

The number of non-ASCII terminals used by the grammar provides a little insight into the assistance provided to the parser. Some non-ASCII terminals are unavoidable to represent the 24 special character sequences (+=), 63 primary reserved words (`class`), 11 alternate reserved words (xor_eq) and 4 parametric tokens (*identifier*). The exact number of such terminals depends upon whether alternate reserved words are resolved by the lexer, and whether groups of similar reserved words such as `private`, `protected` and `public` are replaced by a parametric token such as *access-specifier*. A practical implementation uses more than the minimum number of terminals in order to communicate extra disambiguation information to the parser.

The Roskind figure does not account for some more recent reserved words and has one extra disambiguating token.

The gcc figure is more representative, adding six disambiguating tokens.

The larger value for CPPP is the result of the extra information provided by its lookahead parser.

The values for the superset C++ grammar correspond to exactly one token per reserved word, punctuation sequence, and 5 parametric tokens.

The much larger figure for the FOG multi-pass grammar is mostly caused by the need for two tokens per meta-type, one to prime the parser to perform the correct syntax-driven parse, and one to identify the meta-type of a meta-variable instantiation.

The additional tokens for the FOG superset grammar are required for the non-reserved words.

Performance

The superset approach incurs minimal costs from back-tracking, since a back-track occurs only for very unusual code and the occasional template retry. The parser is probably slightly slower because more reductions are performed for the many levels of expression precedence than for the normal nesting of declarators. This cost could be alleviated by flattening the expression syntax and using precedence as in the CPPP grammar. The extra semantic disambiguation will incur a small cost, but since it occurs within the disciplined context of a tree rather than the less appropriate parser stack, there is probably a small net saving. The need for semantic correction to the AST is rare and so should incur very little cost.

The overall relative speed cost of the superset approach is probably slightly adverse, but well justified given the elimination of intra-statement coupling between syntactic and semantic analysis stages and the possibility of demonstrating that the grammar implementation is accurate.

The parsing tables for the superset grammar approach are clearly smaller. The AST provides an easier environment in which to resolve ambiguities. It is therefore likely that deferring ambiguity resolution to the semantic level may contribute a further small code saving.

5 Files

This chapter is concerned with the code generation stages of the FOG to C++ translator; that is the preparation of the AST for output and then emission of the AST as C++ code.

We first describe some practical problems that arise from the use of a meta-compiler and introduce syntax extensions to resolve these problems and also to replace `#include`. Then we describe the declaration dependency analyses necessary to establish a legal ordering and the subsequent emission of output files using that ordering.

5.1 Practical problems

The compilation model for FOG is shown in Figure 5.1.

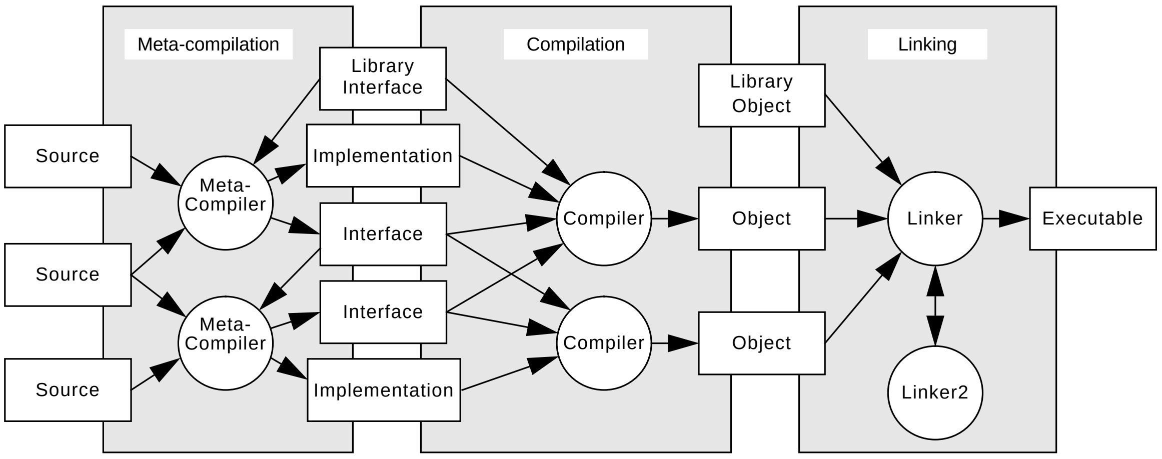

Figure 5.1 Meta-compilation model

The centre and right hand sides show the conventional C++ compilation model. Interface files are shared by independent compilations, which produce object files to be linked together with libraries to produce an executable. The complexities of static construction and template instantiation are conveniently hidden by the “Linker2” activity. Meta-compilation adds the extra stages on the left hand side. The conventional C++ interface and implementation files are generated by one or more meta-compilations from source files (the forward arrows) and from frozen interfaces (the reverse arrows). Sources may be shared between meta-compilations. A single meta-compilation may generate any number of interfaces and/or implementations as its output files.

Three practical problems arise from the use of a meta-compiler as an independent translator prior to a conventional compiler, rather than an integrated part of that compiler.

The new problems concern

- naming and location of output files
- mapping of declarations to output files
- unique and consistent generation of each output file

The problems are resolved by providing a default naming and placement policy, which may be at least partially suitable for many applications. Syntax extensions are necessary to support full control of output files for more interesting applications. These extensions are an unfortunate but unavoidable corollary of the need to generate multiple outputs.

Consistency between sessions is facilitated by the concept of utility and classification of source declarations as

- unique to a meta-compilation session
- shared contributions to multiple sessions
- shared immutable reference declarations

5.2 File disposition

FOG translates extended C++ to plain C++, which must comply with the C++ standard. In particular the resulting C++ declarations must satisfy the One Definition Rule (§3.2); each declared entity must be defined just once. FOG must therefore partition declarations between interface and implementation files.

A conventional C or C++ compilation processes many input files and generates a single output file. The input files and output file are readily specified on the compilation command line in conjunction with additional information to identify search paths for include files. This policy does not extend directly to a meta-compiler that may generate many output files, whose existence may be unknown to the author of the command line.

The default behaviour of FOG is to emit an implementation and an interface file for each non-nested class and each namespace encountered in its source files. Template specializations are emitted with the primary template. Template instantiations may be placed by filespaces (Section 5.4.4). Command line options allow specification of

- default paths for output directories
- output file prefixes (such as sys/)
- output file suffixes (such as .hxx)
- a file name for the global namespace

Files are named from their constituent class or namespace, with interface files acquiring a .hxx suffix, and implementation files a .cxx suffix. Template files are given distinct .H and .C suffixes. The following code is therefore partitioned according to the comments

```
class MyClass                           // MyClass.hxx
{
   int i;                               // MyClass.hxx
   !inline void f() { ... }             // MyClass.cxx (and hxx)
   class Nested {};                     // MyClass.hxx
};

template <class T>
class MyTemplate                        // MyTemplate.H
{
   static T t;                          // MyTemplate.C (and H)
};

template <>
static int MyTemplate<int>::t = 0;      // MyTemplate.C (and H)
```

Include file guards are incorporated into all generated files. The guard name is determined by converting all alphanumeric characters of the file name to upper case and non-alphanumerics to underscore. The file MyClass.hxx is:

```
/*!$@FOG@$!
 *  Generated at Fri Jul  2 14:31:02 1999
 *
 *  by fog 1.0.1 of 08:17:40 Jul  1 1999
 *
 *  from
 *     file.fog
 */
#ifndef MYCLASS_HXX
#define MYCLASS_HXX
class MyClass
{
private:
   int i;
private:
   void f();
private:
   class Nested
   {
   };
};
#endif
```

The file naming and disposition policy outlined above provides a convenient and often adequate default. Programmers may use environment variables, command line tokens or extended declaration syntax to choose their own modularization where appropriate.

5.3 Utility

It is rarely appropriate to emit interfaces and implementations for every class, since this generates new interfaces for standard classes such as `iostream`. Although it could be convenient to generate a new interface with an extra virtual function, this requires availability of the source and an ability to recompile all code that makes use of `iostream`.

In practice, certain external classes must be immutable. In order to support multiple meta-compilations, it is also desirable to group application classes into sub-systems which are mutually immutable.

This problem is resolved by extrapolating from the concept of a class utility [Booch94], where a class utility is free-standing code independent of the current application.

FOG associates a utility level with each source file and source declaration, and determines a utility level for each output file and output declaration. On output, the utility level is used to suppress output for declarations independent of the current meta-compilation, and to diagnose inconsistent packaging of declarations.

Utility levels

The utility levels in increasing order of stringency are:

`pool`

The pooled utility level supports source files contributing to more than one meta-compilation session. The source file contains a pool of declarations, some of which may be output in one session, others in another.

Files and declarations with `pool` utility are not emitted, however once composed with other declarations or into scopes with `emit` utility, the declarations contribute to emitted files.

In a typical usage, a `pool` utility file defines some virtual functions for classes A, B and C. These virtual functions support some algorithm-centric programming concern that cuts across the conventional data-centric organisation of the classes. When the conventional class file (with `emit` utility) is meta-compiled in conjunction with the `pool` file, the `pool` declarations for that class are promoted to `emit` and so complete interface and implementation files for the class are generated. Declarations for other classes remain as `pool` declarations and do not provoke partial or conflicting emission.

`emit`

This is the default utility level. It imposes no constraint on the composition or emission of declarations. All declarations should be emitted to some output file.

`utility`

This is the most stringent utility level. Declarations from a `utility` file are not to be changed or overwritten. This utility level is the same as `frozen`, differing only in that the name indicates that the file was included via `using/utility` rather than `#include`.

Output files with a `utility` or `frozen` utility level are not emitted and any contained declarations of `emit` level are diagnosed.

`frozen`

The `frozen` utility level is applied automatically whenever a file is read as a result of a `#include` directive. Declarations from such a file can only be used, they cannot be changed, and of course the file cannot be overwritten.

Semantics

The more stringent of the prevailing utility level and the enclosing scope utility level is applied to each potential declaration. Composition of potential declarations to produce the final actual declaration again selects the most stringent utility level. No utility level is maintained for forward declarations. The utility level is not inherited.

As each source file is included, any specified non-`emit` utility becomes the prevailing utility to be applied to all declarations in that source file. The previous utility level is restored on completion of the file. The source file utility is specified by a switch as part of the replacement syntax for `#include`.

A more stringent utility level may be explicitly specified for an individual declaration. When that declaration is a scope, the utility level applies recursively to all declarations in that scope.

A more stringent utility level may be also be specified for a potential output file to ensure that its usage is restricted to references from generated `#include` directives. Emission of declarations to that output file is inhibited.

Compilation

The utility of each declaration is recursively propagated to enclosing scopes and containing files, so that any scope or file has the most stringent utility of any declaration in that scope or file. (No propagation occurs for namespaces). The results of this propagation are checked in a verification pass over the AST. Any declarations with a less stringent utility than their enclosing scope are diagnosed as illegal attempts to compose additional functionality where a change is not permitted. Similarly any declarations with a less stringent utility than their containing file are diagnosed as attempts to modify immutable files.

5.4 Syntax

Additional syntax supports

- include declarations (replacing `#include`)
- specification of and allocation to output file names
- specification of generated include file dependencies for scopes
- specification of generated include file dependencies for code

This syntax contributes four new categories of declaration, only three of which are supported as parts of classes. Include declarations are only valid at global scope. The integration of these declarations with the main grammar is described in Section 2.1.5. The syntax to support function-specific and function-body-specific declarations is described in Section 2.1.4.7.

5.4.1 Target File Names

The names of target files are used in two ways.

- to create the file in the file system
- to reference the file in a generated `#include` directive

These two names need not be exactly the same. The name used to create the file must be complete and specified with respect to the current user directory. Names appearing in generated `#include` directives need only be resolvable by the subsequent compiler, with respect to a potentially different current working directory and using a search list of include file paths.

The file name used to refer to a file in a generated `#include` directive is composed from 3 parts

`prefix name suffix`

The *`name`* is normally that of a class or namespace.

The way in which *`prefix`* and *`suffix`* components are determined is summarised in Table 5.1.

The *`suffix`* is normally one of `.cxx`, `.hxx`, `.C`, `.H` depending upon whether the file is associated with the implementation or interface of a non-template or template class. These defaults may be changed by defining the corresponding environment variable. This default setting may in turn be overridden by a command line token. Finer-grained settings can be supplied using the appropriate switch in the source code syntax of a *file-name*.

The *`prefix`* may be used to locate nested include files such as `sys/stdlib.h`. The *`prefix`* has no default, but may be given a value from an environment variable, command line option or source code switch. For template prefixes (and paths) the environment variable option is determined first from a template-specific name, and if that is undefined the non-template name is used.

When files are created, a further path prefix may be added. If no prefix has been defined then the file is created as

`prefix name suffix`

otherwise the file is created as

`path separator prefix name suffix`

component	file	template	default	environment variable	command -line token	source code switch
path	interface			Fog::interface_path	-hd	/path
		template		Fog::template_interface_path Fog::interface_path	-htd	
	imple-menta-tion			Fog::implementation_path	-cd	
		template		Fog::template_implementation_path Fog::implementation_path	-ctd	
prefix	interface			Fog::interface_prefix	-hp	/prefix
		template		Fog::template_interface_prefix Fog::interface_prefix	-htp	
	imple-menta-tion			Fog::implementation_prefix	-cp	
		template		Fog::template_implementation_prefix Fog::implementation_prefix	-ctp	
suffix	interface		.hxx	Fog::interface_suffix	-hs	/suffix
		template	.H	Fog::template_interface_suffix	-hts	
	imple-menta-tion		.cxx	Fog::implementation_suffix	-cs	
		template	.C	Fog::template_implementation_suffix	-cts	

Table 5.1 File Name Component Contributions

where `path` is the specified path, and `separator` is a file system dependent joining character. Only `/` is implemented, which is suitable for any file system with a C interface.

5.4.2 Target File Identities

When the name of a file is specified, it may be qualified to override the default naming or structuring policies.

file-name:
- *string-literal*
- *file-name* / `interface`
- *file-name* / `implementation`
- *file-name* / `template`
- *file-name* / *utility*
- *file-name* / `guard` = *string-literal*
- *file-name* / `noguard`
- *file-name* / `path` = *string-literal*
- *file-name* / `prefix` = *string-literal*
- *file-name* / `suffix` = *string-literal*

The file name is specified as a normal string (or concatenation yielding a string), rather than the subtly different syntax for a string in a `#include` directive. File names containing escape sequences must therefore be appropriately escaped.

It is assumed that the *string-literal* incorporates `prefix`, *name* and `suffix` components, unless `/interface` or `/implementation` is specified, in which case the *string-literal* should comprise just the *name* component.

/interface
/implementation

Specify that a *`prefix`* and a *`suffix`* should be applied to the *name* using values determined in accordance with Table 5.1.

/template

Specifies the use of the template, rather than non-template, policy for any *`path`*, *`prefix`* or *`suffix`*.

/*utility*

Specifies a utility level of an output file. Only `/utility` (or `/frozen`) is meaningful, as a further assurance that a file should not be emitted.

/guard = *string-literal*

Specifies the spelling of the include file guard, overriding the default derived from the file name and its suffix.

/noguard

Specifies that there should be no include file guard in the target file.

/path = *string-literal*

specifies the *`path`* to be prefixed to the name when creating the file. If neither `/interface` nor `/implementation` is used, the implementation rather than interface path is used in accordance with Table 5.1.

/prefix = *string-literal*
/suffix = *string-literal*

specify an override for the *`prefix`* and/or *`suffix`* parts of the file name, but are only used if one of `/interface` or `/implementation` has been used to specify an algorithmic contribution to the name.

5.4.3 Target File Placement

The target file for a particular scope may be changed by a *file-placement-declaration*.

file-placement-declaration:
  `export / implementation =`$_{opt}$ *file-specifier* `;`
  `export / interface =`$_{opt}$ *file-specifier* `;`
  `export / noimplementation ;`

file-specifier:
  file-name
  file-entity
  file-entity `/ implementation`
  file-entity `/ interface`

file-entity:
  declarator-id
  elaborated-type-specifier
  `namespace` *scoped-id*

`export/implementation` specifies that the implementation (non-inline function bodies and initialised variables) should be located in *file-specifier*. Similarly `export/interface` specifies that the interface should be located in *file-specifier*.

Each *file-specifier* may be a *file-name* (the qualified name of a file) or *file-entity* (the name of some declared entity). In the case of a *file-entity*, the required file is either the implementation or the interface file to which the declarations of the entity are emitted. An explicit distinction is made by use of `/implementation` or `/interface` to qualify the entity name. An implicit distinction is made in the absence of an explicit qualifier, by using the `/implementation` or `/interface` qualifying the `export` keyword.

The default placement recursively locates declarations with their enclosing class. Top level classes and namespaces are located in a file named from the declaration name. This policy may be overridden to support

- arbitrary placement

```
class Base
{
    export/implementation "Bases.cxx"; // emit implementation to Bases.cxx
    export/interface "Bases.hxx";      // emit interface to Bases.hxx
};
```

- placement of a derived class with its base:

```
class Derived : public Base
{
    export/implementation Base;  // emit implementation with
                                 //          implementation of Base
    export/interface Base;       // emit interface with interface of Base
};
```

- placement of a local class entirely within an implementation file:

```
class Local
{
    export/implementation Base;                 // emit implementation with
                                                //      implementation of Base
    export/interface Base/implementation;       // emit interface with
                                                //      implementation of Base
};
```

`export/noimplementation` specifies that there should be no implementation, as is often intended to be the case for private constructors and assignment operators. Specification of `export/noimplementation` ensures that any violation of this intent as a result of composition is detected.

5.4.4 Filespaces

A Filespace is a set of declarations grouped in order to share file placement. A filespace therefore supports placement of selected declarations in a specific pair of implementation and interface files. A Filespace is not a declarative region, and so declarations appearing within a Filespace form part of the enclosing scope.

filespace-specifier:
    `namespace / file` *file-name compound-declaration*

filespace-declaration:
    filespace-specifier `;`

The implementation and interface files are determined on the assumption that *file-name* constitutes a suitable *name* for application of *`prefix`* and *`suffix`* components in accordance with Table 5.1. Since both interface and implementation file names are determined algorithmically, the use of `/interface` or `/implementation` on the *file-name* is redundant. A *file-placement-declaration* may be used to change one of both of the file names. The interface file is only used for namespaces and function definitions inlined in the interface, since the declarations of a class must necessarily occur within the one class: they cannot be partitioned across multiple interface files.

In

```
    class A
    {
        namespace/file "FirstRegion"
        {
            // ...
        };
        namespace/file "SecondRegion"
        {
            export/implementation B;
            // ...
        };
    };
```

declarations within `FirstRegion` form part of class A and are typically emitted to `A.hxx`, `FirstRegion.hxx` and `FirstRegion.cxx`. Declarations within `SecondRegion` are similarly part of class A, but are typically emitted to `A.hxx`, `SecondRegion.hxx` and `B.cxx`.

Filespaces may be used to compose a file structure upon the global namespace or upon C source, and to place *explicit-instantiation*s.

5.4.5 Target File Dependencies

Files emitted by FOG have include file guards around declarations and file inclusions that are arranged to ensure that C++ requirements for forward references are satisfied. Include file references and forward declarations are generated by analysis, and as a result are often tighter than those produced by hand.

The current implementation of FOG performs a very limited analysis of function bodies, so FOG may miss dependencies. A perfect analysis could not guarantee to catch all dependencies, given the inadequate type information available for template arguments of templates that remain uninstantiated.

Target file dependency declarations are provided to allow missed declarations to be specified.

file-dependency-declaration:
  `using` / `implementation` =$_{opt}$ *file-specifier* `;`
  `using` / `interface` =$_{opt}$ *file-specifier* `;`

These concepts correspond to “uses (for interface)” and “uses (for implementation)” in [Booch91]. However, whereas a full model requires all dependencies to be specified, FOG only requires those that it fails to deduce.

A dependency declaration may be used as a class *member-declaration* to specify a dependency for all declarations within the class

```
class Base
{
    using/implementation "stdlib.h"; // Entire implementation uses stdlib.h
    using/interface string;          // Interface uses string
};
```

or as an *object-statement* to restrict the dependency to a single function, irrespective of its derivation context

```
class Base
{
    public void f()
    :{                                  // N.B. Start of function scope not body.
        using/implementation iostream;
        { ... };                        // Base::f implementation uses iostream
    };
};
```

or as part of a *function-used-block* to further restrict the dependency to the derivation contexts in which the associated *function-body* is used.

```
class Base
{
    public void f()
    :{                                    // N.B. Start of function scope not body.
        using iostream { ... };           // Function contribution uses iostream
    };
};
```

5.4.6 File inclusion

A replacement syntax for `#include` is provided by overloading the `using` keyword and recognising that `using` followed by a string has no meaning in C++.

<u>*include-declaration*</u>*:*
  `using` *slash-include*$_{opt}$ *slash-utility*$_{opt}$ *string-literal* `;`

<u>*slash-include*</u>*:*
  `/ include`

<u>*slash-utility*</u>*:*
  `/` *utility*

<u>*utility*</u>*:*
  `emit`
  `pool`
  `utility`
  `frozen`

The *utility* indicates whether the included declarations belong to a frozen free-standing external class utility, to a pool of declarations, or whether the included declarations are to be emitted. The default behaviour is for all declarations arising from `#include` files to be treated as frozen utility declarations that must not be changed or re-emitted. The replacement syntax for `#include` provides qualifiers to select the nature of included declarations. The replacement syntax is only valid as a global-scope declaration. Inclusion only occurs upon the first encounter, and so include file guards are unnecessary.

```
using "file.h";                // Include file.h preserving prevailing utility
using/utility "string.h";      // Include string.h as utility declarations
using/pool "shared.h";         // Include shared.h as pooled declarations
```

The utility or pool attributes apply throughout the included file and its nested inclusions. The prevailing mode is restored after the include completes.

Declarations read while in utility mode provide information that enables FOG to correctly analyse and emit the wanted code, but do not directly cause emission of the utility code. However, utility classes may indirectly contribute to emitted code by providing derivation rules or meta-programs that do contribute to classes that are emitted. Any attempt to change the functionality of utility classes can be diagnosed and rejected.

The new include syntax is restricted to top-level declarations, unlike the `#include` directive that could potentially appear in the middle of an expression. The rationale behind this is a corollary of adopting an include-just-once policy to avoid the need for include file guards. This policy implies that a new-style include cannot be included by more than one scope and so should not be included by any; every included declaration should be part of the global name-space. Perhaps a little of the old behaviour should be restored by introducing `using/reinclude` as part of any declaration or statement.

5.5 Dependency Analysis

FOG declarations permit considerable freedom in their ordering and placement. C++ declarations are much more stringent. FOG must therefore find an appropriate form in which to emit the declarations to avoid C++ compilation errors. This requires establishing a legal order for declarations, a partitioning of those declarations into files and incorporation of appropriate forward references and

`#include` directives. While observing the constraints of a legal order, FOG groups similar declarations to improve readability, and normalises the output using an alphabetic sort to maximise the likelihood of an unchanged regeneration.

The legal order is established by a number of passes over the AST. These first establish file names and then build graphs to define the ordering constraints.

An implementation and interface file is identified for each declaration using an explicit specification or an algorithmic default. The default is determined by the enclosing scope, which at the top level associates distinct files with each class or namespace using the policy summarised in Table 5.1.

Usage nodes are defined for each form of usage of each declaration, and a usage dependency graph is built between these usage nodes by traversing each declaration and identifying the declarations upon which it depends. This is most easily understood from an example.

```
class A {};
class B {};
class F;
class C
{
    static A a = 0;
    B b;
    !inline F *f() { return 0; }
};
// class A : public C {};
```

The corresponding usage dependency graph is shown in Figure 5.2, in which solid arcs denote a directed dependency of the node at the start upon the node at the finish. Dashed arcs denote redundant dependencies. The dotted arc is an extra dependency for the commented line at the end of the example.

Six different forms of usage node are used in the diagram:

C_{name}, a forward referenced name of `C` such as `class C;`

C_{head}, the start of the interface for `C`, such as `class C {`

C_{tail}, the end of the interface for `C`, such as `};`

C_{inline}, the start of the inline implementation of C: just a place-holder.

`C::f`$_{int}$, the interface of `C::f`, such as `F *f();`

`C::f`$_{imp}$, the implementation of `C::f`, such as `F *C::f() { return 0; }`

The interfaces of class members are bounded by the head and tail of their class. The external dependencies of the class member interfaces are promoted to the class, since it is illegal to resolve those dependencies within the class. Thus the dependency of `C::f`$_{int}$ upon F_{name} shown dashed is promoted to a dependency from C_{head}.

The differing dependency patterns of `C::a` and `C::b` upon `A` and `B` reflect the distinct dependency of static and non-static member variables. As a result, removing the comment from the final line in the example adds the extra dependency shown dotted, and does not form a loop, although a casual examination of the code might indicate that it does.

The usage dependency graph describes the entire sub-program visible to the meta-compilation session. This is generally smaller than the entire application, but much larger than the distinct subset of declarations used by each output file.

In the example, declarations of F remain unknown. When `C.hxx` is emitted, it must `#include B.hxx`, but need only forward reference A and F. If `A.hxx` is included, subsequent compilations of C will be burdened with unnecessary declarations and

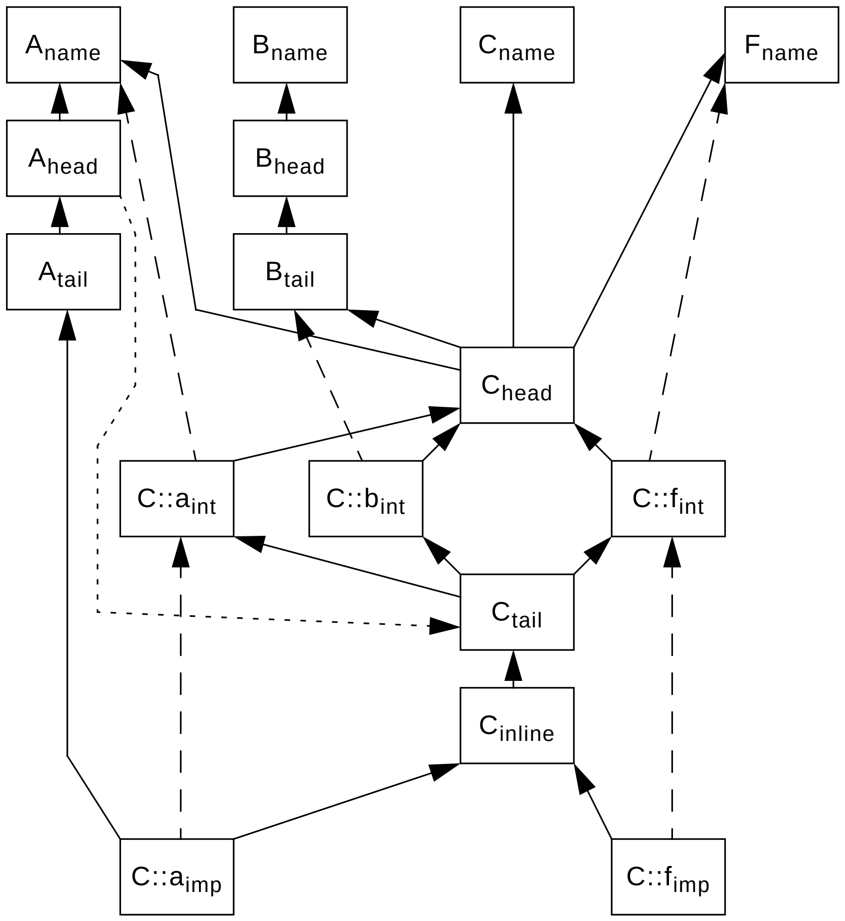

Figure 5.2 Usage Dependency Graph

dependencies. If `F.hxx` is included, a failure may occur since the declarations of F may be in a differently named file.

The declarations emitted in each file must be carefully organised, so that each emitted file is useable without reference to other files. Unnecessary declarations should be avoided. Necessary declarations should be referenced.

The usage graph contains all the necessary information. If appropriately organised, emission of any file just involves selecting all the corresponding usage nodes for emission, and ensuring that all their ancestors are resolved by `#include` directives or forward declarations. The passes over the AST and usage graphs to achieve this appropriate organisation are now described.

5.5.1 Create Usages

The create usages pass traverses the complete AST to create the usage nodes and dependency arcs.

Six different forms of usage were used on Figure 5.2, and show how even a simple program generates a complicated graph. Further forms are used to resolve `friend`s and explicit file dependencies.

The complexities arise from two sources

- the context-sensitive form of each usage in C++ (§3.3)
- the need to observe modularity constraints in the generated files

The C++ complexity requires careful coding of the `create_usages()` virtual function at the relevant AST nodes. Particular care and complexity arises for templates for which instantiations delegate dependencies to the primary template, but for which (partial) specializations compound dependencies. Namespaces, friends, comma-separated declarations, typedefs, inlined functions, default argument lists and anonymous unions provide their own minor complexities.

The modularity constraints arise because declarations cannot be freely ordered; forward referencing constraints must be observed; class declarations must occur together without any interleaving. Grouping of declarations in files must be possible, although this may sometimes necessitate some declarations preceding, and others following a `#include`. Files may therefore appear to overlap. Most of these modularity constraints can be enforced by adding additional dependencies to the usage graph. Thus in the earlier example member dependencies were promoted to the class head to ensure that they were satisfied before class emission started and so ensure that class emission could proceed without interleaving. The remaining modularity constraints are observed, if possible, by careful ordering of usages and files. It is possible for a programmer to over-specify the placement of declarations and so define an insoluble problem. In this case the problem is diagnosed and then approximated.

5.5.2 Usage ordering

The usage graph is traversed to initialise the following worklists

- an initially empty ordered output list
- 25 lists of nodes with no remaining dependencies
- an input list of nodes with dependencies

25 lists are maintained so that each different style of declaration is in a separate list. Thus public/protected/private, static/non-static and function/variable/typedef distinctions are each associated with different lists. The lists are prioritised and as a result the ordering algorithm automatically groups similar declarations. This improves readability and repeatability. Most of the priorities are chosen for aesthetic properties of the output files. However the tail of a scope is given very high priority and the head of a scope given very low priority to avoid overlapped scope declarations. When multiple classes are eligible for selection, selection of the head of one at low priority makes the contents of the class eligible with higher priority, and if the dependency graph has been built correctly, ensures that the contents are emitted and the class closed before one of the other eligible classes is reconsidered.

The worklist ordering algorithm repeatedly chooses the highest priority non-empty dependency-free list, ensures it is in alphabetical order and moves the first entry to the ordered output list. All dependencies on that node are removed with the result that some nodes may be moved from the final list to the dependency-free lists. The algorithm has quadratic asymptotic complexity, but the behaviour is close to linear in practice.

The algorithm stalls if there is a dependency loop. In this case the loop is printed out in an error message and a dependency in the loop is arbitrarily discarded before the algorithm resumes. The asymptotic complexity under error conditions is quartic. This is highly pathological requiring dependency loops between all nodes.

5.5.3 Usage closure

Once the legal order has been established, the transitive closure of the dependencies of each usage node is determined by propagating the set of ancestors to each child. Each usage node now has a set of precursors: declarations that must be visible before the usage node is emitted. This trades off the one-off execution time of this propagation and the memory costs for each set against the cost of repeated traversals of ancestors in later algorithms.

5.5.4 File usage ordering

A similar worklist algorithm is used to establish the ordering of potential `#include` directives, after projecting the usage node dependency on to the much smaller number of file nodes to create a file node dependency graph. Only one list of dependency-free nodes is used.

5.5.5 File pre-ordering

In order to emit dependencies efficiently. it is necessary to know what usage nodes are visible after a file has been included. This information is gathered in two passes.

The first pass is a simple initialisation of all files to identify the usages visible within the file but excluding those visible from nested includes.

5.5.6 File post-ordering

The second pass is performed using a traversal of all files ordered according to their earliest last (most dependent) usage node. This order guarantees that nested include files are processed before less nested includes. In this pass all usage nodes visible in included files are added to those visible in the including files.

5.5.7 Emission

The final emission pass creates the output file from the ordered usage nodes for that file. Code generation is relatively straightforward since each usage node closely corresponds to a partial declaration. Linear traversal of the usage nodes generates the required output.

In order to ensure compact and correct resolution of references, the set of currently visible declarations is maintained during emission. As each usage node is considered for output, its necessary precursors are checked. If any of these is not visible, appropriate forward references or `#include` directives are emitted, with more highly dependent include files favoured to maximise the rate at which necessary declarations become visible. Eventually the usage node can be emitted, but only after the appropriate scope has been established. Namespace scopes are established lazily to avoid repetition of the same scoping construct, since eager establishment of scope would fail to suppress the commented lines in:

```
    extern "C"
    {
        declaration-1;
//  };
//  extern "C"
//  {
        declaration-2;
    };
```

The potential output file is generated in memory so that it can be compared with any pre-existing file. An unchanged file is not overwritten.

5.6 Target File Generation Policies

5.6.1 Global namespace

Class and namespace declarations are normally written to files based upon the declaration name. Unfortunately, the global namespace has no name and creation of name-less files such as `.hxx` and `.cxx` would cause confusion. The global namespace contains many utility declarations from C libraries which do not need repetition in another file. The global namespace contains few declarations from application code, since C++ encourages the use of encapsulation to avoid polluting the global name-space. The default behaviour of FOG is therefore to ignore global namespace declarations. A command line option permits a name to be specified for the global name-space. If this option is used, the global namespace behaves (for output file naming purposes) as if it were a namespace with the specified name.

5.6.2 Friend functions

A significant exception to the above discussion occurs for friend functions. Such functions although logically part of some class are given namespace scope to avoid biasing overload resolution to the left argument

```
    friend MyClass operator+(const MyClass&, const MyClass&);
```

or to use some other class as the first argument

```
    friend ostream& operator<<(ostream&, const MyClass&);
```

Since these are (global) namespace scoped, the implementation would be emitted to the namespace implementation, which is unlikely to be what is required. The implementation of a namespace-scoped function is therefore placed with the implementation of the class that declares it a friend, provided there is exactly one such class.

5.6.3 Source File Protection

The output from FOG comprises C++ source files that may not differ significantly in name or path from files input to FOG. There is ample scope for accidental overwriting of input files by output files, either through ill-considered user commands, or through coding errors in FOG.

FOG protects against such errors by starting every generated file with the character sequence `/*!$@FOG@$!` and only allowing overwrites of files with this signature. If the user really wants to give FOG complete freedom to overwrite anything, then the `-f` command line option must be used.

5.6.4 Suppressed Non-changes

FOG may generate many output files from a single invocation, normalising each file to minimise changes. It may often be the case that some generated files are unchanged and so subsequent dependent recompilations are unnecessary. FOG therefore compares each potential output file with any pre-existing file, and if the potential output file differs only in whitespace (or comments), the existing file is retained, avoiding any change of creation date. A make script can then skip unnecessary compilations.

The operation of this algorithm may be observed using command line options: -nc notifies file changes and -ne notifies preservation of existing files. The policy may be overridden by using `-f` to force all files to be created.

5.6.5 Net dependencies

In order for a makefile to determine whether FOG needs to be executed, the makefile needs to know the dependencies of output files and input files. The `-o` command line option may be used to emit a make include file that defines the dependencies as the dependants of that include file. All other generated files can then be reliably created by side effects.

5.6.6 Pretty Printing

The files generated by FOG are readable, unlike files generated by code generators such as cfront. The generated text differs little from what could have been typed manually.

In the multi-pass implementation, function bodies were emitted directly from the almost unparsed phase 6 token stream. This token stream retained an indication of the original source whitespace, and so the generated output could reproduce much of the user layout. In order to achieve consistent indentation, source text indentation was normalised to the first non-whitespace character in a function-body, and then denormalised after pretty-printing had determined the position of the first character.

Source text normalisation required tab characters to be converted to spaces. The default FOG behaviour assumed the traditional 8 column tab spacing. The increasingly common configuration of 4 column tabs could be specified by adding `-t4` to the command line.

The superset implementation parses function bodies into syntax trees and so loses original whitespace. The entire output is therefore pretty printed without reference to the input.

5.6.7 `#line`

Translators to C or C++ may incorporate `#line` directives in their generated files, so that compiler diagnostics and debugger single stepping refer to the original source context, rather than the intermediate generated files. This is generally very helpful to the programmer, but when the translation process is unreliable or when obscure problems arise, access to the hidden intermediate is more useful.

The default behaviour of FOG is to incorporate `#line` directives in its output. The `#line` directives may be replaced by comments by using the `-comment_line_numbers` command line option, or suppressed altogether by `-no_line_numbers`.

Use of `#line` directives has the unfortunate consequence that almost any change to a source file causes regeneration of most output files, if only to update line number information.

Total suppression of line numbers produces the most readable intermediate, but gives no clue as to the origin of each source code segment.

Preservation of the line numbers as comments avoids the regeneration problem, while retaining traceability. Operation at this level may often be appropriate, since single stepping a misbehaving function composed from many contributions is probably best performed when the function body appears in its composed form; strange compilation errors are also more easily diagnosed when the full context is visible. However, the line numbers are in comments, and intermediate files are not regenerated to update comments, and so commented line numbers may prove to be seriously adrift.

Ideally a debugger would support more than one line number domain. In practice users may have to choose which domain best suits the prevailing needs.

5.6.8 Integrity

There is a danger that the composite declarations in different meta-compilation sessions may be incompatible. This danger may be compounded by command lines requesting more than one meta-compilation session to generate the same output files. This may cause subsequent compilations to be inconsistent. If the user is fortunate, the inconsistencies will lead to helpful compilation errors. More likely there will be confusing linker diagnostics or worse still, run-time failures. The latter problems are violations of the C++ One Definition Rule (§3.2) which the standard allows an implementation to leave undetected; it is a user programming error. In C++, use of shared include files makes this form of mistake relatively rare, although over-enthusiastic incremental compilers can easily trigger such errors.

The increased level of abstraction and greater opportunity for configuration problems in FOG justifies extra effort to assist the programmer. This is an area for future research. Two possible solutions are outlined:

5.6.8.1 Checksum

Incorporation of a cyclic redundancy check (CRC) hash code as part of the name of a static member variable in every FOG generated class solves the integrity problem. The member variable is initialised by a meaningless calculation that causes the linker to reference the equivalent names in all dependent classes. Thus if class A uses classes B and C, the consistency checking code for class A could be:

```
class A
{
public:
    static char _c_r_c_A43F1507; // A43F1507 is CRC of rest of class
};
char A::_c_r_c_A43F1507 = char(&B::_c_r_c_5670BD33
                            + &C::_c_r_c_EE8241C5);
```

The definition originates from the defining session. The references are independently calculated by referencing sessions. If the two sessions are using inconsistent definitions, the problem will eventually show up as a requirement for the linker to resolve two different symbols for the one class. This will fail and prevent run-time anomalies, subject to the CRC algorithm producing highly decorrelated hash codes.

When class A uses class B is a matter for further research and careful definition.

A safe definition could follow the normal C++ usage definition (§3.2-4). This is unnecessarily strong, since meta-compilation does not establish class layout, and so the usage of types during meta-compilation is generally by name rather than by value.

A simpler but not quite safe definition defines the usage as all classes generated by the same meta-compilation session. This is unsafe because derivation rules or meta-programs may have used declarations from base classes.

It would seem that each meta-compilation session should keep track dynamically of each type used in more than name, and that the references in the linker expression should involve all such types.

This approach incurs a one byte per class penalty, which could be eliminated if it were possible to access the linker symbol calculations directly. A zero cost implementation could use the generated symbol name as the name of the virtual function table.

5.6.8.2 Database

The problem of detecting conflicting or redundant generation of an output file by more than one meta-compilation session is less easy to detect early. The checksum approach will detect a conflict late in the build process, and is unlikely to produce a helpful diagnostic. A rebuild using -nc should not result in notification of any file creations so this at least provides a way to investigate the conflicts.

Direct diagnosis of such conflicts, or any diagnosis of redundant but compatible generation from multiple meta-compilations requires a database file to map output files to meta-compilation sessions. Duplicate mappings are errors.

The coordinating database can easily be specified as a command line parameter, and new or repeated meta-compilations will update the database. However meta-compilation sessions do not have a reliable identity, so it is not possible to detect whether a slightly changed meta-compilation command line represents a new or replacement session. It is certainly not possible to detect obsolete sessions. This problem is most easily resolved by manual or automatic deletion of the database whenever make scripts are changed.

This problem and solution is closely related to the template instantiation database necessary to satisfy the C++ requirement for instantiation of precisely the used functionality. The template database is necessary for language compliance and an aid to compiler efficiency. A similar or combined FOG database is just a diagnostic aid.

6 Examples

This chapter provides example uses of FOG, starting with very simple idioms and progressing to more serious applications.

6.1 Idioms

Although some of the simple idioms are little more than one-liners, they should not be dismissed as trivial programs. They demonstrate how real programming problems can have very simple solutions in FOG. The examples support direct expression and realisation of programming intent and consequently improve reliability and maintainability, trading off some declarative complexity against ease of instantiation.

6.1.1 InheritedTypedef

Using inheritance, a programmer may derive a specialisation from a base class. This may involve a refined implementation of a base class operation. In languages such as BETA [Madsen93], the derived implementation is constrained by the inner invocation from the base class. C++ imposes few limitations; the programmer is free to provide a replacement implementation, or to incorporate the derived functionality where appropriate. Incorporation of derived functionality is conventionally provided by invoking the base-class method explicitly.

```
void DerivedClass::method()
{
    // some code
    BaseClass::method();
    // more code
}
```

When `DerivedClass` is derived directly from `BaseClass`

```
class DerivedClass : public BaseClass { /* ... */ };
```

the code fragment operates as might be expected.

However, if `DerivedClass` is indirectly derived

```
class IntermediateClass : public BaseClass { /* ... */ };
class DerivedClass : public IntermediateClass { /* ... */ };
```

experienced programmers would regard the code fragment as at best exhibiting poor style, because the problem exists as to what the meaning of `IntermediateClass::method()` is.

It may be that when the fragment was first written, `DerivedClass` was directly derived, and there was no `IntermediateClass` method to worry about. When subsequent maintenance or evolution introduced `IntermediateClass` and `IntermediateClass::method()`, a bug was also introduced.

If `IntermediateClass::method()` is accidentally bypassed, then the use of the `BaseClass` name may lead to incorrect behaviour.

If `IntermediateClass::method()` is deliberately bypassed, then the code is obscure, since in some respect `DerivedClass` is not a specialization of `IntermediateClass`, or `IntermediateClass` is not a specialization of `BaseClass`. Such obscurity probably hides a design flaw, and certainly merits a comment to explain why the `IntermediateClass::method()` has been bypassed.

Summarising: The use of `BaseClass` is a stylised use of a name that fails to express its intent. This is very similar to the use of a magic number such as 7 to denote the number of days in a week. Experienced C programmers have learnt to use mnemonic definitions to make the intent clear.

```
#define DAYS_IN_WEEK 7
```

```
for (int i = 0; i <= DAYS_IN_WEEK-1; i++)
```

Stroustrup on page 292 of D&E [Stroustrup94] describes discussions for a potential language extension to resolve this problem, and credits Michael Tiemann for providing the following very simple solution:

```
class foreman : public employee {
    typedef employee inherited;
    //...
    void print();
};
class manager : public foreman {            //1a
    typedef foreman inherited;              //1b
    //...
    void print();
};
void manager::print()
{
    inherited::print();
    //...
}
```

The base class is referred to as `inherited` throughout the class, limiting the knowledge of the inheritance hierarchy to lines //1a and //1b. The reader is assured that the intent is to invoke the base class functionality, and may then question any reference to class names other than `inherited`. The maintainer does not risk breaking code when the inheritance hierarchy is reviewed.

FOG offers three alternative solutions that avoid the redundancy of line //1b, and consequently avoid the risk of an inconsistency between line //1a and //1b.

Super meta-variable

FOG provides a built-in meta-variable `Super` that refers to the first base class. There is therefore no need for a typedef. The built-in meta-variable can be used instead.

```
void manager::print()
{
    ${Super}::print();
    //...
}
```

This is not a very elegant solution. It requires a $-expression to appear in normal application code.

Super typedef

The typedef can be reintroduced to hide the $-expression from application code.

```
class manager : public foreman {
    typedef $Super inherited;
```

This clearly expresses the intent and is guaranteed to be locally correct. There is however a possibility that the typedef could be omitted for a derived class, and if the typedef is accessible no compilation error would arise.

InheritedTypedef meta-function

The problem of an inaccurate definition in a derived class, and the inconvenience of providing the definition in every class can be resolved by a derivation rule, and since the final solution is of general utility it is presented as a meta-function.

```
auto declaration InheritedTypedef()
{
    private typedef @Super inherited :{ derived(!is_root()); };
}
```

Expression as a meta-function introduces three changes to the typedef declaration:

A private keyword is added to ensure that the typedef is not visible outside the class. Omission of the keyword would leave the declaration vulnerable to assuming a prevailing accessibility from the invocation context.

The derived(!is_root()) is added to specify the derivation rule, as a result of which the declaration is generated in all classes that derive from the class for which the meta-function is invoked. Thus when invoked as

```
class employee
{
    $InheritedTypedef();
};
```

no typedef is generated at the root (employee) for which Super may be undefined, but typedefs are generated for all classes derived from employee.

The use of @Super rather than $Super defers resolution of the meta-expression until each actual declaration generated as a result of the derivation rule is installed in the derived class. @Super therefore resolves to the super-class of the actual derived class. $Super would be resolved when the potential declaration is defined for the root class, and would result in the typedef always resolving to the super-class of the root class.

Benefit

Use of the meta-function requires a single line of source code for each base class, and ensures consistency throughout an inheritance hierarchy.

The conventional implementation requires one line of source code per derived class in the inheritance hierarchy, and is vulnerable to typographic errors.

Alternatives

This problem cannot be solved reliably without access to the name of the base-class. The Tiemann approach localises the redundant declarations to a single place, potentially adjacent to the necessary declaration.

Applications that make extensive use of preprocessor macros to define class scaffolding can implement the typedef with no additional application code:

```
class DerivedClass : public BaseClass
{
    SCAFFOLDING_INTERFACE(DerivedClass, BaseClass)
    //...
};
SCAFFOLDING_IMPLEMENTATION(DerivedClass)
```

The typedef can be incorporated in the SCAFFOLDING_INTERFACE preprocessor macro. The examples in Sections 6.3.3 and 6.5 show how other standard declarations that may form part of SCAFFOLDING_INTERFACE can also be resolved automatically, rendering both of the SCAFFOLDING_IMPLEMENTATION and SCAFFOLDING_INTERFACE macros obsolete.

6.1.2 NoAssign, NoCopy

C++ provides powerful facilities to support the definition of encapsulated data types. C++ also eases the definition of user defined types by providing default implementation for constructors, destructors and assignment operators. For classes involving pointers or allocated resources, the default implementations are inappropriate; replacement implementations must be provided.

For some classes there is no possible replacement. Consider a class that defines properties of colours, with a single unique object for each used colour combination. Creating a copy of such an object is meaningless because that would break the uniqueness property. Assigning to such an object is illegal since assignment would involve creation of a new combination, and so could not be represented by the old object.

For other classes, the replacement implementation might never be used and so the development cost and code size of the replacement cannot be justified.

These problems are solved in C++ by declaring the relevant methods to be private and not providing an implementation for them [Coplien92] p45.

```
class UniqueColor
{
private:
    UniqueColor(const UniqueColor&);            // not implemented
    UniqueColor& operator=(const UniqueColor&);// not implemented
};
```

Any attempt to copy or assign to a `UniqueColor` object outside of the scope of `UniqueColor` encounters a compilation error though the `private` access restriction. Within the scope of `UniqueColor`, compilation succeeds, but a linker error results from the missing implementations.

This is a well-known idiom, but is obscure and so not always recognised by the novice programmer. An accidental implementation of the not implemented functions may break the informal coding convention.

Meta-functions may be provided in FOG to express the intent more clearly and to enforce the non-implementation constraint.

```
auto declaration NoAssign()
{
    private $Scope& operator=(const $Scope&)
            :{ export/noimplementation; };
};
auto declaration NoCopy()
{
    private ${Scope}(const $Scope&) :{ export/noimplementation; };
};
```

The meta-functions may be invoked as

```
    class UniqueColor
    {
        $NoCopy();
        $NoAssign();
    };
```

clearly expressing the programming intent.

The implementation of each meta-function makes extensive use of $Scope to define declarations appropriate to the invoking scope. The `:{}` declares an *object-statements-clause*, within which function annotations can occur. In this case the function scope contains the single declaration `export/noimplementation`, which ensures that FOG generates a compilation error if any attempt is made to compose an implementation.

Alternatives

NOCOPY and NOASSIGN preprocessor macros could be defined, however the name of the surrounding scope would have to be passed as a parameter.

```
class UniqueColor
{
    NOCOPY(UniqueColor);
    NOASSIGN(UniqueColor);
};
```

Enforcement of no-implementation requires the additional language support provided by FOG.

6.1.3 Mutate

ANSI C introduced the `const` qualifier to types to define unchanging values. C++ extends `const` to apply to objects and consequently member functions. Use of `const` ensures that any attempt to change an object is detected at compile time.

However, it may be appropriate for the implementation of an object to perform lazy evaluation of some of its properties, caching the results to avoid a re-evaluation. The conventional `complex` class has a Cartesian representation, and must therefore calculate a polar representation each time the polar representation is requested. A more sophisticated complex number class could cache the polar representation lazily, so that no calculation cost was incurred when the polar representation was unused, and avoid additional calculation cost for uses after the first. From an external perspective, the complex number object is unchanged by the caching of a polar representation and so the usage method should be `const`. Internally the cached context changes and so the object is not `const`.

This distinction is referred to as physically-const and logically-const on p26 of [Coplien95b]. On p76, [Meyers92] uses the term conceptual constness.

Implementation of logically-const code requires that the `const` qualifier be cast away.

```
((ComplexNumber *)this)->_polar_value = ...
```

Usage of casts in application code is considered poor style. They are prone to error; any kind of change can be performed, although only a very subtle change was intended. When reviewing code it is difficult to locate casts with searching tools, and it is not always obvious what the intent of the cast is.

A simple meta-function can resolve the problem of visibility and intent.

```
auto declaration Mutate()
{
    private $Scope& mutate() const { return *($Scope *)this; }
}
```

The casting function may be installed in a class by

```
class ComplexNumber
{
    $Mutate();
};
```

allowing subsequent usage as

```
mutate()._polar_value = ...
```

Alternatives

The introduction of the `mutable` *storage-class-specifier* resolves many of the problems of logically-const `const`. In the above example, declaring `_polar_value` as `mutable` would be a complete solution. However `mutable` is a *storage-class-specifier* and not a *cv-qualifier* and so non-const methods cannot be invoked without some form of cast.

```
mutate().set_polar_value(...);
```

The introduction of const_cast<T> resolves the danger of inadvertently casting to a different class, since the type of T can be statically checked. However

```
const_cast<ComplexNumber>(*this)._polar_value = ...
```

is a little harder to read, contains a redundant typename, and does not work on old compilers. The improved functionality of const_cast can be exploited to give a more robust meta-function:

```
auto declaration Mutate()
{
    private $Scope& mutate() const
        { return const_cast<$Scope>(*this); }
}
```

6.1.4 Clone, Prototype

In languages such as Smalltalk, creating a copy of an object presents no problem, since there is direct language support. In C++, creating a copy of an object whose type is known at compile-time makes use of the copy constructor. However when the type is not statically known, the programmer must provide support code.

Stroustrup on p424 of [Stroustrup97] refers to this support code as a virtual constructor. The technique is also referred to as cloning and forms part of the Prototype pattern [Gamma95].

The support code requires that every concrete[1] class implements a virtual function to create a clone of itself.

```
class RootClass
{   //...
public:
    virtual RootClass *clone() const = 0;
};
class IntermediateClass : public RootClass
{   //...
};
class ConcreteClass : public IntermediateClass
{   //...
public:
    virtual RootClass *clone() const;
};
RootClass *ConcreteClass::clone() const
{
    return new ConcreteClass(*this);
}
```

Invocation of the virtual function upon an object of unknown type therefore invokes the appropriate class-specific method to create the clone.

```
const RootClass& someObject = ...;
RootClass *clonedObject = someObject.clone();
```

Implementation of this idiom requires contributions to the interface and to the implementation of each concrete class. These will often be in different files, because of the need to avoid excess include file dependencies. Observance of the protocol is largely enforced by the use of a pure virtual function. However an inaccurate implementation can arise through failing to create an instance of the

1. a class with no pure virtual methods

correct class, or though failing to implement `clone` when one concrete class inherits from another concrete class.

A FOG meta-function using a derivation rule can generate all derived class code automatically

```
auto declaration Prototype()
{
    public virtual $Scope[2] *clone() const = 0
    :{
        derived(!is_pure()) { return new @{Scope}(*this); };
    };
}
```

The `clone` functionality is woven by a single invocation from the root class.

```
class RootClass
{
    $Prototype();
};
```

The pure virtual function is defined in the invoking class. The function body has an associated derivation rule requiring implementation in all concrete classes. The function body, and consequently its declaration, is therefore generated in each concrete class. The use of `@{Scope}` within the function body ensures that resolution of the name is deferred until the function body is installed in its actual class, and so ensures that a new instance of the concrete class is created.

It is important to use `derived(!is_pure())` rather than a less restrictive rule such as `derived(true)` to avoid generation of code that creates instances of abstract classes, since compilers are required to generate error messages if an abstract class is constructed.

Alternatives

This problem is insoluble without an automatic code generator. Traditional approaches require extensive use of scaffolding macros.

6.2 Patterns

The general problem of providing implementations to patterns is not soluble, because patterns are too vague and require tailoring to suit the application context. However, implementations of patterns suitable for more restrictive contexts are possible. A few such implementations are presented in the following sections.

6.2.1 StaticFlyweight

Components are easier to use when components that exhibit similar functionality provide it in a predictable way. This can be achieved by providing an isomorphic external interface for a standardised internal behaviour, often realised by the use of virtual functions to provide polymorphic behaviour at run-time. The concept of an isomorphic interface at compile-time is not normally used in C++, but lies at the heart of the Standard Template Library, where many of the templates operate on any type that complies with the defined isomorphic interface. The concept of isomorphism is independent of polymorphism and inheritance: the templates work for a variety of independent inheritance trees; there is no need for a common base class.

2. `$Scope` may be changed to `@Scope` to use the derived type as the return type, which is permitted in standard C++ but not allowed in earlier implementations of C++.

Provision of meta-functions and meta-variables makes the advantages of isomorphism more visible to the programmer. Families of isomorphic meta-functions can be declared, one per class, to provide the same functionality for the programmer, but using a distinct implementation appropriate to each class.

The Flyweight pattern [Gamma95] describes how shared objects can be used to reduce allocation costs. The pattern comprises a Flyweight manager responsible for managing a pool of Flyweight objects. The manager creates a new Flyweight, or provides access to an existing one, in response to a request from the client. In the general implementation, the pool may be large and so the factory manager needs some form of map to locate the Flyweights. A useful variation occurs when the Flyweight objects can be enumerated at compile-time and may have the same life-time as the program. The manager then degenerates to a set of static member functions one per flyweight.

```
class DayOfWeek // ...
{
public:
    static const String& monday()
        { static const String theDay("Mon"); return theDay; }
    static const String& tuesday()
        { static const String theDay("Tue"); return theDay; }
    // ...
};
```

The `DayOfWeek` class makes the textual representation of each day available as flyweight objects, using function scope to define the object lifetime and thereby avoid the race conditions during object construction that can arise from the use of static member variables.

Using a family of isomorphic meta-functions to capture this variant of the Flyweight pattern avoids the need for an application programmer to understand the construction protocols of the relevant flyweight.

```
class String
{   /* ... */
    auto declaration StaticFlyweight(identifier name,
                                     string init = "")
    {
        public static const String& ${name}()
        {
            static const String staticInstance($init);
            return staticInstance;
        }
    }
};
```

The `DayOfWeek` class may then be simplified to

```
class DayOfWeek // ...
{
    $String::StaticFlyweight(monday, "Mon");
    $String::StaticFlyweight(tuesday, "Tue");
    // ...
};
```

The programming complexity is now partitioned appropriately. Provision of a flyweight requires just a single line. The two to five lines to implement flyweight construction appear just once as part of the flyweight class, rather than repeated throughout each static flyweight manager. Since the construction protocol for the flyweight is encapsulated by the meta-function, different protocols may be used, without affecting the callers, except in so far as additional initialisation arguments might be required. For instance, if an implementation of `String` makes use of the Flyweight pattern to share all identical strings across an application, the above code will not work. The `String` class must define private constructors and/or destructors to prohibit static instances. The following may be necessary:

```
auto declaration String::StaticFlyweight(identifier name,
                                          string init = "")
{
    public static const String& ${name}()
    {
        static const StringHandle staticInstance($init);
        return *staticInstance;
    }
}
```

where `StringHandle` is a smart pointer to a `String`.

Alternatives

It is possible to use the preprocessor to solve this problem

```
#define STRING_STATIC_FLYWEIGHT(name, init) \
    static const String& name() \
    { \
        static const String staticInstance(init); \
        return staticInstance; \
    }
```

However the preprocessor approach is unnatural and so programmers tend to avoid its use. Meta-functions fit within the context of the language and so provide a useful addition to a programmer’s tool-box.

This kind of problem can sometimes be solved using templates. However in this case the initializer is a string which is not a legal template parameter, so in the following example the initializer is changed to a character to produce legal code.

```
class String
{   /* ... */
public:
    template <const char initString>
    class StaticFlyweight
    {
    public:
        const String& operator()() const
            { static const String iT(initString); return iT; }
    };
};
class DayOfWeek
{   /*...*/
public:
    static const String::StaticFlyweight<'M'> monday;
    static const String::StaticFlyweight<'T'> tuesday;
};
```

The initializer is cached as the template parameter so that access of the flyweight as

```
DayOfWeek::monday()
```

invokes

```
String::StaticFlyweight<'M'>::operator()()
```

which maintains the appropriate flyweight instance. The above code compiles, but fails to link. It is unfortunately necessary to duplicate the declarations of the flyweights in the interface by defining implementations of the functional objects, even though they are never used and have no content.

```
    const String::StaticFlyweight<'M'> DayOfWeek::monday;
    const String::StaticFlyweight<'T'> DayOfWeek::tuesday;
```

This example shows many of the limitations of templates:

- limited validity of parameter types
- failure to achieve one line invocation
- relatively obscure implementation

Note that use of `operator()()` is the only way in which instantiation of a template can result in a user-defined function name. Since it is necessary to provide a corresponding definition, this approach requires two declarations for a user-defined name, just the same as for renaming with an inline function or a typedef.

When more than a single declaration is required, the template approach fails, although the preprocessor approach may remain viable.

6.2.2 Member

It is customary to declare member variables and associated support declarations individually, typically involving

- member declaration
- initialization
- get methods

and sometimes

- set methods

in addition to any actual application functionality. Any intended mode of behaviour of the member is left to be inferred from the miscellaneous declarations.

For variables with primitive types, coding errors can easily arise from missing initializers.

For variables with more complicated types, the associated declarations may be less obvious.

The standard declarations are easily provided by meta-functions, whose name demonstrates the programming intent, and whose use offers opportunities for uniform code evolution along the conventional development path, or for meta-programming to compose additional functionality.

The three declarations required to define a member variable (*name*), a protected set accessor (`set_`*name*) and a public get accessor (`get_`*name*) are provided by:

```
    auto declaration ScalarMember(identifier type,
                            identifier name, expression init = 0)
    {
        private $type $name = $init;
        protected void set_${name}(const $type& aValue)
            { $name = aValue; }
        public const $type& get_${name}() const { return $name; }
    };
```

A similar isomorphic meta-function can be provided to cover a smart pointer:

```
    auto declaration SmartPointerMember(identifier type,
                        identifier name, expression init = 0)
    {
        private SmartPointer<$type> $name = $init;
        protected void set_${name}(const $type& aValue)
            { $name = aValue; }
        public const $type& get_${name}() const { return *$name; }
    };
```

Member variables and their standard accessors may then be defined as

```
class Application
{   /* ... */
    $ScalarMember(bool, is_valid, false);
    $SmartPointerMember(Client, client);
};
```

The example uses multiple parameters to pass the type, name and initial values. A complete declaration may be passed as

```
$ScalarMember(bool is_valid = false);
```

at the expense of a little extra declaration effort:

```
auto declaration ScalarMember(variable_specifier var)
{
    private $var.type() $var.name() = $var.value();
    protected void set_${var.name()}(const $var.type()& aValue)
        { $var.name() = aValue; }
    public const $var.type()& get_${var.name()}() const
        { return $var.name(); }
};
```

Alternatives

Much of this functionality can be provided by preprocessor macros, however FOG extensions are required to support member or static member initialisation from a single invocation.

Default initialisation can be enforced by a templated member such as

```
Initialized<bool> _is_valid(false);
```

and `operator()()` can be used to support a sensibly named get method. However, further flexibility cannot be provided by templates.

6.2.3 ReferenceCount[3]

C++ has no garbage collector. It is therefore the C++ application programmer's responsibility to ensure that allocated memory is appropriately released. For simple forms of object use, allocated memory can be freed later in the same function. For slightly more complicated situations, objects may be organised in trees, with the parent objects assuming responsibility for releasing the resources of children; the application programmer's responsibility is then only with the roots of the trees. In the general case, it is too difficult, inconvenient or even impossible for the program structure to ensure that resources are released. In the absence of a garbage collector, the problem is resolved by making each object responsible for releasing its own resources. Users of the object register their usage with the object. When no users are registered, the object detects that it is no longer required and self-destructs, releasing the redundant resources. It is not necessary for the object to know the identities of its users, merely their number. This form of resource management is therefore implemented by reference counting: maintaining a count of the number of registered users [Coplien92].

Implementation of reference counting involves two collaborators. The reference counted object maintains a count of its users. Each user registers its usage, often using a smart pointer, construction and destruction of which increment and decrement the reference count.

The reference count may be intrusive or non-intrusive.

3. A version of this example that was tested using the multi-pass implementation of FOG was presented at TOOLS Eastern Europe [Willink99a]. The version presented here is an untested revision using the improved syntax for derivation rules and clarifying the heap/non-heap allocation policies.

An intrusive reference count adds an extra member variable to the reference counted object. It

- increases object size
- must be part of the class declaration
- is efficient

A non-intrusive reference count, maintains the count in a disjoint area of memory, possibly accessed by an associative look-up from the object address, possibly at a location known to the user. It

- need not be part of the class declaration
- is inefficient in space and/or time

The example here supports injection of the intrusive functionality. An isomorphic meta-function could be defined for the non-intrusive behaviour.

External Interface

In order to support reference counting, we need an application class to have an external interface such as

```
class ApplicationClass
{
protected:
    virtual ~ApplicationClass();
public:
    void annul() const;
    void share() const;
    friend inline void annul(const ApplicationClass *anObject)
        { if (anObject) anObject->annul(); }
};
```

`share()` and `annul()` increment and decrement the reference count, with `annul()` provoking self-destruction once all references have been removed. Since the object manages its own destruction, it is important to prevent deletion by any other mechanism. In particular

```
ApplicationClass *countedObject = ...;
delete countedObject;
```

could cause premature deletion. Use of delete should therefore be trapped as a compile-time error whenever possible, with a run-time trap as well to double check. Making the destructor protected ensures a compile-time error.

`annul()` acts as a replacement for `delete` and so the friend function is provided to create a closer analogue for `delete` as

```
annul(countedObject);
```

(An exact analogue could be provided using a syntax macro, but at the expense of defining `annul` as a reserved word.)

Internal implementation

The internal representation involves a counter, which is constructed with value 1, so that decrementing to 0 triggers self-destruction. Care is required to ensure that copy construction and assignment preserve registrations. The copy constructor therefore also defines the count to one. Assignment leaves the counts unchanged. It is convenient to encapsulate this behaviour in a class, so that the unconventional implementation of copy construction and assignment can be automatically incorporated into otherwise conventional implementations of the same functions in client classes.

```
class ReferenceCount
{
private:
    mutable unsigned int _shares;
public:
    ReferenceCount() : _shares(1) {}
    ReferenceCount(const ReferenceCount&) : _shares(1) {}
    ReferenceCount& operator=(const ReferenceCount&)
        { return *this; }
    ~ReferenceCount() { /* ASSERT(_shares == 1); */ }
    bool annul() const
        { return (_shares == 1) ? false : (_shares--, true); }[4]
    bool heap_only_annul() const { return --_shares != 0; }
    void share() const { _shares++; }
    unsigned int shares() const { return _shares; }
};
```

The destructor may beneficially validate that the share count is 1. This may detect a premature deletion through the use `delete`, or some more obscure problem such as double destruction through some unpleasant pointer recursion. The implementation of `share()` performs the increment for registration of an additional user. The implementation of `annul()` collaborates with the counted object to decrement when a usage is removed; the return status indicates whether the counted object should continue in existence. The count is not decremented if destruction is due, in order to ensure consistent count behaviour for the three following usage patterns

```
void test()
{
    static ApplicationClass staticallyAllocatedObject;
    ApplicationClass stackAllocatedObject;
    ApplicationClass *heapAllocatedObject = new ApplicationClass();
    // ...
    annul(heapAllocatedObject);
}
```

For the statically and stack allocated objects, construction and destruction bound the lifetime, any additional sharing through `annul` and `share` is optional. Only the statically allocated object may live on beyond the function return. For the heap allocated object the life-time of the object terminates with respect to the local function with the `annul`, but any additional registration in the commented section may prolong the lifetime. If only heap usage is required, the simpler implementation of `heap_only_annul` is adequate.

Glue code

Installing the `ReferenceCount` into an `ApplicationClass` class requires the following code to convert the partial functionality of the `ReferenceCount` class into the required external interface of the `ApplicationClass` class.

```
class ApplicationClass
{
private:
    ReferenceCount _shares;
protected:
    virtual ~ApplicationClass();
public:
    void annul() const
```

4. This rather contrived form of conditional ensures that there is only one return statement, which is a prerequisite for inlining by some compilers.

```
        { if (!_shares.annul()) delete (ApplicationClass *)this; }
    void share() const { _shares.share(); }
    friend inline void annul(const ApplicationClass *anObject)
        { if (anObject) anObject->annul(); }
    // ...
};
```

This is rather too much to be entered by the programmer. In the past, the author has used a preprocessor macro for all except the virtual destructor which has been dealt with manually. A meta-function provides a more powerful solution:

```
auto declaration ReferenceCount::install(bool heapOnly = false)
{
    if (!defined(has_reference_count))                      //1.1
    {                                                       //1.2
        auto bool has_reference_count = true;               //1.3
        private ReferenceCount _shares;                     //2.1
        public void share() const { _shares.share(); }      //2.2
        if ($heapOnly)                                      //2.3a
        {
            public void annul() const                       //2.4a
                { if (!_shares.heap_only_annul())           //2.5a
                    delete ($Scope *)this; }
            protected ~${Scope}() :{ derived(true) {} };    //2.6a
        }
        else                                                //2.3b
            public void annul() const                       //2.4b
                { if (!_shares.annul())                     //2.5b
                    delete ($Scope *)this; }
        friend inline void annul(const $Scope *anObject)    //2.7
            { if (anObject) anObject->annul(); }            //2.8
    }                                                       //1.4
}
```

The meta-function may be invoked as

```
    class ApplicationClass /* ... */
    {   /* ... */
        $ReferenceCount::install();
    };
```

leaving little opportunity for error or misunderstanding.

Lines 2.1 to 2.8 of the meta-function closely follow the glue code above, save for the use of per-declaration rather than prefix *access-specifier*s and for the use of $Scope to access the invoking class name.

The meta-function contains an outer conditionalisation on lines 1.1 to 1.4. to guard against double installation. When first invoked, the `has_reference_count` meta-variable is not defined, and so the conditional succeeds. The meta-variable is then defined on line 1.3, and any reinvocation of the meta-function within the `ApplicationClass` class or its derived classes is suppressed by the conditional.

The guard code is followed by the definition of the member variable (2.1), two member functions (2.2, 2.4) and a friend function (2.7). The _shares member variable provides the run-time storage needed by the share count, and interacts via its constructors and destructors with the equivalent functions for the reference counted class. The `share()` member function simply delegates the member variable. The `annul()` member function completes the share counting protocol by deleting the reference counted object when the final share is removed. The `annul()` friend function just provides a more convenient destruction option avoiding the need to worry about null pointers.

The inner conditionalisation (2.3) upon the formal parameter `heapOnly` selects between declarations that inhibit non-heap object construction, and the more flexible default behaviour allowing static as well as heap objects.

Line 2.6a defines the destructor to be `protected`, in the invoking class and as a result of the derivation rule, in all derived classes. This provides a compile-time check to catch most attempts to create counted objects statically or on the stack. The `virtual` keyword is omitted to avoid prejudicing the class designer's decision on whether the destructor should be virtual.

Accidental double installation within the same class may seem unlikely for conventionally structured code, however when more than one meta-function is invoked as in

```
class ApplicationClass /* ... */
{   /* ... */
    $NonIntrusiveList::install();
    $NonIntrusiveMap::install();
};
```

a multiple installation could occur indirectly. Accidental double installation within an inheritance tree can easily occur, and is partially trapped by the inheritance of `has_reference_count`. The guard is not proof against a later installation of a reference count into a base class, or against multiply inherited reference counts. Solutions to these problems require installation to be split into two phases, first to determine the overall requirements, and then to generate the corresponding declarations. The `Monitor` example in Section 6.4.2 shows how the more general problem can be resolved, without affecting application or glue code.

Alternatives

There is an efficient solution to this problem, for the case of unconstrained usage from a well-defined class, using the Curiously Recurring Template Pattern [Coplien95a] in which a derived class parameterises its base class.

```
class ApplicationClass : /* ... */,
                public ReferenceCounted<ApplicationClass>
{ /* ... */ };
```

In this application, the template parameter is required to support a `static_cast` from a pointer to the base `ReferenceCounted<ApplicationClass>` to a pointer to the derived `ApplicationClass` for use by the `delete` in a complete rather than partial implementation of `annul()`.

If protected destructors are used to enforce heap-only usage, an extra friend declaration is required to allow the base class access to the derived destructor, violating the goal of a single mention in the instantiation glue (unless a meta-function is used to define base class and friend).

A non-templated approach is more flexible, and necessary when multiple inheritance is involved:

```
class ApplicationClass : /* ... */,
                public virtual ReferenceCounted
{ /* ... */ };
```

The latter approach can always be used, but at the expense of an extra indirection for the much commoner single inheritance cases.

The meta-programming approach supports an arbitrary request for reference counting functionality, leaving the meta-program to choose an efficient implementation strategy. The conventional approach requires the programmer to choose and implement the strategy directly.

6.2.4 WholePart

Soukup [Soukup94] makes a persuasive case for implementing patterns using pattern classes. A pattern class is just a grouping of functions templated by the types of each collaborator. The pattern class contains no member variables and is a friend of each collaborator. A pattern is used by invoking the static member function of the pattern class with the appropriate collaborator instances as parameters. Since the pattern class is a friend of each collaborator, the function is free to peek and poke the working variables in the collaborator objects to perform the required actions. This approach achieves a very regular style of implementation, and has very beneficial effects in reducing include file dependencies. However the extensive use of friend declarations runs counter to normal programming practice.

Instantiation of a pattern using a pattern class requires instantiation of the pattern class, and insertion of friendship declarations and working variables into the collaboration classes. Instantiation of the pattern class is readily resolved by conventional C++ template instantiation. Insertion of declarations into the collaborators could be performed by manual editing. [Soukup94] describes a custom preprocessor for the CodeFarms library that performs this insertion automatically, provided the programmer has left a hook in each class of the form.

```
class State
{
    MEMBER_State   // Hook to enable Cpp to insert declarations
    /*...*/
};
class Town
{
    MEMBER_Town    // Hook to enable Cpp to insert declarations
    /*...*/
};
```

The custom preprocessor scans pattern class instantiations such as

```
WHOLE_PART(State,Town) // Declaration that State has many Towns
```

to produce conventional Cpp definitions such as

```
#define MEMBER_Town \
    friend class WholePart<State,Town>; \
    State *_whole;
```

Multiple patterns are readily accommodated; the Cpp macro just grows.

This approach demonstrates that implementation of a particular pattern solution requires that declarations be injected into the code for collaborator classes. Only for the degenerate case of a pattern involving a single class can injection be avoided. In the terminology of Aspect Orientation, the declarations associated with each aspect (or pattern) must be woven together to create composite declarations acceptable to the C++ compiler. FOG supports this weaving and eliminates the need for a custom preprocessor and for the preprocessor hooks that support the CodeFarms library.

```
template <class Whole, class Part>
auto declaration WholePart::install()
{
    class $Whole
    {
        friend class $Dynamic;
        private list<$Part> _parts;
        /* optional construction, delegations and destruction */
    };
    class $Part
    {
        friend class $Dynamic;
        private $Whole *_whole;
        /* optional construction, delegations and destruction */
    };
}
```

```
$WholePart<State,Town>::install();
```

Invocation of the installation function just adds the required friend declaration and member variable to each collaborator class identified by the template parameters. The semantics of meta-function execution involve replacement only of formal parameters in the scope of the meta-function, before returning the declarations in the meta-function body for interpretation within the invocation scope. The meta-function apparently has no formal parameters. It actually has four. All meta-functions have two built-in formal parameters `Static` and `Dynamic` corresponding to the declared scope of the meta-function and the actual scope, which differs if invoked for a derived class. In addition, each template parameter is also a formal parameter. The usage of `$Whole` rather than `Whole` therefore ensures that a replacement occurs before the body is returned to the calling scope where `Whole` may be undefined or differently defined.

The example shows only the minimum to activate the pattern solution. Additional declarations could enforce appropriate construction and destruction protocols, and provide delegation so that users are unaware that a pattern class is in use.

It is not necessary to use Soukup's pattern class approach, although it has a pleasant symmetry. Installation can be organised with respect to a dominant collaborator, probably the Whole class in this case. It is then only necessary to perform code injection into the other collaborators. However, whatever approach is adopted, a pattern solution with more than one collaborator class requires either manual editing to spread the pattern solution or automatic code injection.

6.2.5 Visitor

An implementation of the Visitor pattern [Gamma95] provides a more complete example of some of the facilities of FOG.

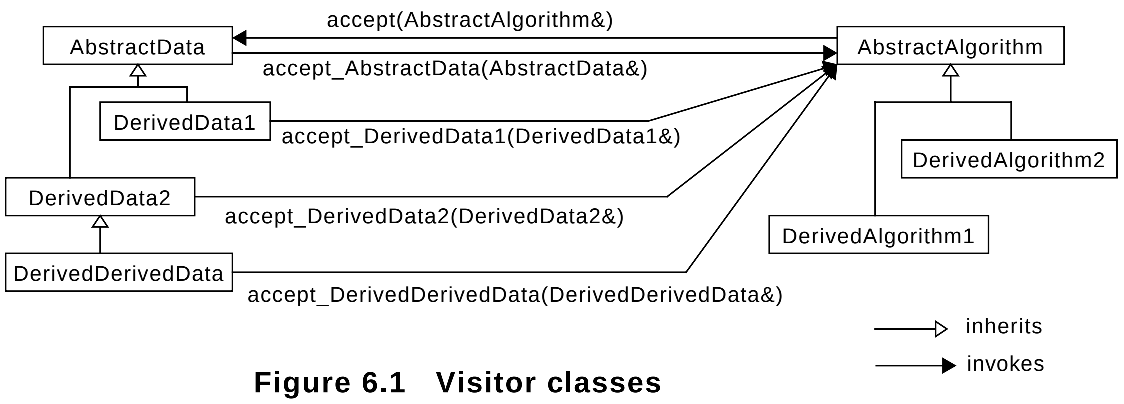
accept(AbstractAlgorithm&)
AbstractData
AbstractAlgorithm
accept_AbstractData(AbstractData&)
DerivedData1
accept_DerivedData1(DerivedData1&)
DerivedAlgorithm2
DerivedData2
accept_DerivedData2(DerivedData2&)
DerivedAlgorithm1
DerivedDerivedData
accept_DerivedDerivedData(DerivedDerivedData&)
inherits
invokes

Figure 6.1 Visitor classes

The pattern involves a hierarchy of data classes (`DerivedData1`, `DerivedData2`, etc.) and a number of algorithms that may be performed on the data. The algorithms are realised by algorithm classes (`DerivedAlgorithm1`, `DerivedAlgorithm2`, etc.) and inherit from the abstract Visitor class `AbstractAlgorithm`. The data classes similarly inherit from an abstract class `AbstractData`.

Usage of the pattern involves a two dimensional (double) dispatch to ensure that the appropriate data and algorithm dependent action is performed. This is achieved by an invocation of the virtual method `AbstractData::accept(AbstractAlgorithm&)`, whose derived implementation invokes `AbstractAlgorithm::accept_DerivedData`*n*`(DerivedData`*n*`&)`, which can in turn have a derived implementation to perform the required action as `DerivedAlgorithm`*m*`::accept_DerivedData`*n*`(DerivedData`*n*`&)`. In a system with A algorithms and D data classes, there may be as many as A*D functions to be declared and implemented. The implementations in the algorithm classes perform the required actions in response to the double dispatch. The scaffolding code required in the data element classes can be generated automatically by FOG.

The pattern has two degrees of freedom, which contributes to the inconvenience of a conventional manual approach. Addition of an extra algorithm class is relatively benign, requiring just that the new algorithm class implements as many of the data functions as required. Addition of an extra data class requires that the data class complies with the inherited protocol and that an additional method be defined for the abstract algorithm. It may also be necessary to implement this method in every derived algorithm class.

The example implementation uses two meta-functions, one to be invoked in the root data class (`AbstractData`), and another to be invoked in each derived data class (`DerivedData`*n*).

A derivation rule cannot satisfy the derivation requirements of this pattern, since a derivation rule can only regenerate potential declarations in derived contexts. In this case, derivation of a data element class (`DerivedData`*n*) needs to export a declaration to the abstract visitor class (`AbstractAlgorithm`). A generalisation of the rule could be considered, but appears to be necessary only for this pattern. The cost of a one-line invocation in each derived data element class is small compared to the associated response code in the algorithm classes, and provides flexibility for deliberate omission if abstract levels of the data element hierarchy do not need support.

```
auto declaration VisitorBaseElement(identifier V)
{
    auto identifier Visitor = $V;
    public virtual void accept($V& aVisitor) = 0;
};
auto declaration VisitorDerivedElement()
{
    public virtual void accept($Visitor& aVisitor)
        { aVisitor.accept_${Scope}(*this); }
    public virtual void ${Visitor}::accept_${Scope}($Scope& aData)
        {}
};
```

The abstract classes may invoke the pattern as

```
    class AbstractAlgorithm /* ... */
    {
        // ...
    };
    class AbstractData /* ... */
    {
        $VisitorBaseElement(AbstractAlgorithm);
        // ...
    };
```

Invocation of `VisitorBaseElement` initializes the `AbstractData::Visitor` meta-variable with the value `AbstractAlgorithm`, avoiding the need to pass the same parameter to the invocations in derived classes:

```
class DerivedData1 : /* ... */ public AbstractData /* ... */
{
    $VisitorDerivedElement();
    //...
};
```

```
class DerivedData2 : /* ... */ public AbstractData /* ... */
{
    $VisitorDerivedElement();
    //...
};
class DerivedDerivedData : /* ... */ public DerivedData2 /* ... */
{
    $VisitorDerivedElement();
    //...
};
```

Definition of the derived algorithm classes, like the abstract algorithm classes requires no explicit code. The declarations are provided automatically by the second declaration in the `VisitorDerivedElement` meta-function, which also provides a default empty algorithm implementation. Since the signatures of the algorithm are defined in the base class, implementation of the derived algorithm code need only mention the name.

```
    class DerivedAlgorithm1
    {
        using accept_DerivedData1
        {
            // ...
        }
    };
```

The short form *using-declaration* requires usage within class braces. Independent usage can be provided by a long form *using-declaration* at the expense of a double mention.

```
using AbstractAlgorithm::accept_DerivedData1
            DerivedAlgorithm1::accept_DerivedData1
{
    // ...
}
```

or directly at the cost of requiring distributed editing to change a signature:

```
void DerivedAlgorithm1::accept_DerivedData1(DerivedData1& aData)
{
    // ...
}
```

The pattern is expressed compactly, and instantiated so that its use is clear. Compliance with the pattern is ensured because the pattern provides all the relevant declarations. The manually contributed code is reduced to that necessary to provide the actual implementation. The scaffolding is almost completely removed.

6.3 Meta-Programming

6.3.1 OS Traits

Conditional compilation is essential to support a variety of configuration options, often to resolve distinctions between different operating systems. Control values are variously predefined by the compiler, supplied by command line or defined in header files.

```
    #if defined(UNIX)
        static const char *temp_path = "/tmp/";
    #else
        static const char *temp_path = "C:\\Temp\\";
    #endif
```

C++ statements occur only within functions and express evaluations to be resolved at run-time. In FOG, meta-statements are declarations and so the example simplifies to:

```
    auto bool unix = $std::get_cpp("UNIX") != "";
    if (unix)
        static const char *temp_path = "/tmp/";
    else
        static const char *temp_path = "C:\\Temp\\";
```

The invocation of `std::get_cpp` looks up `UNIX` in the C preprocessor namespace, providing controlled access to command line definitions.

Using an Object-Oriented perspective eliminates the need for conditionalisation. The characteristics of each configuration option may be packaged as meta-variables (and meta-functions) of a (meta-)class, extending the traits concepts of [Myers95].

```
class OsTraits_Abstract
{
    auto static bool NT = false;                // default value
    auto static bool UNIX = false;
    //...
};
class OsTraits_Nt : public OsTraits_Abstract
{
    auto static bool NT = true;                 // overriding value
    auto static string temp_path = "C:\\Temp\\";
    //...
};
class OsTraits_Unix : public OsTraits_Abstract
{
    auto static bool UNIX = true;
    auto static string temp_path = "/tmp/";
    //...
};
```

The appropriate configuration may then be chosen using `std::get_cpp("OS")` to resolve `OS` from the command line. Thus

```
    class OsTraits : public OsTraits_$std::get_cpp("OS") {};
```

maps the required configuration to `OsTraits`. The appropriate operating system may be specified on the FOG command line by:

```
    fog ... -D OS=Nt ...
```

A file may then be opened on the operating system specific temporary path by:

```
    std::ofstream s($OsTraits::temp_path "results.dat");
```

(The pair of strings concatenate to give the required file name).

Having isolated the configuration in separate classes and an associated header file, a new operating system can be supported by providing a prefix file characterising the new system and invoking it with an appropriate command line. Existing source files need no change. This could be achieved directly using multiple layers of name substitutions with C preprocessor, but it never is. Modularization is much easier when supported by the programming environment. This cannot be achieved by templates, which lack the ability to perform string manipulations.

6.3.2 SynchronisedEnum

In Section 3.4.7 on page 95, an example was given showing how a meta-function could transform a list of enumerations into an array of text strings. An alternative approach may sometimes be preferable, defining additional enumerators and text array elements incrementally.

```
class ApplicationClass
{
    public enum Enum {};
    public static const char *texts[] = {};
    auto declaration define(identifier aName, expression aValue)
    {
        enum Enum { $aName = $aValue };
        static const char *texts[] = { [$aValue] ""$aName };
    }
};
```

Application code may then define enumerators with explicit values (and text elements) by:

```
    $ApplicationClass::define(LABEL, 40);
```

The example can be usefully simplified and made more reliable by using the sequentially allocated enumerator values directly:

```
    auto static declaration define(identifier aName)
    {
        enum Enum { $aName };
        static const char *texts[] = { ""$aName };
    }
```

Alternatives

The above example cannot be expressed in C++. Conventional practice requires that a maintainer update both enumeration and array of text strings consistently. The FOG pattern updates both at once, and provides freedom for each invocation of `ApplicationClass::define` to be located with code associated with the invocation, avoiding the need to fragment code to satisfy compiler constraints.

6.3.3 RTTI

Prior to the adoption of RTTI into C++, it was necessary for large Object Oriented programs to implement RTTI as part of the application code. Each third party library had its own policy, that was awkward to implement internally and imposed significant compliance burdens upon clients. RTTI within standard C++ eliminates much of the add-on functionality, however it pursues the C++ philosophy of avoiding costs of unused functionality by providing only minimal functionality. Extra functionality must still be provided extra-lingually, using the unique RTTI type identifier as an index to custom information. Applications may therefore still need to provide RTTI, albeit tailored to exploit the built-in functionality. FOG meta-programming supports conversion of declarations into a format suitable for use at run-time. A simple example supporting just class name diagnostics and inheritance tables is provided here. A more extensive example involving formatting of member variable names is provided to support marshalling in Section 6.4.1.

Class-name information

Class-name information may be provided by:

```
class Rtti;

auto declaration Rtti::class_names()
{
    virtual const char *class_name() const
        :{ derived(true) { return ""@Scope; } };
}
```

and invoked as

```
class ApplicationClass
{
    $Rtti::class_names();
}
```

to have class_name() return the class name for ApplicationClass and all its derived classes. This does not work for template classes, whose full name is not known at meta-compile-time. A rather more elaborate approach is necessary that concatenates the names of each argument during construction of a static instance within the invoked method.

Inheritance information

```
class Rtti
{
public:
    struct Rtti::BaseInfo        // List element in list of bases
    {
        const Rtti *_base_rtti;  // Base class description
        int _offset_to_this;     // Base position
    };
    Rtti(const char *className, const BaseInfo baseList[]);
};
auto declaration Rtti::base_names()
{
    public virtual const Rtti& dynamic_type_info() const      //1
        :{ derived (true) { return static_type_info(); } };
    public static const Rtti& static_type_info()              //2
        :{ derived (true)
            {
                static const Rtti rtti(""@Scope, _base_info);  //3
                return rtti;
            }
        };
    protected static const Rtti::BaseInfo _base_info[];       //4
        :{ derived (true); };
    auto ~${Scope}()                                          //5
    {
        for (iterator b = $bases(); b; ++b)                   //6
            using _base_info =
            { {
                &${b}::static_type_info(),                    //7
                (char *)&($b&)($Scope&)*(char *)0x1000 -      //8
                        (char *)0x1000
            } };
        using _base_info = { { 0, 0 } };                      //9
    };
};
```

The functionality is installed in the same way as before with a single line at the root of an inheritance hierarchy:

```
class ApplicationClass
{
    $Rtti::base_names();
};
```

as a result of which each class has two methods static_type_info (2) and dynamic_type_info (1) that return an Rtti descriptor object, a null-terminated array of base class descriptors (4) and a meta-destructor (5). The dynamic method is virtual and so ensures that the appropriate static method is invoked to return a local Rtti instance constructed (3) from the (derived) class name and the local list of base class descriptors _base_info. The meta-destructor executes in each class to loop over all base classes (6). Each iteration adds a list element comprising a pointer to the base-class type information (7), and an offset of the base-class within the derived object (8). The extraordinary number of casts are necessary to ensure that the compiler changes the non-zero value `0x1000` firstly to represent the derived object, and then the base object, before performing a byte level difference. Finally the list is null-terminated (9).

This technique generates predictable functionality in derived classes automatically, and so compares very favourably with traditional approaches relying on multiple preprocessor scaffolding macros per class.

6.4 Aspects

Aspect Oriented Programming [Mens97] seeks to isolate independent programming concerns as aspects, each of which can be implemented (and re-used) independently. The next two examples show partitioning of a problem into the application, some additional concern implemented as an aspect, and a very small amount of glue code to initiate aspect weaving. The examples also demonstrate how application code is simplified and consequently made more reliable by the initial expenditure of extra declarative effort to achieve re-use. This is a natural extension of C++ philosophy, where a class can encapsulate difficult concepts making use easy although definition difficult. FOG provides additional declarative power supporting better encapsulation.

6.4.1 Marshalling[5]

Communication between programs requires messages to be passed between those programs. Each message is usefully represented as an object, and so the programmer is presented with the problem of transferring the contents of one object between programs. This is readily achieved using an Interface Definition Language and CORBA when such high level facilities are available, however when working at a lower level the problem must be solved by the programmer.

A typical approach involves the conversion of each object into a sequence of bytes with a common header that describes the format and length of the subsequent bytes. The sending program must marshal the data elements of each object into the byte stream and the receiving program must perform the corresponding unmarshalling back into an object. Preparation of this marshalling code is straightforward, but not amenable to automation with conventional compilers. In order to show how this can be resolved by FOG, it is helpful to first show one possible conventional solution. The exposition matches the subsequent automated solution. Numbered comments (//3.0) may assist the reader in correlating the two solutions.

All messages inherit from the `Message` class, that defines the marshalling and unmarshalling interfaces and an enumeration, whose values distinguish between possible message formats.

```
typedef unsigned char uchar;     // Short name to shorten lines
```

5. A version of this example that was tested using the multi-pass implementation of FOG was presented at TOOLS Eastern Europe [Willink99b]. That version used ad hoc token pasting within function bodies. The version presented here is an untested revision to exploit the syntactic composition of function bodies available with the superset implementation.

```
class Message
{   /* ... */
protected:
    enum MessageTypes                                       //1.0
    {
        MESSAGE_StockReport /* , ... */                     //1.1
    };
public:
    virtual size_t marshal(uchar data[]) const;             //2.0
    static Message *unmarshal(uchar data[]);                //3.0
};
```

Invocation of the `marshal` function fills `data[]` with the byte stream and returns the message size. The `unmarshal` function is passed a byte stream and returns a pointer to an object if the message is valid, or 0 on failure. A single very simple message comprising just two data elements is used for this example.

```
class StockReport : public Message
{   /* ... */
private:
    unsigned long _item_number;
    short _stock_level;
private:
    inline StockReport(uchar data[]);                       //5.0
    inline void initialize(uchar *p);                       //6.0
public:
    static StockReport *make(uchar data[]);                 //4.0
    virtual size_t marshal(uchar data[]) const;             //2.1
};
```

The message-specific marshalling into `data[]` is performed by a virtual function:

```
size_t StockReport::marshal(uchar data[]) const            //2.2
{
    uchar *p = data;                                        //2.3
    *p++ = MESSAGE_StockReport;  // Message type            //2.4
    *p++ = 6;                    // Message body length     //2.5
    *p++ = (_item_number >> 24) & 0xFF;                    //2.6
    *p++ = (_item_number >> 16) & 0xFF;
    *p++ = (_item_number >> 8) & 0xFF;
    *p++ = _item_number & 0xFF;
    *p++ = (_stock_level >> 8) & 0xFF;                      //2.7
    *p++ = _stock_level & 0xFF;
    return p - data;                                        //2.8
}
```

Unmarshalling from `data[]` selects the message-specific routine:

```
Message *Message::unmarshal(uchar data[])                  //3.1
{
    switch (data[0])                                        //3.2
    {                                                       //3.3
        case MESSAGE_StockReport:                           //3.4
            return StockReport::make(data);
        /* ... */
        default:                                            //3.5
            return 0;      // 0 for bad message type error.
    }                                                       //3.6
}
```

Then the message-specific object is created, but only if the length is valid:

```
StockReport *StockReport::make(uchar data[])            //4.1
{
    if (data[1] != 6)
        return 0;         // 0 for bad message length error.
    else
        return new StockReport(data);
}
```

Finally, construction and an initialization routine perform the message-specific unmarshalling:

```
StockReport::StockReport(uchar data[])                  //5.1
{
    initialize(data+2);                                 //5.2
}
void StockReport::initialize(uchar *p)                  //6.1
{
    {                                                   //6.2
        unsigned long temp = *p++;
        temp = (temp << 8) | *p++;
        temp = (temp << 8) | *p++;
        temp = (temp << 8) | *p++;
        _item_number = temp;
    }
    {                                                   //6.3
        unsigned long temp = *p++;
        temp = (temp << 8) | *p++;
        _stock_level = short(temp);
    }
}
```

The `initialize` routine can be folded into the constructor. However, it is easier to compose code into a function that is not overloaded, so an extra layer of inline function call is used, rather than a typedefed signature as in Section 2.1.4.4

The marshalling and unmarshalling code is very predictable and in principle easy to write, however when there are many messages, it is tedious and error prone. When a data type is changed or a member variable added, there are many places where updates are required. It is preferable to generate the code automatically. This requires a meta-program that can reflect upon the message class declarations and generate code accordingly.

Application aspect

The marshalling support may be separated completely from the application code. The message classes express their own inheritance relationships, their data contents, and any other application declarations that may be necessary.

```
typedef unsigned char uchar; // Short name to shorten lines
class Message { /* ... */ };
class StockReport : public Message
{   /* ... */
private:
    unsigned long _item_number;
    short _stock_level;
};
```

Aspect weaving

The marshalling aspect is added (woven) by invoking the installation meta function of the `Marshal` meta-class.

```
using "Marshal.fog";
```

```
    class Message
    {
        $Marshal::install();
    };
```

Marshalling aspect

In the following code the extended *using-declaration* is exploited to avoid repeating declarations. A repeated occurrence of

```
    public virtual size_t marshal(uchar data[]) const
```

is written as just

```
    using marshal;
```

when the name visible as `marshal` is required in the current scope, or as

```
    using marshal X::marshal;
```

when the name visible as `marshal` in the current scope is required in `X`.

```
auto class Marshal {};
auto declaration Marshal::install()
{
    auto class MessageClass = $Scope;                         //10
    protected enum MessageTypes {};                           //1.0
    public virtual size_t marshal(uchar data[]) const;        //2.0
    public static $MessageClass *unmarshal(uchar data[]); //3.0
    auto ${Scope}()                                           //11
    {
        protected enum ${MessageClass}::MessageTypes
            { MESSAGE_$Scope };                               //1.1
        using unmarshal ${MessageClass}::unmarshal            //3.1
        {
            case MESSAGE_$$Scope:                             //3.4
                return $${Scope}::make(data);
        }
        using marshal;                                        //2.1
        public static $Scope *make(uchar data[]);             //4.0
        private inline ${Scope}(uchar data[])                 //5.0/1
            { initialize(data+2); }                           //5.2
        private inline initialize(uchar *p) {}                //6.0
    }
    auto ~${Scope}()                                          //12.0
    {
        auto unsigned byte_count = 0;                         //12.1
        for (iterator i = $all_variables(); i; ++i)           //12.2
            if (!i->is_static())                              //12.3
                i->type().marshal($i->name());                //12.4
        using marshal                                         //2.2
        :{
            auto body = {
                          uchar *p = data;                    //2.3
                          *p++ = MESSAGE_$Scope;              //2.4
                          *p++ = $byte_count;                 //2.5
                          @body;                              //2.6/7
                          return p - data;                    //2.8
                       };
        };
```

```
            using make                                                 //4.1
            {
                if (data[1] != $byte_count)
                    return 0;
                else
                    return new ${Scope}(data);
            }
            if ($Scope == $MessageClass)
            {
                using unmarshal                                        //3.1
                :{
                    auto body = {
                                    switch (data[0])                   //3.2
                                    {                                  //3.3
                                        @body;                         //3.4
                                        default:                       //3.5
                                            return 0;
                                    }                                  //3.6
                                 };
                };
            }
        };
    };
```

The `install` meta-function is invoked from a class declaration for `Message` and executes as part of the source file reading and analysis compilation phase. All lines declare declarations that are added to the `Message` class. A meta-variable (10), meta-constructor (11) and meta-destructor (12.0) are declared in addition to more conventional declarations (1.0, 2.0, 3.0).

The first declaration (10) caches the name of the invocation scope in the meta-variable `MessageClass` for access by the meta-constructor, meta-destructor and their derivations[6]. Then the marshalling interface is defined (1.0, 2.0, 3.0) for the `Message` class.

The meta-constructor (11) is invoked for `Message` and all its derived classes during the meta-construction compilation phase. Invocations occur in least derived first order. The meta-constructor first defines an enumerator and switch case contribution to the root message class, whose identity was cached in the `MessageClass` meta-variable.

An enumerator is defined (1.1) in the `MessageTypes` enumeration of the `Message` class. The additional enumerator extends the enumeration and so acquires a unique value for each message class. The enumerator name is formed by concatenation of the prefix `MESSAGE_` and `$Scope`, the derived message class name.

A switch case is defined (3.4) as part of the unmarshalling function of the root message class. The invocation of `Scope` built-in variable occurs within the lexically nested `MessageClass` function, necessitating the additional `$` to resolve `Scope` in the surrounding lexical context. (In this case `$Dynamic` could be used instead).

The meta-constructor then defines the interface and trivial parts of the implementation for the derived message class, on whose behalf the meta-constructor is executing.

The meta-destructor (12.0) is similarly invoked on behalf of each message class, during the meta-destruction phase, by which time all member variables have been defined. It comprises a loop to resolve the member variable dependent code and

6. `MessageClass` is not strictly necessary. Its use at 3.0 can be replaced by `$Scope`, with `$Static` replacing subsequent usage.

count the number of bytes in the message. It then finishes with those parts of the implementation that need knowledge of the byte count.

The body of the `marshal` routine is redefined (2.2) to wrap pre-amble and post-amble code around the contributions accrued during the earlier loop.

The `make` routine is defined in its entirety (4.1) embedding the calculated count.

A pre-amble and post-amble are also wrapped around the `unmarshal` routine (3.1) but only when the meta-destructor is executing on behalf of the `MessageClass`, since the `unmarshal` routine is a static present only in that class. Although the pre-amble and post-amble are independent of the member variable calculations, this code must be defined after the internal switch cases have been composed in order to wrap the switch around rather than amidst the cases. Accruing the cases during meta-construction and applying the surrounds during meta-destruction ensures this ordering.

The byte count is maintained in a meta-variable initialised to `0` (12.1). The loop (12.2) iterates over all member (and inherited member) variables of the derived class, and (12.3) skips static member variables. Within the loop (12.4), invocation of the `marshal` meta-function for the data-type of each member variable causes emission of member-specific marshalling code. The member variable name is passed as a parameter to a type-specific implementation such as:

```
auto declaration unsigned long::marshal(expression name)  //13.0
{
    byte_count += 4;                                          //13.1
    using marshal                                             //2.2
    {
        *p++ = ($name >> 24) & 0xFF;                          //2.6
        *p++ = ($name >> 16) & 0xFF;
        *p++ = ($name >> 8) & 0xFF;
        *p++ = $name & 0xFF;
    }
    using initialize                                          //6.1
    {
        {                                                     //6.2
            unsigned long temp = *p++;
            temp = (temp << 8) | *p++;
            temp = (temp << 8) | *p++;
            temp = (temp << 8) | *p++;
            $name = temp;
        }
    }
}
```

Meta-functions can be defined for built-in types as well as user defined types. The above declaration for the `unsigned long` ‘class’ supports the marshalling of `unsigned long` member variables. The formal parameter name is replaced throughout the body before the body is interpreted in the invoking context, that of the derived message class. The update of the `byte_count` (13.1) therefore maintains the counter of the derived message class, and the two declarations (2.2, 6.1) provide additional code for the body region of the derived message class routines. The member variable iteration is in declaration order, and the ordering of function body contributions is preserved, so the final ordering of the many contributions is well-defined. The contributed code (2.6, 6.2) just performs the very simple operations appropriate to the data type.

A similar isomorphic meta-function for `short` is needed to complete the example (2.7, 6.3), and further routines for every other primitive data type. Nested data types can be resolved by a nested iteration, which can be specified as a general-purpose meta-function, passing the nested member name to the nested call, necessitating the use of the `expression` rather than `name` or `identifier` for the parameter type.

```
    auto declaration Marshal::marshal(expression name)
    {
        for (iterator i = $all_variables(); i; ++i)
            if (!i->is_static())
                i->type().marshal(${name}.$i->name());
    };
```

The general purpose meta-function can be installed by meta-inheritance for use in a nested type

```
    struct NestedDataType : auto Marshal { /* ... */ };
```

This declares `Marshal` as an additional base class of the `NestedDataType`, but only at (meta-)compile time. The meta-names of `Marshal` are therefore visible to the derived class, providing the required resolution of `NestedDataType::marshal()`.

This example shows how application code can be generated in response to the actual application declarations. The code is fully under the programmer's control. The programmer can freely choose an alternate implementation using data tables to describe each message rather than monolithic functions. Inheritance of messages can be exploited to trade size for speed, by changing compile-time iterations to serve only the local member variables, and changing the run-time code to invoke base class methods for inherited members. More sophisticated code can be provided to support swizzling of pointer types for database applications.

The generated code is portable, since all members are referred to by name. The example code for `unsigned long::marshal` has a portability problem for processors with a greater than 32 bit `unsigned long`, but this is a limitation of the example solution, not of the approach.

6.4.2 Monitor[7]

The ability to use FOG to separate different programming concerns is demonstrated by application of a synchronisation monitor to a stack. One line of glue code is necessary to weave the otherwise independent functionality of monitor and stack.

Application Aspect

We first define a simple stack class.

```
temaplate <class T>
class Stack
{
    $NoCopy();
    $NoAssign();
private:
    T *_elements;
    size_t _capacity;            // Allocated size of _elements[]
    size_t _tally;               // Used size of _elements[]
```

7. A version of this example, that was tested using the multi-pass implementation of FOG, appears in a position paper for the AOP workshop at ECOOP'99 [Willink99c]. The version here resolves reverse and multiple inheritance conflicts. This version has been adapted to use the revised syntax of derivation rules as a part of an *object-statement-clause*. It is untested.

```
public:
    Stack() : _elements(0), _capacity(0), _tally(0) {}
    ~Stack() { /* ... */ }
    bool is_empty() const volatile { return _tally == 0; }
    T pop() { /* ... */ }
    void push(const T&) { /* ... */ }
    T top() const { return _elements[_tally-1]; }
};
```

The `const` qualifier is used conventionally to indicate that no change should occur. Concurrent readers are therefore permissible, but concurrent writing should not be permitted once a monitor aspect has been added.

The `volatile` qualifier is used to indicate that access may occur without the use of a lock, allowing interleaved reading and writing by other threads.

> The qualification of `is_empty()` as `volatile` as well as `const` therefore goes beyond conventional practice, but is safe because the implementation involves a single read, whereas `top()` involves at least two reads.
>
> Qualification of `is_empty()` as `volatile` is of limited utility, since the return accurately reflects a state that existed but that state may no longer exist when the calling code interprets the result. It would seem that the `volatile` qualification is redundant since calling code must establish a lock to encompass both a `!is_empty()` and a subsequent `pop()`. However, the `volatile` qualification is useful when `is_empty()` is invoked within a polling loop that can recover on the next iteration. The presence of `volatile` avoids incurring locking costs for such a loop.
>
> The usage is consistent because `volatile` indicates that concurrent change may occur and so inhibits any optimisation that could reorder the sequence of memory accesses.

The Monitor Aspect (run-time)

The monitor functionality is provided by a `Monitor` class, whose detailed implementation is not relevant to this example.

```
    class Monitor
    {
        friend class Monitor::ReadOnlyLock;
        friend class Monitor::ReadWriteLock;
    private:
        void acquire_exclusive() { /* ... */ }
        void acquire_shared() { /* ... */ }
        void release() { /* ... */ }
    };
```

`acquire_exclusive` and `acquire_shared` block until exclusive or shared access is available to the resource(s) managed by the monitor. `release` terminates the resource reservation.

Reservation of the monitored resource is managed by a pair of nested lock classes, `ReadOnlyLock` and `ReadWriteLock`. They differ only in whether `acquire_shared` or `acquire_exclusive` is invoked.

```
public class Monitor::ReadOnlyLock
{
private:
    Monitor& _monitor;
public:
    ReadOnlyLock(Monitor& aMonitor)
        : _monitor(aMonitor) { _monitor.acquire_shared(); }
    ~ReadOnlyLock() { _monitor.release(); }
};
```

A lock class invokes `acquire_shared` to acquire the resource during construction and ensures its release from the destructor, whose invocation C++ guarantees.

Aspect composition

The `Stack` application code above is written independently of the synchronisation code. The presence of the `volatile` keyword is an optional optimisation.

The monitor aspect is added to the application aspect by providing additional declarations that are woven into the application code.

```
using "monitor.fog";

template <class T>
class Stack
{
    $Monitor::install();
};
```

The Monitor Aspect (compile-time)

The remainder of the code for this example forms part of the `monitor.fog` include file. There are two relatively independent code injections to be performed to install the monitor. The class declaration must be updated to incorporate an instance of `Monitor`. Function declarations must be updated to establish locks.

Direct installation of an instance of `Monitor` is relatively straightforward and could be achieved by just adding a member variable. However, making the meta-function work in a more general purpose fashion is harder. There are five problems to be resolved:

- `Monitor::install` may be invoked more than once on the same class
- `Monitor::install` may be invoked later for a derived class
- `Monitor::install` may be invoked later for a base class
- `Monitor::install` may be invoked for more than one base class
- `Monitor::install` may be invoked for a derived monitor

There must be only one synchronisation monitor in each object, so multiples must be suppressed, retaining only the one monitor in the least derived class. If a monitor is multiply inherited, virtual inheritance must be used to share it. If ‘multiple’ monitors use different implementation classes, we will generate a compiler diagnostic.

Since virtual inheritance must be used to resolve the multiple inheritance problem, it is convenient to implement the monitor by non-virtual multiple inheritance rather than as a member variable. Resolving multiple inheritance then just requires composing the virtual keyword on the simpler and much commoner inheritance.

Resolving the uniqueness problem when the invocation order of `Monitor::install` class cannot be known requires splitting the structural problem into two phases. A third phase is required to update the functions.

Semantic analysis phase

The first phase is executed directly by the meta-function invocation from the semantic analysis. It sets flag variables indicating the class requirements, and arranges for the second phase to occur later.

```
auto const class Monitor::needs_monitor = 0;              //1

auto declaration Monitor::install()                       //2
{
    class $Scope : auto $Dynamic {};                      //3
    auto const class needs_monitor = $Dynamic;            //4
}
```

//1 declares a meta-variable in the `Monitor` class whose 0 (nil) initialisation flags that a `Monitor` class does not need `Monitor` functionality inserted into it.

//2 declares the compile-time meta-function invoked by the application glue code.

//3 adds the `Monitor` class as a meta-base class of `Stack`.

```
    template <class T> class Stack : auto Monitor { ... };
```

`$Scope` is not a formal and so resolves to the prevailing scope in the invocation context.

`$Dynamic` is a built-in formal that resolves to actual definition scope, typically `Monitor`, but `DerivedFromMonitor` if `Monitor::install` is invoked as the inherited meta-function `DerivedFromMonitor::install`.

Installation of `Monitor` as a meta-base-class ensures that the functionality of the `Monitor` meta-constructors and meta-destructors is executed as part of `Stack` (and all its derived classes).

//4 declares the `Stack::needs_monitor` meta-variable with a non-0 value to signal that `Monitor` functionality is required. The value of `Dynamic` is used as the non-0 value, so that in combination with the `const`, any attempt to install a different class of monitor will be caught. Re-installation of the same monitor class is allowed.

Meta-construction phase

The first phase sets the `needs_monitor` flag non-0 in all classes that are specified as requiring monitor functionality, and arranges for the meta-constructor to be invoked. Invocation in the first phase occurs in an unpredictable order. Invocation during the meta-construction phase occurs in a least derived first order, which can be exploited to install the monitor in the least derived alternative. Resolution of the multiple inheritance conflict requires a further pass, which is implemented by performing an iteration over the multiple bases.

```
auto Monitor::Monitor()
{
    if (needs_monitor)                                        //1
    {
        auto class baseMonitors[] = $find(has_monitor);       //2
        if (baseMonitors.size() == 0)                         //3
        {
            class $Scope : public $needs_monitor {};
            auto const class has_monitor = $Scope;
        }
        else if (baseMonitors.size() == 1)                    //4
        {
            auto const class needs_monitor = $baseMonitors[0];
        }
        else                                                  //5
        {
            for (iterator m = $baseMonitors; m; ++m)
            {
                class $*m : virtual $needs_monitor {};
                auto const class needs_monitor = $*m;
            }
        }
    }
}
```

//1 Execution of the meta-construction code is guarded by a test for a non-0 flag, thereby inhibiting installation of monitor functionality in monitor classes.

//2 `find` returns a list of all visible definitions of the `has_monitor` flag, which is used to identify the location(s) where monitor functionality is already installed.

//3 If there are no definitions visible, this must be a least-derived requirement and so the monitor class is specified as a public base class, composing with the existing specification as a meta-base class. The `has_monitor` flag is defined to indicate the location of the monitor functionality.

//4 If there is exactly one definition visible, then the inherited functionality is adequate and no further functionality is required in this class. (Re-)declaration of the derived `needs_monitor` provokes an error message if conflicting monitor classes are in use.

//5 If there is more than one definition visible, then a multiple inheritance conflict must be resolved. The iteration loops over all definitions and redefines the base-class to use virtual inheritance, and detects conflicting monitor classing.

Meta-destruction phase

The final phase of monitor installation should occur after any concurrent meta-programming has defined additional member functions, so that all member functions may have locking code inserted.

```
auto Monitor::~Monitor()
{
    if (needs_monitor)                                          //1
    {
        for (iterator f = $functions(); f; ++f)                 //2
        {
            if (f->is_static())                                 //3
                ;
            else if (f->is_volatile())                          //4
                ;
            else if (f->is_const())                             //5
            {
                $f->signature()
                :{
                    entry { ReadOnlyLock aLock(*this); }
                };
            }
            else                                                //6
            {
                $f->signature()
                :{
                    entry { ReadWriteLock aLock(*this); }
                };
            }
        }
        if ($friends().size() != 0)                             // 7
            $std::error("friend of monitored " $Scope " detected.");
    }
}
```

//1 Once again functionality is guarded to prevent operation on monitor classes.

//2 The loop over all functions uses the decl-specifiers to determine whether monitor code needs inserting.

//3 `static` member functions are not associated with any object and so have nothing to monitor access to.

//4 `volatile` is recognised as a requirement to bypass locking.

//5 `const` member functions require a shared lock, which is provided by specifying an entry code segment for the function whose full name is returned by `function::signature()`.

//6 Similarly non-`const` member functions require an exclusive lock.

//7 Finally the problem of friend functions and classes subverting the protection is resolved in a very heavy handed fashion by banning friends. (Direct access by friends must be changed to use access functions into which locks can be inserted automatically. It is very hard and probably impossible to analyze all code associated with a friend class or function to guarantee that it does not violate access constraints.)

Function Weaving

FOG performs function weaving by concatenating the code from multiple function bodies, within the five named regions `entry`, `pre`, `body`, `post` and `exit` (see Section 3.4.8). The entry region precedes the default body region and so the above meta-program generates the additional contribution

```
template <class T>
T Stack::top() const                // from the meta-destructor
:{
    entry { ReadOnlyLock aLock(_monitor); };
};
```

to be woven with the application function:

```
template <class T>
T Stack::top() const                // from application aspect
    { return _elements[_tally-1]; }
```

to generate the final C++ result:

```
template <class T>
T Stack::top() const
{
#line ...
    ReadOnlyLock aLock(_monitor);
#line ...
    return _elements[_tally-1];
}
```

References

Monitor and Stack are fundamental concepts and consequently staples for numerous articles in many Computer Science fields.

[Stroud94] used OpenC++ version 1 to enforce atomic integrity by intercepting method calls at run-time.

[Hedin97b] considered the monitor from an Aspect Oriented perspective, and introduced an attribute extension language to enable a preprocessing stage to validate that the requisite coding constraints had been observed. In this example we use reflection to synthesise the required code directly.

[Bjarnason97] advocates an extensible language, so that the required monitor protocol can be incorporated into the extended language. Language extension involves manipulation of syntax trees, and it is not clear how practical this is for a language with as challenging a syntax as C++.

6.5 A Real Example - BURG

FOG grew out of work to improve productivity in a different field of compilation technology.

It is difficult to apply high level programming concepts to Digital Signal Processors because of the very poor quality of the available compilers [Willink97b]. This is in part due to lack of awareness of the need for better support and partly due to the extreme difficulty of matching the performance of hand-crafted assembler on rather challenging architectures. Research therefore started to apply modern VLIW scheduling concepts, using an intermediate representation supporting data parallelism [Muchnick93]. The intermediate representation was extended to support user-characterised types [Willink97a], as part of a relatively general purpose compiler framework.

One of the activities of a compiler involves selection of appropriate machine instructions (such as ADD or MOVE) to implement the program, usually represented by a tree of Abstract Syntax Tree nodes [Aho86]. An effective approach to solving this problem involves a Bottom-Up Rewrite System [Proebsting95], which searches the tree from the leaves upwards identifying the lowest cost solution that has each node covered exactly once by a machine instruction. The tree may then be rewritten in terms of the selected machine instructions. In order to support multiple target architectures, alternative instruction sets must be covered. Implementation of this diversity is assisted by the use of a Bottom-Up Rewrite Generator to transform a description of each machine instruction into the form needed for an efficient tree search. An example of this form of generator is lburg that forms part of the lcc C compiler [Fraser95].

lburg is a compact C program comprising just three files:

- lburg.c has 690 lines and 4652 tokens
- lburg.h has 66 lines and 259 tokens
- gram.y is a 19 rule, 37 state yacc parser grammar

(token counts are non-comment, non-whitespace preprocessor tokens.)

lburg supports single dispatch architectures (such as SPARC). An enhanced version was required in order to support less conventional processor architectures, and so a highly Object Oriented C++ rewrite was undertaken using reference counting and smart pointers to share common partial instructions. The resulting program was substantially larger, due to the extra declarations for encapsulated C++ classes, rather than the original free access to structure elements, and due to the added functionality. Preprocessor macros were used extensively to factor out common declarations.

A further revision to exploit FOG without any other change to functionality forms the basis of the comparison for this example. An implementation based on the use of preprocessor macros is compared with an implementation using meta-functions, meta-variables and derivation rules.

The benefit of using FOG for the 10 non-yacc modules are presented in Table 6.1. The pre-FOG version comprises sources that compact `.cxx` and `.hxx` into a single `.xx` file. Raw (comment and blank included) line counts are presented for these. The post-FOG version comprises a single `.fog` file per module.

Use of (the multi-pass implementation of) FOG reduced the token count by 14%, from 20250 to 17500. The per-module reduction varied between 9% and 48%. The larger reductions occur in small classes, where the benefits of derivation rules and simplification of interface and implementation declarations are most apparent.

module	`.xx`		`.fog`	% reduction
	lines	tokens	tokens	
Burg	1420	8301	7681	9.3
BurgCodeScope	496	2901	2496	14.0
BurgEntry	98	383	227	40.8
BurgNonTerm	204	849	679	20.0
BurgParserValue	78	276	248	10.1
BurgRule	361	1903	1583	16.8
BurgSharedRules	86	327	171	47.7
BurgSubExpr	428	2545	2062	19.0
BurgTerm	163	651	483	25.8
BurgTree	386	2111	1821	13.7
total	3720	20247	17451	14.0

Table 6.1 Token size reduction through use of FOG

A reduction in token count is an easily measured reduction in programming effort. Less easily measured are the more aesthetic improvements of better modularity, improved expression of programming intent, and automatic compliance with programming protocols. A pair of short before/after extracts are therefore provided for readers to make their own judgements. The code is complete save for the removal of 4 functions whose lexical structure exactly duplicates functions that remain. Code for this example is chosen because it is shortest, and so demonstrates the changes more clearly. Providing the large number of unaffected function body lines from a more typical module would not provide extra insight. The definitions of the preprocessor macros or meta-functions is not shown. The two are of comparable lexical size, the meta-function has a higher token count through the use of $ operators and meta-type names, but a lower token count through the use of more appropriate facilities. The meta-functions are modular having fewer interdependencies than the preprocessor macros, and more readable through the use of more conventional structuring and the elimination of back-slash continuation lines.

The original preprocessor macros are almost completely eliminated. The `CUSTOM_RTTI` support is provided automatically by derivation. The remaining 6 macros supporting smart pointers are all subsumed by `MapOfSmartPointerSpecialisations`. Other meta-functions such as `Mutate` just implement simple idioms.

Original interface file:

```
#ifndef ENTRY_HXX
#define ENTRY_HXX
#include <Burg.h>
#include <Id.hxx>                // A smart string class
#include <Object.hxx>
#include <ReferenceCount.hxx>
#include <SmartPointer.H>

class Entry : public Object
{
   CUSTOM_RTTI_WITH_1_BASE_DECLARATION(Entry, Object)
   REFERENCE_COUNT_DECLARATION(Entry)
   NULL_OBJECT_DECLARATION(Entry)
private:
   const Burg& _burg;
   const IdHandle _id;      // Handle for a smart string
private:
   Entry(const Entry&);                         // No copy
   Entry& operator=(const Entry&);              // No assign
protected:
   Entry();
   Entry(Burg& aBurg, const Id& anId);
public:
   const Burg& burg() const { return _burg; }
   const Id& id() const { return *_id; }
   virtual Term *is_term();
   const Term *is_term() const
      { return ((Entry *)this)->is_term(); }
   virtual void mark_reachable();
   virtual ostream& print_this(ostream& s) const;
};
#endif
```

Original implementation file

```
#include <Entry.hxx>
#include <Burg.hxx>
#include <MapOfSmartPointer.H>

CISTOM_RTTI_WITH_1_BASE_IMPLEMENTATION(Entry, Object)
REFERENCE_COUNT_IMPLEMENTATION(Entry)
NULL_OBJECT_IMPLEMENTATION(Entry)
SMART_POINTER_IMPLEMENTATION(Entry)
MAP_OF_SMART_POINTER_IMPLEMENTATION(Entry)

Entry::Entry()
   : _burg(Burg::null_object()) {}

Entry::Entry(Burg& aBurg, const Id& anId)
   : _burg(aBurg), _id(anId) { aBurg.add_entry(*this); }

Term *Entry::is_term() { return 0; }
void Entry::mark_reachable() {}
ostream& Entry::print_this(ostream& s) const { return s << _id; }
```

Revised FOG code, with use of FOG extensions italicised

```
using "Burg.fog";           // Improved form of #include.

class Entry : public Object
{
    using/interface "Burg.h";// Need a #include <Burg.h>
    $NoCopy();              // Section 6.1.2
    $NoAssign();            // Section 6.1.2
    $Mutate();              // Section 6.1.3
private:
    const Burg& _burg = Burg::null_object();
    const IdHandle _id;

protected:
    !inline Entry() {}      // Uses default initialiser value

public:
    const Burg& burg() const { return _burg; }
    const Id& id() const { return *_id; }
    virtual Term *is_term() { return 0; }
    const Term *is_term() const { return mutate().is_term(); }
    virtual void mark_reachable() {}
    virtual ostream& print_this(ostream& s) const
        { return s << _id; }
};

$MapOfSmartPointerSpecialisations(Entry);

protected Entry::Entry(Burg& aBurg, const Id& anId)
    : _burg(aBurg), _id(anId) { aBurg.add_entry(*this); }
```

7 Related Work

7.1 Language Constructs

7.1.1 Syntax

Stroustrup has highlighted the inadequacies of the C preprocessor in [Stroustrup97], where he calls for its eventual demise. Very little work has been published on practical alternatives. Straightforward lexical alternatives such as m4 suffer from many of the same problems by operating independently of the underlying language. Operation in collaboration with C++ involves tackling the challenge of C++ syntax, which is difficult to parse and difficult to extend. It is hard to add new syntax to C++ and so the limited extensions available through meta-functions and meta-variables have an inevitably inferior appearance to a solution that introduces new keywords. Werther [Werther96] provides a sensible proposal for a completely new C++ syntax using more conventional syntactical styles like Ada or Pascal. Within a clean syntactical framework, it would be much easier for researchers to examine alternative syntaxes, and it would be possible for a meta-program to perform syntax extension.

7.1.2 Macros

Macros have a very long history, much of it rather old since macros were important to augment early high level languages. Macros remain essential for assembler programming. [Solntseff74] provides a survey of 22 extensible languages, classifying them as Type A to G according to whether language extension is performed during

- lexical analysis
- syntactic analysis
- parse tree production
- intermediate analysis (tree optimisation)
- code generation
- code conversion

Consideration of code conversion was then purely hypothetical but foreshadows Java load-time activities. No distinction is made for semantic analysis, which makes it difficult to classify FOG, where substitution occurs as trees are incorporated without further analysis during syntactic analysis. Re-use of parse trees corresponds to a Type C extension, but operation during syntactic analysis is Type B.

A different classification of just macros rather than language extension mechanisms is made by [Cheatham66]. Extension may occur

- preceding lexical analysis - text macros
- during syntactic analysis - syntactic macros
- following syntactic analysis - computational macros

Text macros correspond to C preprocessor macros, and computational macros correspond to C++ inline functions and templates. It is syntactic macros that are missing from C++. Since there is no distinction between syntactic and semantic analysis, FOG substitution and prototypes fall tidily into the syntactic macro classification.

An easier to understand classification is based upon the structure of the replacement. [Weise93] identifies

- character-level substitution - text macros
- token-level substitution
- syntax-level substitution - syntax macros
- semantic-level substitution

Token-level substitution as practised by the ANSI C preprocessor occurs between lexical analysis and syntactic analysis. (Or possibly as a late phase of lexical analysis as in Figure 4.4.) This does not quite fit the earlier classifications.

Semantic-level substitution is used to classify the semantically-sensitive macros in [Maddox89]. Lacking any proficiency in Scheme, I fail to understand what is important about this approach. It appears to be a solution to a problem with names being resolved in inappropriate environments, presumably because in a Lisp-derived language, substitution occurs at run-time and there is no distinction between declaration and execution domains. The macros therefore allow environments to be passed around as parameters. This limitation of Lisp-like languages is also addressed by [Hieb92] where problems of losing source context are addressed by hygienic macros. In FOG the situation is rather different. ‘Macro’ operation is occurring at (meta-)compile-time rather than run-time, and in a language that supports hierarchical naming through classes and namespaces. The problems of resolution in the correct environment are addressed by appropriate use of @ or $’s as the substitution operator, and if necessary by passing an appropriate meta-object as a parameter. Source line tracing presents no significant problems for FOG.

The syntactic macro was introduced by [Cheatham66] and [Leavenworth66]. The macro and its arguments have syntactic types corresponding to parts of the language grammar. [Vidart74] gave these concepts a sound and efficient foundation using an Abstract Syntax Tree (AST) to represent them and avoid repeated syntactical analysis of source text.

[Weise93] applies these ideas to ANSI C and exploits a Lisp-like backquote to support a pattern template for substitution, avoiding the need for extensive call trees to rearrange the AST. Weise’s approach is very much an extension to ANSI C introducing new keywords and 9 lexical operators. The approach in FOG is in some ways very similar to Weise’s, however by giving existing concepts a compile-time meaning, and retaining a degree of consistency for all the corollaries, FOG achieves a notation that is more compact, supports character- or syntax-based substitution with only two new lexical operators ($ and @) and no new reserved words. Where Weise needs backquotes and an explicit return statement to activate source-like declarations, FOG just treats all declarations as the return. In FOG, all concepts are put into a C++ Object Oriented perspective.

7.1.3 Joins

Relatively well-behaved coarse-grained merging of application functionality is provided by functions. More flexible but ill-disciplined merging is provided by macros. The BETA fragment system [Knudsen99] provides a finer-grained composition that observes predictable semantics for code fragments that satisfy the attributes of pre-declared slots.

7.1.4 Meta-classes

The concepts of meta-classes were first defined for Smalltalk. Languages such as CLOS have been extended with a MOP (Meta-Object Protocol) [Kiczales91]. Even Java [Gosling97] has a meta-class object for every class. C++ has rather lagged behind, perhaps through a mismatch of the run-time characteristics of traditional

MOPs and the statically compiled philosophy of C++, perhaps through the compiler writer’s desire to prevent further explosion of language complexity. FOG provides statically compiled meta-functionality, which can be used to define customised run-time meta-functionality.

Meta-classes were first introduced to support the configuration of objects at run-time in Smalltalk, and have subsequently become an important part of most Object Oriented languages. Limitations of the Smalltalk implementation led to the development of the simpler Deltatalk [Borning87]. The problems are resolved in the pure Object Model of ObjVlisp [Cointe87]. However, [Maes87] argues that the pure Object Model fails to distinguish the meta-level adequately. The restricted Object Model in FOG does precisely this; only meta-objects exist at compile-time, and only real objects at run-time.

A more versatile Object Model allowing the inheritance of meta-classes to differ from their classes is supported by CLOS and SOM. [Graube89] identified the resulting compatibility problems that arise during traversal of parts of the cycle: class, derived-class, derived-meta-class, meta-class, class. The history of these problems and their solutions in SOM 1, 2, and 2.1 are described in [Danforth94], however the complexity of the solution appears to be worse than the problem. [Bouraqadi-Saâdani98] synthesises additional meta-classes with multiple inheritance to resolve the problems. These problems do not arise in FOG since the two inheritance hierarchies are the same. They also do not arise since there is no object creation at meta-compile time and so no level traversal.

[Briot89] explains the motivation for distinct meta-class inheritance as an ad hoc solution to the propagation problem whereby a concrete class inherits the inappropriate property of abstractness from its abstract base class. This problem does not arise in FOG, since C++ offers at least two distinct solutions to the original problem using pure virtual functions or protected constructors.

The lack of meta-classes has always been a deficiency of C++, for which various proposals were suggested during standardisation. Eventually the standardisation committee compromised on the relatively limited functionality known as Run-Time Type Information (RTTI). A more substantive proposal [Buschmann92] is largely proprietary and so it is difficult to assess accurately. It defines a run-time Meta-Information Protocol providing more extensive data structures with global functions to support iteration. FOG provides a compile-time meta-level, in which application meta-programs may be used to create whatever run-time data structures are appropriate. These may vary from just class names to large descriptive tables for use by a run-time environment that supports run-time meta-programming.

7.2 Meta-level and Reflection

A procedure that manipulates declarations at compile-time might seem to be a simple generalisation of a macro, however harnessing the increased power, provided by this form of self-modifying code, offers ample scope for some fairly difficult papers.

7.2.1 The tower

[Smith84] coined the term reflective for a program that is self-aware. He introduces a minor Lisp variant 2-Lisp that subsumes Scheme, but provides a semantically rationalised notion of evaluation. The 2-Lisp dialect forms a sound foundation using which the reflective 3-Lisp dialect is realised. Further application dialects can then be defined recursively to form the ‘reflective tower’.

The use of a different language dialect at each level of the tower presents challenges to analysis. [Wand88] describes the behaviour without using reflection. Multiple dialects are well-understood in theory, but a major inconvenience in practice. The difficulties are overlooked by many implementations, leading to the problem of meta-circularity when more than one meta-program operates upon the same program. Which meta-program operates first? Does the second meta-program operate on the source or the results of the first meta-program? How does a meta-program behave when it changes it's own declarations? [Chiba96] notes that each level of the reflective tower must exist to define an ordering, so that one meta-program operates consistently on the results of a more nested meta-program and is isolated from its own and less nested meta-programs.

7.2.2 Meta-Object Protocols

A reflective program manipulates the meta-objects that describe the program. The programming interface to these objects is defined by a Meta-Object Protocol such as that provided for CLOS by [Kiczales91].

Each meta-object represents the reification of some programming concept, and so different reflective languages support very different MOPs. This may be as minimal as the RTTI facilities of C++, or as substantial as the meta-computation protocol of [Sobel96].

7.2.3 Languages

OpenC++ version 1 [Chiba93] provided a run-time MOP for C++, through the recognition of comment mark-ups requesting indirection of method calls through meta-objects. This approach was abandoned in favour of a largely compile-time MOP in OpenC++ version 2 [Chiba95], using a two-stage compilation process, first to build an enhanced compiler for the extended language and then to use that compiler. The two stage process avoids the costs of interpreted meta-execution, but unfortunately prevents the realisation of the reflective tower. Although OpenC++ extends most C++ concepts to the meta-level, somehow the language doesn't feel like C++; the extensions are rather haphazard, and much of the meta-programming involves considerable insight into the operation of a compiler. The working representation is a rather strange hybrid between a Lisp-like list and a C++ syntax tree in which punctuation remains significant. This is perhaps attributable to the development path through S++, a form of Scheme supporting C++ concepts.

OpenC++ claims to be based on the principles of the CLOS MOP [Kiczales91], but C++ concepts are so different that it is difficult to see any resemblance. The CLOS MOP was developed for run-time support, and is a natural formalisation of an API that is present anyway. Lists are well-supported by CLOS and so while the list manipulations involved in meta-programming may be difficult for a C++ programmer to understand, they are consistent and compact. When these concepts are transferred to C++, the alien nature of list processing, the consequent lack of language support and the very different C++ perspectives make for an uncomfortable programming environment.

MPC++ [Ishikawa96a] provides a compile-time meta-level that like OpenC++ supports fairly extensive interception of compiler activities and subsequent peek and poke meta-programming, using conventional C++ syntax shifted to the meta-level by `$meta`. MPC++ avoids the problems of the tower by supporting a stack of its output streams, but since these are text rather than syntax-based, they lack the integrity of composition of declarations, statement nesting or character concatenation in FOG.

OpenC++ and MPC++ share the same principle of operation. A meta-program searches for patterns[1] in the code, and modifies the syntax tree by construction of alternate partial trees. The FOG approach is completely different. The pattern

1. Not a design pattern, although the FOG invocation is design pattern-like.

is explicitly identified by the invocation of some declaration (often a meta-function) that implements the pattern. Additional declarations may be composed to elaborate the implementation for more sophisticated requirements. The peek and poke approach is able to achieve most programming goals, but requires the programming skills to peek and poke syntax trees and poses the challenge of recognising all variants of the target pattern. The declarative approach in FOG is less flexible, but only requires relatively conventional skills and may need the interception points to be explicitly identified, albeit by inline function calls. In FOG two declarations are woven by providing the two declarations. The peek and poke approach requires a program to be invoked, the join point to be identified in one tree, and the tree for the other declaration to be hand assembled and merged.

The ease with which OpenC++, MPC++, Sina/St (a precursor to C++/CF) and D (a precursor to AspectJ) could be used is discussed in [Lin99]. All are found lacking for a three-level architecture and so Adapter++ is presented. The problem would appear to be directly soluble with FOG.

Iguana [Gowing96] provides a run-time MOP in which many different activities can be reified on a per-object, per-method or per-class basis. The implementation cost is therefore determined by the required degree of reification. This provides considerable flexibility at run-time but offers little at compile-time, since Iguana is not a reflective complier. [McAffer95] takes reification even further with seven components to a message invocation.

Meta-classes form Java part of the Java language definition, and so there is more language support for reflection and the interesting opportunity to perform meta-programming at load-time. Dalang [Welch98] and Kava [Welch99] provide an ability to intercept method calls. Guaraná [Oliva98] introduces composer meta-objects to enforce composition policies. OpenJava [Chiba98a] migrates the peek and poke concepts of OpenC++ to Java, providing a way of implementing parameterised classes [Chiba98b] that is rather at odds with the more serious language proposals such as for GJ [Bracha98], NextGen [Cartwright98] or [Agesen97]. Java Beans are used as the basis for a two-layer meta-level model by [Wu98], and by [Lorenz98] to provide a reflective implementation of the Visitor pattern.

7.2.4 Applications

Reflection has been used to solve problems in a wide variety of applications.

[Cartwright98] reports on the successful use of OpenC++ to simplify and enforce the interface to an AI library, but only after flattening their meta-program to avoid OpenC++ restrictions.

[Kasbekar98] again using OpenC++, identifies run-time data-dependencies so that a roll-back to a checkpoint can be done efficiently.

[Yokote92] describes the reflective Operating System Apertos.

Further applications involving aspects, communication, constraints, distribution, patterns, persistence and synchronisation are discussed elsewhere.

7.3 Programming Styles

7.3.1 Patterns

The original GoF patterns book [Gamma95] has provoked considerable interest and a growing number of specialised conferences and workshops. Example implementations of the patterns were mostly in C++ but just occasionally in Smalltalk, so translations exist for Java [Grand98] and Smalltalk [Alpert98]. An apparently unhelpful German version exists as well. The POSA book [Buschmann96] provides further fundamental coding patterns.

Unfortunately, from a programmer's perspective, the ability to present any problem as a pattern has broadened the field so that the additional coding patterns contributions provided by the PLOPD conferences [Coplien95b], [Vlissides96], [Martin97] and [Harrison99] are diluted by management and organisational patterns. However, remaining within the programming domain, [Fowler97] covers the rather different perspective required during analysis.

Patterns are very abstract and so pose considerable classification challenges. A pattern is often named rather arbitrarily by its author, and so looking for a pattern that concerns a particular kind of problem is hampered by the lack of clear vocabulary for problem or solution. Two attempts at classifying the basic patterns have been made. [Gil97] distinguishes between clichés (straightforward use of prevalent mechanisms), idioms (a built-in facility in some languages) and cadets (not built-in to any language). [Agerbo98] takes a harsher view of just the GoF patterns, discarding 2 as not patterns at all, 7 as Language-Dependent Design Patterns, and 2 as closely Related Design Patterns: only 12 survive to be classified as Fundamental Design Patterns.

A much more fundamental perspective on patterns is taken by [Pree94], where all collaborations are reduced to seven meta-patterns corresponding to different forms of 1:1 and 1:N relationship. Pree claims a high-level perspective, which is surely wrong: a complete set of object to object relationships is a suitable low-level abstraction that can form one of the layers for a parameterised pattern. The GenVoca approach [Batory97] with its cascade of small orthogonal behaviours would appear to combine well with Pree's meta-patterns, once a suitable tool, such as FOG, is available to glue the behaviours together.

Most attempts to represent patterns in code are informal. Soukup addresses the problems of implementing patterns, with [Soukup95] summarising the much more extensive treatment in [Soukup94]. Soukup's solution, supported by the CodeFarms library, realises each pattern as a data-less class that is declared as a friend of each collaborator. Pattern operations are realised by static member functions of the pattern class, so the programming interface is unnatural: operations are invoked upon the pattern, not the primary collaborator. However, the behaviour is very regular and has extremely good characteristics with respect to include file dependencies. The implementation is hampered by the same limitations that motivated the removal of the One Definition Rule from C++: a special preprocessor has to be used, which could beneficially be replaced by FOG.

An alternative text-based mechanism is provided by SNIP [Wild96], although the enhancements effectively introduce two new languages to implement a kind of marked up source text with a rather accidental semantics for lexical composition.

A reflective implementation of two patterns using OpenJava is described in [Tatsubori98]. Instantiation of a pattern is quite tidy. Definition of a pattern involves a significant amount of peek and poke code and the introduction of reserved words.

The difficulty of actually fielding a reusable implementation of a particular solution is rather neglected. Vlissides participated in the development of a GUI tool to generate code for patterns automatically [Budinsky96], but then expresses considerable reservations in his book [Vlissides98]. It is indeed difficult to conceive an automatic generator that will have sufficient flexibility to balance all the conflicting forces and select the appropriate cookbook implementation. There is rightly much generality and ambiguity in pattern descriptions. However programmers regularly re-use particular pattern implementations with which they are familiar, and providing an improvement over cut and paste for such re-use is essential.

Some experimental work on a GUI-based interface for pattern instantiation in Smalltalk has been reported in [Florijn97] and [Meijers96]. It is difficult to assess quite what has been achieved. Their system seems to exploit the ability of a Smalltalk program to reconfigure dynamically at run-time allowing pattern objects to be cloned into the program interactively.

Early descriptions of patterns identified the participants as the collaborators, which tended to be classes in an implementation. More recent work [Riehle97] and [Riehle98] has concentrated on the different roles that the participants play, and thereby begun to establish a hierarchy in which some more sophisticated patterns are composites of simpler patterns.

Application of a pattern requires roles to be associated with classes. The early descriptions tended to assume that a particular set of roles was performed by each collaborator, blurring the distinction between roles and classes. The role-based perspective provides a more generic insight and offers more opportunity for providing flexible tool support.

Composition of patterns is also addressed by [Lauder98]. A very generalised abstract pattern is identified for decoupled collaborations that can be trimmed to satisfy the more specific behaviour of the Adapter, Facade, Mediator, Observer or Reactor [Schmidt95] patterns.

Although patterns and roles (and aspects) appeared rather late in the evolution of UML, the generalisation of a collaboration diagram described in the User Guide [Booch99] permits parameterised instantiation. This appears to provide the required notational support. Unfortunately the change is minor and recent, and so attracts little attention in the Reference Manual [Rumbaugh99] or the Unified Process [Jacobson99] and temporarily lacks graphical tool support.

An approach to the enforcement of compliance with pattern constraints is provided by [Hedin97a] using attribute extension [Hedin97b] to extend a language grammar to incorporate patterns directly.

Reverse engineering patterns from code is provided by DP++ [Bansiya98]. Heuristics are required to recognise the relatively ill-defined implementations that need to be found, and it is not possible to distinguish structurally similar but operationally different patterns.

Direct generation of code from CAD tools has been an unrealised goal for many years. The flexibility for the invocation of a single meta-function to generate declarations in many classes, and for meta-programming to enforce or create non-trivial program structure may provide the necessary support for a CAD tool that just emits parameterised calls to a suite of meta-functions implemented by a meta-library developer.

7.3.2 Aspects

Traditional programming approaches decompose a problem into functions or objects that often have a direct realisation in an implementation. An alternative decomposition can be made in terms of the concerns, properties or aspects of the system that the programmer must consider in order to satisfy system requirements. Effective decomposition identifies loosely coupled modules that can be implemented and tested independently, but decomposition from one perspective generally destroys modularity from other perspectives. For instance, a concern for error logging may pervade many functions and objects. Since a practical implementation is likely to be a function or object, the problem arises as to how to modularise the error logging aspect. Development and programming of the aspect should be isolated from, but the implementation must be inter-woven with implementation from the more traditional decomposition.

An informal estimate indicates that about 80% of an application is well suited to a structural decomposition, whereas the remaining 20% cuts across this structure and is more appropriately decomposed into aspects or programming concerns.

[Aksit96] discusses the need to maintain separation between the different concerns, and advocates the use of Composition Filters [Glandrup95], which reify the different stages of message passing, so that meta-code can intercept messages according to a variety of source and destination criteria. The practical implementation of C++/CF involves a custom preprocessor of limited capability that imposes considerable restrictions and obligations on the programmer. The greater capabilities of FOG could remove much of the inelegance.

The concepts of aspects have been taken up more generally in the new field of Aspect Oriented Programming. The review of the first ECOOP workshop [Mens97] considered whether AOP was really new or just a more palatable name for meta-programming, given that AOP problems could be solved by reflection. [Kiczales97] provides an extensive discussion of the need for AOP and some interesting examples involving loop fusion, arguing that a meta-programming approach is just a stepping stone. Eventually each aspect could be supported by a customised programming language or environment [Fradet99], [Seinturier99].

Combining the functionality of the aspect code with the non-aspect code requires a mechanism to define how the code should be combined and requires a weaver to perform the composition. [Ossher98] discusses the join points where composition occurs and highlights the dangers imposed by the invasive characteristics of extra Aspect-Oriented code in comparison to the additive characteristic of Object-Oriented code. In a rather different field, [Mulet95] describes constraints upon the composition of functions so that composition occurs predictably via nested invocation rather than ad hoc cut and paste.

AspectJ [Lopes98] provides the flexibility for Java programmers to make controlled additions to classes via introduce and advise weaves. An aspect is introduced into Java as an instantiable entity, providing the flexibility to dynamically associate aspects and objects at run-time. This provides direct language support for a particular and certainly useful pattern, and by introducing the support at load-time[2] rather than compile-time is able to do so without specifying detailed implementation semantics. The aspect syntax forces tight coupling to the application, and so [Beaugnard99] suggests a relaxation to allow aspect, join and application to be independent.

FOG provides more extensive and more varied facilities than AspectJ, but needs to resort to meta-programs to implement the limited capabilities directly available in AspectJ.

The flexibility of load-time weaving is also exploited by [Welch99]. Run-time weaving with support for aspects at the meta-level is advocated in [Böllert99], [Lunau98] and [Pryor99]. A more pragmatic compromise in which some aspects are statically woven at compile-time while others are retained at run time is suggested by [Matthijs97].

A slightly different take on programming concerns arises in Subject Oriented Programming [Harrison93], where the differing perspectives of the same objects appropriate to different applications is considered. The concepts of different views of an object provided by CV++ [Shilling89] are taken much further so that each application may be written with its own subjective view of each object. When such objects are shared between subsystems or applications, an update to one subject must make consistent changes to the underlying object and alternate subjective views. Composition policies and language extensions to achieve this are described by [Ossher95].

2. The current implementation weaves at compile-time.

7.3.3 Generative Programming

Generative Programming [Czarnecki97] seeks to provide highly configurable components that can be combined and optimised at compile-time so that minimal overheads are incurred by an application.

An early perspective into reusable components is provided by [Batory92]. Two independently developed domain-specific module generators, Genesis and Avoca, were found to have very similar design and implementation. The common concepts were combined to give the GenVoca principles of composing very thin fairly orthogonal layers to create a desired component. In [Batory93] these concepts are applied to C++ libraries, resulting in fewer source concepts to generate more, smaller and faster library components than standard C++ libraries. Further improvements in speed and flexibility are reported in [Batory94], using a succession of customised preprocessors and compilers: P1, P2 leading to P++ that adds support for `realm` and `component` to C++ [Singhal96]. Composition of components is based on a realm of interchangeable components with a common interface. The realm may therefore be used as a type signature. However, in practice, not all components are completely interchangeable, there may be constraints on, or prerequisites for, the ordering of compositions. [Batory97] identifies the need for upward and downward checking of constraints, using pre/post-conditions/restrictions. A related implementation of container libraries using template meta-programming is described in [Czarnecki99].

A direct form of generative programming is possible with imaginative use of C++ templates. [Myers95] describes the concepts of traits classes (template classes of parameters), which are used to pass a set of parameters and interrelationships as a single template parameter, thereby considerably simplifying the instantiation interface. Inference of expression types at compile-time is exploited in [Veldhuizen94] to generate customised inline functions that outperform conventional library implementations. Control structures are realised by recursive template instantiation in [Veldhuizen95] supporting generation of more sophisticated customised inline functions for sine, FFT or bubble sort.

The practise of composition of behaviour using mix-ins is reported to incur a potentially exponential growth in the length of template class names in [Smaragdakis98], where an extra outer mix-in layer is introduced to resolve the problem.

The GenVoca approach seems well suited to the implementation of efficient reusable components from very simple building blocks. Template meta-programs provide effective techniques for composing the building blocks at compile-time, provided the composition results in a function or type. Unfortunately interesting components are more complicated. [Eisenecker97] suggests that GP subsumes AOP, but the template approach requires aspect functionality to use pre-existing parameterisation, rather than an independently developed weave. The facilities of FOG are required to compose more general declarations.

7.4 Applications

C++ has been extended in minor ways by practical compilers [Stallman98] and [Microsoft97], and a few isolated language extensions such as [Baumgartner97] have been published. Researchers in many fields have chosen to use C++, but found it inadequate for their purposes. There are therefore many domain specific extensions to C++, just some of which are mentioned here.

7.4.1 Design by contract

Design by contract advocates the use of pre- and post-conditions and invariants to define the behaviour of components, and it is beneficial for these properties to be expressed in implementation code.

Support for contracts is an integral part of Eiffel [Meyer92]. It has to be added for C++.

A++ [Cline90] extends C++ in a fairly natural way to support class assertions and invariants, which can in principle be optimised at compile-time.

CCEL [Duby92] adds a form of meta-programming using assertions in a predicate calculus so that constraints can be validated.

[Porat95] proposes some language extensions to support pre-conditions, post-conditions and invariants.

7.4.2 Persistence and Marshalling

A low level understanding of object layout is necessary for persistent storage of objects in databases or for marshalling objects whether for signalling between nodes in a communication network or distribution between nodes in a parallel processor.

Persistence is commonly supported by an extended language adding a `persistent` keyword to C++ as in OQL or E [Vemulapati95], or a replacement allocator such as pnew in O++. An alternative approach is taken by [Park96] using an object pre-header at negative address offset, so that persistent objects are interchangeable with non-persistent objects. A MOP approach is recommended by [Stroud94] to avoid the inflexibility imposed by the extended language PC++. A simple reflective system based upon the Java API is described by [Lee98].

Wilson and Lu [Wilson96] provides extended articles by 16 of the leading research teams using C++ for parallel processing. Some researchers used only library classes and run-time support code, and so remain entirely within the normal confines of the C++ language. Others introduce language extensions, which are variously implemented as translators to C++, or modified C++ compilers. MPC++ [Ishikawa96b] exploits meta-level facilities to support an extended syntax within a “standard” C++ compiler. Many of the C++ extensions appear unnecessary and some authors recognise that more imaginative use of C++ facilities, particularly those not readily available at the start of their research could have reduced the need for divergence.

Domain specific extensions, when fully integrated with C++, can provide a clean solution to the domain problem. However, many extensions are poorly integrated because of the size and complexity of C++ and so provide little more than a research tool. Many of the problems dealt with in a domain-specific fashion can be resolved in a domain-independent way by using the meta-level programming facilities of FOG. However FOG meta-programming is restricted to declarations and so the more radical changes of C** [Larus96] in which data parallel semantics are introduced to expressions could probably not be addressed.

The different approaches demonstrate that language extension can occur at three different levels:

- Library classes and run-time environments can be developed without any language or compiler changes. FOG’s increased capabilities at compile-time provide library developers with more options, perhaps supporting simpler interfaces, reduced requirements for user support code, or stronger compile-time detection of protocol violations.
- Translators that recognise one or two extra reserved words require development of the translator but do not affect the underlying compiler. FOG provides an ability to introduce custom extensions to C++, enabling

many of the characteristics of custom translators to be achieved by a general-purpose translator.

- New forms of code generation necessitate significant revision to both language and compiler. FOG offers very little to applications that need to rewrite the basic compiler.

7.4.3 Synchronisation

Synchronisation is critical for reliable multi-process or multi-processor applications. Concurrent access to data has to be restricted, a problem resolved in principle by monitors [Hoare74]. In practice there are typographical difficulties in ensuring that monitor protocols are not accidentally bypassed and genuine difficulties in ensuring that a synchronisation policy is sensibly applied by derived classes. Alternative strategies are considered in [Matsuoka93]: reflection is considered necessary to solve the inheritance anomaly and is exploited by [Stroud95].

7.5 Summary

The need for many different research teams to develop customised variants of C++ demonstrates the need to be able to extend C++ to support new domains.

Research in the fields of patterns and Generic Programming shows an increasing ability to structure large software components from smaller ones.

Aspect Oriented Programming demonstrates the need to combine relatively independent software modules into a composite whole.

FOG provides facilities to assist in all these areas. Unfortunately, it is difficult to answer the critique that C++ is too large, and that adding meta-functionality is an enhancement too far. However, it is also difficult to avoid recognising that the absence of meta-functionality is restrictive for some domains and an inhibition to re-use for all.

8 Summary

The many achievements of FOG will now be reviewed before highlighting what remains to be done, and the limitations upon what can sensibly be done. The relevance to other languages will then be discussed before finally concluding with a brief summary of how the problems with C++ described in the introduction have been resolved.

8.1 Novelty

There are few ideas in software engineering that are totally new. Most are the result of a revision or combination of prior work. The functionality of FOG combines concepts from many areas, adapting them to fit the philosophical, semantic and syntactic constraints of the C++ language. The combination is certainly novel.

8.1.1 Scoped preprocessing

Macros and preprocessing are a neglected, perhaps scorned, field in software engineering. Little work has been done and no work that considers macros within the hierarchical context of C++. Resolution of macro-names with a prevailing scope, with the consequent benefits that can accrue from isomorphism and inheritance is new in FOG.

8.1.2 Deferred substitution

Resolution of names within the correct name-space at the correct-time is a traditional concern of language designers and consequently programmers. The functional argument (funarg) problem in Lisp demonstrates the problems of avoiding name capture. The distinction between ' and & substitution operators in VAX/VMS DCL show the need to control resolution time. FOG applies related concepts to substitution within meta-programs through the $ and @ operators.

8.1.3 Context-free syntactical C++ parsing

Processing C++ declarations before their semantics has been determined necessitates context-free parsing. A clear distinction between syntax and semantics is not normally made because C++ is perceived to be inherently context-dependent requiring lexical, syntactic and semantic analyses to be tightly coupled and consequently to blur the distinctions. The official “(informative)” grammar provides a mixture of lexical, syntactic and a few semantic rules. The main body of the standard does not always distinguish whether described constraints are syntactic or semantic. Tradition therefore perpetuates the perception that parsing must be difficult.

FOG draws a pragmatic distinction between syntax and semantics. Syntax is what can be analysed by an LALR(1) parser such as yacc. Semantics is what has to be analysed later.

Examination of the grammar shows that syntactic analysis without type information causes only a minor ambiguity for expressions using casts. The ambiguity is entirely deterministic and readily deferred for resolution at the post-yacc semantic level.

Accurate syntactic analysis without template information is impossible. However, an iteration through all alternatives of a template < / arithmetic < ambiguity can be performed to determine a consistent, but not necessarily correct, syntactic analysis. Instrumentation of practical programs shows that approximately 1% of statements contain a template ambiguity, and that for approximately 1% of those statements, the consistent parse is incorrect. A syntactically consistent parse is therefore possible without template information, subject to the requirement that the semantic processing must repair the incorrect parse for approximately 0.01% of statements.

8.1.4 Back-tracking in yacc

LALR parsers such as yacc do not support back-tracking unlike their LL counterparts. Ambiguity problems that cannot be resolved within the grammar need assistance from a separate lookahead parser. Implementation of back-tracking within yacc proves to be fairly straightforward enabling lookahead parsing to be performed within yacc. The ambiguity between arithmetic and template interpretation of an < exploits this back-tracking technique to perform a binary tree search to identify a syntactically consistent parse.

8.1.5 Superset grammar

Traditional C++ grammar implementations attempt to maximise the semantic resolution of the yacc grammar, since this minimises subsequent coding. This is also motivated by the need to incorporate a large amount of semantic intelligence to resolve recursive declaration/expression ambiguities accurately.

Context-free parsing makes resolution of declaration/expression ambiguities impossible. The ambiguity must be deferred for semantic resolution rather than attempt to resolve it syntactically.

The superset FOG grammar recognises that the declaration/expression ambiguity derives from the fundamental C language design: declarations should mimic their usage in expressions. The ambiguity is therefore not a series of inconvenient barriers to be surmounted, but rather a series of partially overlapping sub-syntaxes. A generalisation of declaration and expression and a few other productions is used to perform context-free parsing in the superset grammar.

8.1.6 Semantic semantic analysis

Removal of semantic considerations from the syntactic analysis considerably simplifies the grammar, but requires additional semantic processing. However, this processing is making the same decisions as before, but in the controlled context of an AST rather the difficult partial environment offered by reduction actions while avoiding shift-reduce conflicts.

Resolution of the declaration/expression and related conflicts at the semantic level involves a straightforward but not quite trivial dataflow algorithm propagating a bit-mask of satisfied semantic hypotheses from the leaves of the AST to the root, where any residual ambiguity can be resolved by applying the defined ambiguity resolution rules. The propagation makes use of type information, tree structure and associated semantic constraints in determining whether a hypothesis such as is-parameter-declaration is satisfied.

Deferring the ambiguity aids error diagnosis as well. An error during syntactic processing indicates that the syntactic analysis has failed to understand the token sequence and as a result an error recovery mechanism must be invoked to resynchronize. Since analysis failed, it is often difficult to make a better error diagnosis than “syntax error near line *x*”. The more general syntax to defer the ambiguity accepts many sentences corresponding to simple typographical programming errors with the result that the syntactical analysis does not lose synchronisation, and so a more appropriate error diagnostic can be produced.

8.1.7 Extended regular expressions

Demonstrating that the superset grammar covered the existing syntax required analysis of the C++ grammar. An extended form of regular expression was introduced to describe sentences of C++ enabling the traditional ambiguities to be deduced and the superset justified.

8.1.8 Polymorphic syntax

Syntax macros are traditionally syntax-driven: the known syntactical requirements of the macro (a meta-function or meta-variable in FOG) are used to guide the syntactic analysis. This introduces a context-dependency of the syntactic analysis upon the semantically defined macros, unless an extra-lingual mechanism such as `#define` is used to define macros. It also introduces a data-dependency requiring the course of syntax analysis to respond to the value of a meta-program variable rather than the local source token.

The superset grammar unifies many C++ constructs and the approach is extended to define a production that encompasses almost the entire C++ grammar, enabling a syntax independent and consequently context-free parse of meta-function arguments and meta-variable initializers. Syntactic analysis of meta-functions and meta-variables is therefore context-free in FOG. Syntactic analysis of syntax macros requires dynamic changes to the table of reserved words and a data-dependency upon the number but not type of parameters.

This is only novel within the context of C++. In a language with a clean syntax a generic parse should be trivial, but still worth implementing to remove context-dependency.

8.1.9 Literal source

Meta-code surrounds ordinary statements and declarations so that there is no need for special syntax or procedures to define source syntax literals.

The source syntax is its own literal.

```
auto if (statistics_gathering_enabled)
    log_successful_call();
```

and an entire function body is returned

```
auto declaration declare_pointer_classes(identifier aClass)
{
    class $aClass;
    typedef PointerTo<$aClass> ${aClass}Pointer;
    typedef PointerTo<const $aClass> ${aClass}ConstPointer;
}
```

8.1.10 Potential and Actual

Meta-programming is traditionally practised in Smalltalk and Lisp-like languages, where meta-programming occurs at run-time re-using functionality necessary to establish an Object Oriented execution environment. More recently meta-programming has been possible in Java at run-time and rather more interestingly and uniquely at load-time.

Compile-time (or static) meta-programming is not widely used since it is only available in research languages such as OpenC++ or MPC++. These languages support programmed manipulation of the declaration pool.

FOG with its syntactic support for source literals introduces the distinction between potential declarations and actual declarations, allowing meta-programs to operate consistently on declarations with determined or undetermined scopes.

8.1.11 Derivation rules

Derivation rules are perhaps just a little bit of syntactic sugar to simplify meta-programming. However many realistic problems involve a policy that has to be observed by classes within an inheritance hierarchy. This requirement is captured directly by derivation rules. Related work on automatic generation of code appears to concentrate more on resolution of composition conflicts.

8.2 Further Work

The version of FOG currently available on the net uses the multi-pass grammar approach. The more stream-lined and efficient approach supported by the superset grammar is still under development.

FOG is currently written in C++ using a trivial custom preprocessor that just splits interface and implementation from a single file and performs code synthesis only for include files and their guards. The source code for FOG should be revised to exploit FOG functionality, and thereby demonstrate and test the use of FOG more convincingly. A measurement of the lexical source size reduction should show how beneficial FOG is for large programs with deep inheritance hierarchies.

C++ is a large language, which FOG should as a minimum parse and emit unchanged. FOG has a useful degree of functionality in many areas, however practical experience with interesting small examples tends to encounter unimplemented or misimplemented functionality.

Use in a diverse range of applications needs to be assessed and a meta-library of common utilities developed.

8.3 Limitations

FOG operates as a translator to C++ and so necessarily precedes C++ compilation. FOG cannot operate on actual compilation results, only upon predictions of those results. This has two consequences.

It is not guaranteed that FOG sees the final state of declarations. In a multi-session compilation, a class may appear to be a leaf class in one session, but further derived classes may exist in other sessions. Decisions predicated on leaf-ness will therefore be in error. A complete fix of this problem requires global knowledge. Detection of the anomaly can be resolved as described in Section 5.6.8.

Template instantiation occurs during or after compilation. It is therefore difficult for FOG to know which parameter combinations will be used, or to detect which member functions will actually be required. In the general case where FOG is used to prepare library code, FOG cannot know what the instantiations will be. It is therefore impossible for meta-programs to manipulate template instantiations usefully. Meta-programming in FOG is limited to manipulation of template declarations.

8.4 Other Languages

Although the work described in this thesis is primarily concerned with resolving deficiencies in the use of C++, the work is of greater applicability, mainly to languages that involve significant compilation activity such as Eiffel, Ada or Java. Introduction of extra compilation stages is inappropriate for languages such as Smalltalk or CLOS where object structure is defined at run-time.

The distinction between potential and actual declarations, the concept of derivation rules and a flexible substitution based upon tree-literals combined with a lexical concatenation are not specific to C++, although some of the detailed syntactical issues are. Implementation of these concepts in other languages is likely to be a little simpler, since few other modern languages have quite such a challenging syntax as C++.

The observation that the One Definition Rule is a major hurdle to implementation of patterns and Aspect Oriented Programming is again applicable to all languages. Language designers should endeavour to support interleaved declarations.

Programming involves repetition at many levels, and programmers naturally seek to factor the repetition into some parameterisable reusable construct, which may be a loop, subroutine, class, template, macro, file or library. Omission of any of these capabilities simplifies a language, but limits the programmer's or the program's efficiency. Some form of macro to perform lexical processing and meta-programming is therefore beneficial to all languages, although the precise syntax must be carefully chosen to fit within the traditional style of each language.

8.5 Resolution of Goals

The introductory discussion highlighted problems that arise with C++. The way in which these are resolved in FOG will be summarised.

Cpp should be replaced rather than eliminated.

Object-like and function-like macros are replaced by meta-variables and meta-functions.

Substitution by imposition is replaced by a substitution invited by $ or @.

## and # are replaced by adjacent lexical element concatenation.

Conditional processing is replaced by meta-programming.

Compile-time programming is necessary to configure declarations.

Reflection is useful for simple applications.

Reflection is almost essential for sophisticated applications.

Meta-programs can manipulate declarations.

Patterns and AOP require weaving.

The One Definition Rule must be circumvented.

Interleaved declarations should be allowed.

The One Definition Rule has been relaxed to allow declarations to be introduced outside the confines of class braces. This supports interleaved declarations and weaving within classes. Multiple contributions to the same declaration are composed, supporting weaving of individual declarations.

Lexical redundancy should be eliminated.

The need for distinct interface and implementations has been removed.

Derived code can reuse inherited declarations.

Predictable code should be provided automatically.

Derivation rules support automatic generation of derived code.

Meta-programs can generate code for more specialised applications.

A concept should be instantiated by a single invocation.

Invocation of a meta-function can provide complete instantiation, exploiting composition to inject code as appropriate.

9 Glossary

9.1 Acronyms

AI	Artificial Intelligence
ANSI	American National Standards Institute
AO(P)	Aspect-Oriented (Programming)
ARM	Annotated Reference Manual [Ellis90]
AST	Abstract Syntax Tree
BNF	Backus-Naur Form
BURG	Bottom-Up Rewrite Generator
CAD	Computer Aided Design
CFG	Context-Free Grammar
CORBA	Common Object Request Broker Architecture
Cpp	C preprocessor
CRC	Cyclic Redundancy Check
DFA	Deterministic Finite Automaton
DSP	Digital Signal Processor/Processing
FFT	Fast Fourier Transform
FOG	Flexible Object Generator (in this thesis) Fragmented Object Generator (in [Gourhant90])
GNU	GNU is Not Unix
GoF	Gang of Four book [Gamma95]
GP	Generative Programming
GUI	Graphical User Interface
LALR(k)	Look-Ahead parsing based on Left-to-right scanning of the input, with Right-most derivation in reverse, using k input symbols of lookahead.
LL(k)	Parsing based on Left-to-right scanning of the input, with Left-most derivation, using k input symbols of lookahead
LR(k)	Parsing based on Left-to-right scanning of the input, with Right-most derivation in reverse, using k input symbols of lookahead
MOP	Meta-Object Protocol
NFA	Non-deterministic Finite Automaton
ODR	One Definition Rule (§3.2)
OO(P)	Object-Oriented (Programming)
RTTI	Run-Time Type Information
SO(P)	Subject-Oriented (Programming)
UML	Unified Modeling Language
VLIW	Very Long Instruction Word
yacc	yet another compiler compiler

9.2 Terms

grammar	The composite syntactical definition of a language, comprising many (production) rules and one distinguished non-terminal.
lexeme	Synonym for terminal or token.
lexical analysis	Analysis determining a lexeme or token from a source character sequence.
meta-	Prefix denoting elevation of a run-time concept to compile-time.
meta-base (class)	Meta-class from which another meta-class inherits.
meta-class	Class that describes a class.
namespace	The specific form of name-space established by a C++ namespace.
name-space	Any context in which names may be resolved.
non-terminal (token)	A term in a production rule defined by the left-hand side of one (or more) production rules.
production (rule)	Rule describing the grammatical equivalence of a left-hand side non-terminal with a sequence of right-hand side terminals and non-terminals. Multiple rules sharing a common left-hand side are often loosely referred to as a single production.
reduction (rule)	Synonym for production (rule).
reflect(ion)	Inspection of a program by itself.
root class	The least derived class in an inheritance hierarchy.
root scope	The least derived scope associated with a derivation rule.
rule	See production (rule).
semantic analysis	Analysis determining whether a (syntactically valid) sentence satisfies semantic constraints.
sentence	A sequence of source tokens generally satisfying some syntax.
syntactical analysis	Analysis determining whether and in what way a source sentence satisfies a grammar.
syntax	A specific subset of a grammar.
terminal (token)	An element in a production rule directly corresponding to a product of lexical analysis.
token	See terminal.

10 References

[Agerbo98] Ellen Agerbo and Aino Cornils. *How to preserve the benefits of design patterns.* Proceedings of the 1998 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications, OOPSLA'98, ACM SIGPLAN Notices, vol. 33, no. 10, 134-143, October 1998.

[Agesen97] Ole Agesen, Stephen N. Freund and John C. Mitchell. *Adding Type Parameterization to the Java Language.* Proceedings of the 1997 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications, OOPSLA'97, ACM SIGPLAN Notices, vol. 32, no. 10, 49-65, October 1997.

[Aho86] A.V. Aho, R. Sethi and J.D. Ullman. *Compilers: Principles, Techniques and Tools*. Addison-Wesley, 1986.

[Aksit96] Mehmet Aksit. *Composition and separation of concerns in the Object-Oriented model.* ACM Computing Surveys, vol. 28A, no. 4es, 148, December 1996. http://www.acm.org/pubs/citations/journals/surveys/1996-28-4es/al48-aksit/al48-aksit.html

[Alpert98] Sherman R. Alpert, Kyle Brown and Bobby Woolf. *The Design Patterns, Smalltalk Companion*. Addison-Wesley, 1998.

[Bansiya98] Jagdish Bansiya. *Automating design-pattern identification*. Dr. Dobb's Journal, 20-22,24,26,28, June 1998.

[Batory92] Don Batory and Sean O'Malley. *The Design and Implementation of Hierarchical Software Systems with Reusable Components.* ACM Transactions on Software Engineering and Methodology, vol. 1, no. 4, 355-398, October 1992. ftp://ftp.cs.utexas.edu/pub/predator/tosem-92.ps

[Batory93] Don Batory, Vivek Singhal, Marty Sirkin and Jeff Thomas. *Scalable Software Libraries.* Proceedings of the Symposium on the Foundations of Software Engineering, ACM SIGSOFT'93, 191-199, December 1993. ftp://ftp.cs.utexas.edu/pub/predator/sigsoft-93.ps

[Batory94] Don Batory, Vivek Singhal, Jeff Thomas, Sankar Dasari, Bart Geraci and Marty Sirkin. *The Gen Voca Model of Software-System Generators*. IEEE Software, vol. 11, no. 5, 89-94, September 1994. ftp://ftp.cs.utexas.edu/pub/predator/ieee_sw-94.ps

[Batory97] Don Batory and Bart J. Geraci. *Composition validation and subjectivity in GenVoca Generators.* IEEE Transactions on Software Engineering, vol. 23, no. 2, 67-82, February 1997.

[Baumgartner97] Gerald Baumgartner and Vincent F. Russo. *Implementing signatures for C++*. ACM Transactions on Programming Languages and Systems, vol. 19, no. 1, 153-187, January 1997. http://www.acm.org/pubs/citations/journals/toplas/1997-19-1/p153-baumgartner

[Beaugnard99] Antoine Beugnard. *How to make aspects re-usable, a proposition.* Position paper for *Aspect-Oriented Programming Workshop at the European Conference on Object-Oriented Programming (ECOOP'99),* June 1999. http://wwwtrese.cs.utwente.nl/aop-ecoop99/papers/beugnard.pdf

[Bjarnason97] Elizabeth Bjarnason. *Tool support for framework-specific language extensions.* In *Proceedings of the Language Support for Design Patterns and Frameworks Workshop at ECOOP'97*. In *Workshop Reader of the European Conference on Object-Oriented Programming (ECOOP'97),* LNCS 1357, Springer-Verlag, June 1997. http://bilbo.ide.hk-r.se:8080/~bosch/lsdforg/bjarnason.ps

[Böllert99] Kai Böllert. *On Weaving Aspects.* Position paper for *Aspect-Oriented Programming Workshop at the European Conference on Object-Oriented Programming (ECOOP'99),* June 1999.
http://wwwtrese.cs.utwente.nl/aop-ecoop99/papers/boellert.pdf

[Booch91] Grady Booch. *Object-Oriented Analysis and Design with Applications*. Benjamin/Cummings, 1991.

[Booch94] Grady Booch. *Object-Oriented Analysis and Design with Applications, Second Edition*. Benjamin/Cummings, 1994.

[Booch99] Grady Booch, James Rumbaugh and Ivar Jacobson. *The Unified Modelling Language User Guide*. Addison Wesley, 1999.

[Borning87] A. Borning and T. O'Shea. *Delta Talk: An Empirically and Aesthetically Motivated Simplification of the Smalltalk-80 Language.* In J. Bezivinin and P. Cointe. *Proceedings of the European Conference on Object-Oriented Programming ECOOP'87*. LNCS 276, 1-10, Springer, June 1987.

[Bouraqadi-Saâdani98] Noury M.N. Bouraqadi-Saâdani, Thomas Ledoux and Fred Rivard. *Safe metaclass programming.* Proceedings of the 1998 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications, OOPSLA'98, ACM SIGPLAN Notices, vol. 33, no. 10, 84-96, October 1998.

[Bracha98] Gilad Bracha, Martin Odersky, David Stoutamire and Philip Wadler. *Making the future safe for the past: Adding genericity to the Java programming language.* Proceedings of the 1998 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications, OOPSLA'98, ACM SIGPLAN Notices, vol. 33, no. 10, 183-200, October 1998.

[Briot89] Jean-Pierre Briot and Pierre Cointe. *Programming with Explicit Meta-classes in Smalltalk-80.* Proceedings of the 1989 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications, OOPSLA'89, ACM SIGPLAN Notices, vol. 24, no. 10, 419-431, October 1989.

[Budinsky96] F.J. Budinsky, M.A. Finnie, J.M. Vlissides and P.S. Yu. *Automatic code generation from design patterns*. IBM Systems Journal, vol. 35, no. 2, 151-171, 1996.
http://www.research.ibm.com/journal/sj/budin/budinsky.html

[Buschmann92] Frank Buschmann, Konrad Kiefer, Frances Paulisch and Michael Stal. *A Run-Time Type Information System for C++*. Object-Oriented Reflection and Metalevel Architectures Workshop at the 6th European Conference on Object-Oriented Programming, ECOOP'92, June 1992.
http://jerry.cs.uiuc.edu/reflection/utrecht/paulisch.ps

[Buschmann96] Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad and Michael Stal. *Pattern-Oriented Software Architecture: A system of patterns*. John Wiley, 1996.

[C89] ANSI X3.159-1989*. American National Standard for Information Systems - Programming Language - C.* American National Standards Institute, 1989.

[C++96] ANSI X3J16/96-0225. *Working Paper for Draft Proposed International Standard for Information Systems - Programming Language - C++*. American National Standards Institute, 1996 (also known as Committee Draft 2).
http://www.maths.warwick.ac.uk/cpp/pub/dl/cd2/CD2-PDF.tar.Z.

[C++98] ISO/IEC 14882:1998(E). *International Standard - Programming languages - C++*. American National Standards Institute, July 1998.

[Cartwright98] Robert Cartwright and Guy L. Steele Jr. *Compatible Genericity with Run-time Types for the Java^{TM} Programming Language.* Proceedings of the 1998 ACM SIGPLAN Conference on Object-Oriented Programming

Systems, Languages and Applications, OOPSLA'98, ACM SIGPLAN Notices, vol. 33, no. 10, 201-215, October 1998.

[Cavarroc98] Joël Cavarroc, Sabine Moisan and Jean-Paul Rigault. *Simplifying an Extensible Class Library Interface with OpenC++.* Proceedings of the Workshop on Reflective Programming in C++ and Java at the 1998 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications, OOPSLA'98. UTCCP Report 98-4, Center for Computational Physics, University of Tsukuba Japan, ISSN 1344-3135, October 1998.
http://www.softlab.is.tsukuba.ac.jp/~chiba/oopsla98/proc/jpr.pdf

[Cheatham66] T.E. Cheatham. *The introduction of definitional facilities into higher level programming languages*. Proceedings of the 1966 Fall Joint Computer Conference, AFIPS, vol. 29. 623-637, 1966.

[Chiba93] Shigeru Chiba and Takashi Masuda. *Designing an Extensible Distributed Language with a Meta-Level Architecture.* Proceedings of the 7th European Conference on Object-Oriented Programming, ECOOP'93, LNCS 707, 482-501, July 1993.
http://www.softlab.is.tsukuba.ac.jp/~chiba/pub/chiba-ecoop93.ps.gz

[Chiba95] Shigeru Chiba. *A Metaobject Protocol for C++.* Proceedings of the 1995 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications, OOPSLA'95, ACM SIGPLAN Notices, vol. 30, no. 10, 285-299, October 1995.
http://www.softlab.is.tsukuba.ac.jp/~chiba/pub/chiba-oopsla95.ps.gz

[Chiba96] Shigeru Chiba, Gregor Kiczales and John Lamping. *Avoiding Confusion in Metacircularity: The Meta-Helix.* Proceedings of the 2nd International Symposium on Object Technologies for Advanced Software, ISOTAS'96, LNCS 1049, 157-172, 1996.
http://www.softlab.is.tsukuba.ac.jp/~chiba/pub/chiba-isotas96.ps.gz

[Chiba98a] S. Chiba and M. Tatsubori. *Yet Another java.lang.Class.* In *Proceedings of the Reflective Object-Oriented Programming and Systems Workshop at ECOOP'98*. In *Workshop Reader of the European Conference on Object-Oriented Programming (ECOOP'98),* LNCS 1543, Springer-Verlag, June 1998.
http://www.softlab.is.tsukuba.ac.jp/~chiba/pub/chiba-ecoop98ws.ps.gz

[Chiba98b] Shigeru Chiba. *Javassist - A Reflection-based Programming Wizard for Java.* Proceedings of the Workshop on Reflective Programming in C++ and Java at the 1998 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications, OOPSLA'98. UTCCP Report 98-4, Center for Computational Physics, University of Tsukuba Japan, ISSN 1344-3135, October 1998.
http://www.softlab.is.tsukuba.ac.jp/~chiba/oopsla98/proc/chiba.pdf

[Cline90] Marshall P. Cline and Doug Lea. *The Behavior of C++ Classes.* Proceedings of the Symposium on Object-Oriented Programming Emphasizing Practical Applications, SOOPPA, 81-91, September 1990.
http://gee.cs.oswego.edu/pub/papers/A++-sooppa.ps

[Coëtmeur93a] Alain Coëtmeur. *bison++ 1.21-7 based on bison 1.21.* 1993.
ftp://ftp.cdrom.com/pub/cug/vol_400/404_01.zip (PostScript)
ftp://ftp.cdrom.com/pub/cug/vol_400/404_02.zip (sources)

The 1.21-8 version used for developing FOG is no longer available on the web, but is part of the FOG distribution in
http://www.ee.surrey.ac.uk/Resaearch/CSRG/fog

[Coëtmeur93b] Alain Coëtmeur. *flex++ 2.3.8-5 based on flex 2.3.8.* 1993.
ftp://ftp.cdrom.com/pub/cug/vol_400/405_01.zip (sources)
ftp://ftp.cdrom.com/pub/cug/vol_400/405_02.zip (PostScript)

The 2.3.8-7 version used for developing FOG is no longer available on the web, but is part of the FOG distribution in http://www.ee.surrey.ac.uk/Resaearch/CSRG/fog

[Cointe87] Pierre Cointe. *Metaclasses are First Class: The ObjVlisp Model.* Proceedings of the 1987 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications, OOPSLA'87, ACM SIGPLAN Notices, vol. 22, no. 12, 156-167, December 1987.

[Coplien92] James O. Coplien. *Advanced C++ Programming Styles and Idioms*. Addison-Wesley, 1992.

[Coplien95a] James O. Coplien. *Curiously recurring template patterns*. C++ Report, 24-27, February 1995. Reprinted in [Lippman98].

[Coplien95b] James O. Coplien and Douglas C. Schmidt. *Pattern Languages of Program Design*. Addison-Wesley, 1995.

[Cox86] Brad J. Cox. *Object Oriented Programming: An Evolutionary Approach*. Addison-Wesley, 1986.

[Czarnecki97] Krzysztof Czarnecki, Ulrich W. Eisenecker and Patrick Steyaert. *Beyond Objects: Generative Programming.* In Cristina Lopes, Kim Mens, Bedir Tekinerdogan and Gregor Kiczales. *Proceedings of the Aspect-Oriented Programming Workshop at ECOOP'97*. In *Workshop Reader of the European Conference on Object-Oriented Programming (ECOOP'97),* LNCS 1357, Springer-Verlag, June 1997. http://www.parc.xerox.com/spl/projects/aop/ecoop97/aop-ecoop97-proceedings.pdf

[Czarnecki99] Krzysztof Czarnecki and Ulrich W. Eisenecker. *Synthesizing Objects.* Proceedings of the 13th European Conference on Object-Oriented Programming(ECOOP'99), 18-42, LNCS 1628, Springer, June 1999.

[Danforth94] Scott Danforth and Ira R. Forman. *Reflections on Metaclass Programming in SOM.* Proceedings of the 1994 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications, OOPSLA'94, ACM SIGPLAN Notices, vol. 29, no. 10, 440-452, October 1994.

[Duby92] Carolyn K. Duby, Scott Meyers and Steven P. Reiss. *CCEL: A Metalanguage for C++*. Proceedings of the 1992 USENIX C++ Conference, 99-115, August 1992.

[Eisenecker97] Ulrich W. Eisenecker. *Generative Programming (GP) with C++.* Proceedings of the Joint Modular Language Conference '97, LNCS 1204, 351-365, Springer, 1997.

[Ellis90] Margaret A. Ellis and Bjarne Stroustrup. *The Annotated C++ Reference Manual*. Addison-Wesley, 1990.

[Florijn97] Gert Florijn, Marco Meijers and Pieter van Winsen. *Tool Support for Object-Oriented Patterns.* Proceedings of the 11th European Conference on Object-Oriented Programming(ECOOP'97), 472-495, LNCS 1241, Springer, June 1997. http://www.serl.nl/people/florijn/papers/pattern-tool-overview.ps.gz (preliminary version)

[Forman94] Ira R. Forman, Scott Danforth and Hari Madduri. *Composition of Before/After Metaclasses in SOM.* Proceedings of the 1994 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications, OOPSLA'94, ACM SIGPLAN Notices, vol. 29, no. 10, 427-439, October 1994.

[Fowler97] Martin Fowler. *Analysis Patterns: Reusable Object Models*. Addison-Wesley, 1997.

[Fradet99] Pascal Fradet and Mario Südholt. *An Aspect Language for Robust Programming.* Position paper for *Aspect-Oriented Programming Workshop*

at the European Conference on Object-Oriented Programming (ECOOP'99), June 1999.
http://wwwtrese.cs.utwente.nl/aop-ecoop99/papers/fradet.pdf

[Fraser95] Christopher W. Fraser, David R. Hanson. *A retargetable C Compiler: Design and Implementation*. Benjamin/Cummings 1995.
Compiler is ftp://ftp.cs.princeton.edu/pub/packages/lcc/4.1.tar.gz.

[Gamma95] Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides. *Design patterns, Elements of reusable object-oriented software*. Addison-Wesley, 1995.

[Gil97] Joseph Gil and David H. Lorenz. *Design patterns vs. language design.* In *Proceedings of the Language Support for Design Patterns and Frameworks Workshop at ECOOP'97*. In *Workshop Reader of the European Conference on Object-Oriented Programming (ECOOP'97),* LNCS 1357, Springer-Verlag, June 1997.
http://bilbo.ide.hk-r.se:8080/~bosch/lsdforg/lorenz.ps

[Glandrup95] M. Glandrup. *Extending C++ Using the Concepts of Composition Filters.* MSc Thesis, University of Twente, November 1995.
http://wwwtrese.cs.utwente.nl/Docs/Tresepapers/AspectLanguages/C++/glandrup.thesis.pdf

[Gosling97] J. Gosling, B. Joy and G. Steele. *The Java language specification*. Addison-Wesley, 1997.

[Gourhant90] Yvon Gourhant and Marc Shapiro. *FOG/C++: A Fragmented-Object Generator*. Proceedings of the 1990 USENIX C++ Conference, April 1990.
ftp://ftp.inria.fr/INRIA/Projects/SOR/papers/1990/FOG_UsenixC++90.ps.gz

[Gowing96] Brendan Gowing and Vinny Cahill. *Meta-Object Protocols for C++: The Iguana Approach.* Proceedings of Reflection'96, April 1996.
ftp:://ftp.dsg.cs.tcd.ie/pub/doc/dsg-97.ps.gz

[Grand98] Mark Grand. *Patterns in Java, Volume 1*. Wiley, 1998.

[Graube89] Nicolas Graube. *Metaclass Compatibility.* Proceedings of the 1989 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications, OOPSLA'89, ACM SIGPLAN Notices, vol. 24, no. 10, 305-315, October 1989.

[Harrison93] William Harrison and Harold Ossher. *Subject-Oriented Programming (a critique of pure objects).* Proceedings of the 1993 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications, OOPSLA'93, ACM SIGPLAN Notices, vol. 28, no. 10, 411-428, October 1993.

[Harrison99] Neil Harrison, Brian Foote and Hans Rohnert. *Pattern Languages of Program Design 4*. Addison-Wesley, 1999 (planned).

[Hedin97a] Görel Hedin. *Language support for design patterns using attribute extension.* In *Proceedings of the Language Support for Design Patterns and Frameworks Workshop at ECOOP'97*. In *Workshop Reader of the European Conference on Object-Oriented Programming (ECOOP'97),* LNCS 1357, Springer-Verlag, June 1997.
http://bilbo.ide.hk-r.se:8080/~bosch/lsdforg/hedin.ps

[Hedin97b] Görel Hedin. *Attribute Extension - A Technique for Enforcing Programming Conventions.* Nordic Journal of Computing, vol. 4, no 1, 93-122, 1997.

[Hieb92] Robert Hieb, R.Kent Dybvig and Carl Bruggeman. *Syntactic abstraction in Scheme*. University of Indiana Computer Science Technical Report 355, June 1992 (revised July 1992). Journal of Lisp and Symbolic Computation, vol. 5, no. 4, 295-326, 1993.
ftp://ftp.cs.indiana.edu/pub/scheme-repository/doc/pubs/iucstr355.ps.gz

[Hoare74] C.A.R. Hoare. *Monitors: An Operating System Structuring Concept.* Communications of the ACM, vol. 17, no 10, 549-557, October 1974.

[Ishikawa96a] Yutaka Ishikawa, Atsushi Hori, Mitsuhisa Sato, Motohiko Matsuda, Jörg Nolte, Hiroshi Tezuka, Hiroki Konaka, Munenori Maeda and Kazuto Kubota. *Design and Implementation of Metalevel Architecture in C++ - MPC++ approach.* Proceedings of Reflection'96, April 1996.
http://pdswww.rwcp.or.jp/lab/pdslab/papers/reflection96.ps.gz

[Ishikawa96b] Yutaka Ishikawa, Atsushi Hori, Hiroshi Tezuka, Motohiko Matsuda, Hiroki Konaka, Munenori Maeda, Takashi Tomokiyo, Jörg Nolte and Mitsuhisa Sato. *MPC++.* In [Wilson96].

[Jacobson99] Ivar Jacobson, Grady Booch and James Rumbaugh. *The Unified Software Development Process*. Addison Wesley, 1999.

[Johnson75] S.C. Johnson. *Yacc - Yet Another Compiler Compiler*. Computing Science Technical Report 32, AT&T Bell Laboratories, Murray Hill, New Jersey, 1975.
http://www.cs.uaf.edu/~cs631/yacc-docs.txt

[Kasbekar98] Mangesh Kasbekar, Chandramouli Narayanan and Chita R. Das. *Using Reflection for Checkpointing Concurrent Object Oriented Programs.* Proceedings of the Workshop on Reflective Programming in C++ and Java at the 1998 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications, OOPSLA'98. UTCCP Report 98-4, Center for Computational Physics, University of Tsukuba Japan, ISSN 1344-3135, October 1998.
http://www.softlab.is.tsukuba.ac.jp/~chiba/oopsla98/proc/kasbekar.pdf

[Kernighan78] Brian W. Kernighan, and Dennis M. Ritchie. *The C Programming Language*. Prentice Hall, 1978.

[Kiczales91] G. Kiczales, J. des Rivières and D.G. Bobrow. *The Art of the Metaobject Protocol*. MIT Press, 1991.

[Kiczales97] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Videira Lopes, Jean-Marc Loingtier and John Irwin. *Aspect-Oriented Programming.* Proceedings of the 11th European Conference on Object-Oriented Programming(ECOOP'97), LNCS 1241, Springer-Verlag, June 1997.
http://www.parc.xerox.com/spl/groups/eca/pubs/papers/KIczales-ECOOP97/for-web.pdf

[Knudsen99] Jørgen Lindskov Knudsen. *Aspect-Oriented Programming in BETA using the Fragment System.* Position paper for *Aspect-Oriented Programming Workshop at the European Conference on Object-Oriented Programming (ECOOP'99),* June 1999.
http://wwwtrese.cs.utwente.nl/aop-ecoop99/papers/knudsen.pdf

[Larus96] James R. Larus, Brad Richards and Guhan Viswanathan. *C**.* In [Wilson96].

[Lauder98] Anthony Lauder and Stuart Kent. *Precise visual specification of mined pattern languages.* Proceedings of the 12th European Conference on Object-Oriented Programming, ECOOP'98, LNCS 1445, 114-134, Springer, July 1998.
http://www.cs.ukc.ac.uk/people/staff/sjhk/detail/publications/ECOOP98.html

[Leavenworth66] B.M. Leavenworth. *Syntax macros and extended translation*. Communications of the ACM, vol. 9, no. 11, 790-793, November 1966.

[Lee98] Arthur H. Lee and Ho-Yun Shin. *Building a Persistent Object Store using the Java Reflection API.* Proceedings of the Workshop on Reflective Programming in C++ and Java at the 1998 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications, OOPSLA'98. UTCCP Report 98-4, Center for Computational Physics, University of Tsukuba Japan, ISSN 1344-3135, October 1998.
http://www.softlab.is.tsukuba.ac.jp/~chiba/oopsla98/proc/lee.pdf

[Lesk75] M.E. Lesk. *Lex - a Lexical Analyser Generator*. Computing Science Technical Report 39, AT&T Bell Laboratories, Murray Hill, New Jersey, 1975.
http://www.cs.uaf.edu/~cs631/lex-docs.txt

[Levine90] John R. Levine, Tony Mason and Doug Brown. *Lex & Yacc*. Nutshell Handbook, O'Reilly & Associates, Inc., (first published) May 1990.

[Lilley97] John Lilley. *John Lilley's PCCTS-Based LL(1) C++ Parser, Version 1.5.* February 1997.
Docs\Theory_and_operation.doc in http://www.empathy.com/pccts/cppgrammar.zip

[Lin99] Chao-Hsin Lin. *A Three-level Reflective Architecture for Abstracting Intra-Object Control.* Proceedings of the Eastern European Conference on Technology of Object Oriented Languages and Systems, ISBN 954-90484-1-1, 64-77, Blagoevgrad, Bulgaria, June 1999.

[Lippman98] Stanley B. Lippman. *C++ Gems.* Cambridge University Press, 1998.

[Lopes98] Cristina Videira Lopes and Gregor Kiczales. *Recent developments in AspectJ.* In *Proceedings of the Aspect-Oriented Programming Workshop at ECOOP'98*. In *Workshop Reader of the European Conference on Object-Oriented Programming (ECOOP'98),* LNCS 1543, Springer-Verlag, June 1998.
http://wwwtrese.cs.utwente.nl/aop-ecoop98/papers/Lopes.pdf

[Lorenz98] David H. Lorenz. *Visitor Beans: An Aspect-Oriented Pattern.* In *Proceedings of the Aspect-Oriented Programming Workshop at ECOOP'98*. In *Workshop Reader of the European Conference on Object-Oriented Programming (ECOOP'98),* LNCS 1543, Springer-Verlag, June 1998.
http://wwwtrese.cs.utwente.nl/aop-ecoop98/papers/Lorenz.pdf

[Lunau98] Charlotte Pii Lunau. *Is Composition of Metaobjects = Aspect Oriented Programming.* In *Proceedings of the Aspect-Oriented Programming Workshop at ECOOP'98*. In *Workshop Reader of the European Conference on Object-Oriented Programming (ECOOP'98),* LNCS 1543, Springer-Verlag, June 1998.
http://wwwtrese.cs.utwente.nl/aop-ecoop98/papers/Lunau.pdf

[Maddox89] William Maddox. *Semantically-Sensitive Macroprocessing.* Report No. UCB/CSD 89/545, (Master's Thesis), University of California, Berkeley, 1989.
http://www.cs.berkeley.edu/~maddox/papers/msrept/xlrept.ps

[Madsen93] Ole Lehrmann Madsen, Birger Møller-Pedersen and Kristen Nygaard. *Object-Oriented Programming in the BETA Programming Language*. Addison-Wesley, 1993.

[Maes87] Pattie Maes. *Concepts and experiments in computational reflection.* Proceedings of the 1987 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications, OOPSLA'87, ACM SIGPLAN Notices, vol. 22, no. 12, 147-155, December 1987.

[Martin97] Robert C. Martin, Dirk Riehle and Frank Buschmann. *Pattern Languages of Program Design 3*. Addison-Wesley, 1997.

[Matthijs97] Frank Matthijs, Wouter Joosen, Bart Vanhaute, Bert Robben and Pierre Verbaeten. *Aspects should not die.* In Cristina Lopes, Kim Mens, Bedir Tekinerdogan and Gregor Kiczales. *Proceedings of the Aspect-Oriented Programming Workshop at ECOOP'97*. In *Workshop Reader of the European Conference on Object-Oriented Programming (ECOOP'97),* LNCS 1357, Springer-Verlag, June 1997.
http://www.parc.xerox.com/spl/projects/aop/ecoop97/aop-ecoop97-proceedings.pdf

[Matsuoka93] Satoshi Matsuoka and Akinori Yonezawa. *Analysis of Inheritance Anomaly in Object-Oriented Concurrent Programming Languages.* In *Research*

Directions in Concurrent Object-Oriented Programming, 107-150, MIT Press, 1993.
ftp://ftp.yl.is.s.u-tokyo.ac.jp/pub/papers/book-inheritance-anomaly-a4.ps.Z

[McAffer95] Jeff McAffer. *Meta-level Programming with CodA.* Proceedings of the 9th European Conference on Object-Oriented Programming(ECOOP'95), LNCS 952, 190-214, Springer-Verlag, June 1995.
ffp://ftp.yl.is.s.u-tokyo.ac.jp/pub/memebers/jeff/docs/ecoop95.a4.ps.gz

[Meijers96] Marco Meijers and Gert Florijn. *Tool Support for Object-Oriented Design Patterns.* Master's Thesis, CS Dept., Utrecht University, INF-SCR-96-43, November 1996.
http://www.serl.nl/people/florijn/papers/pattern-tool.ps.gz (working draft)

[Mens97] Kim Mens, Cristina Lopes, Bedir Tekinerdogan and Gregor Kiczales. *Aspect-Oriented Programming workshop report.* In Cristina Lopes, Kim Mens, Bedir Tekinerdogan and Gregor Kiczales. *Proceedings of the Aspect-Oriented Programming Workshop at ECOOP'97*. In *Workshop Reader of the European Conference on Object-Oriented Programming (ECOOP'97),* LNCS 1357, Springer-Verlag, June 1997.
http://www.parc.xerox.com/spl/projects/aop/ecoop97/aop-ecoop97-proceedings.pdf

[Meyer92] Bertrand Meyer. *Eiffel: The Language*. Prentice Hall, 1992.

[Meyers92] Scott Meyers. *Effective C++: 50 Specific Ways to Improve Your Programs and Designs*. Addison-Wesley, 1992.

[Microsoft97] *C++ Language Reference*. In Microsoft Visual C++ 5.0 Programmer's Reference Set, Volume 4. Of Microsoft Visual C++ Language Reference, Microsoft Press, 1997.

[Muchnick93] V.B. Muchnick, A.V. Shafarenko and C.D. Sutton. *F-code and its implementation: a portable software platform for data parallelism*. The Computer Journal, vol. 36, no. 8, 712-722, 1993.

[Mulet95] Philippe Mulet, Jacques Malenfant and Pierre Cointe. *Towards a methodology for explicit composition of metaobjects.* Proceedings of the 1995 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications, OOPSLA'95, ACM SIGPLAN Notices, vol. 30, no. 10, 316-330, October 1995.

[Myers95] Nathan C. Myers. *Traits: A New and Useful Template Technique*. C++ Report, vol. 7, no. 5, June 1995. Reprinted in [Lippman98].
http://www.cantrip.org/traits.html

[Oliva98] Alexandre Oliva and Luiz Eduardo Buzato. *Composition of Meta-Objects in Guaraná.* Proceedings of the Workshop on Reflective Programming in C++ and Java at the 1998 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications, OOPSLA'98. UTCCP Report 98-4, Center for Computational Physics, University of Tsukuba Japan, ISSN 1344-3135, October 1998.
http://www.softlab.is.tsukuba.ac.jp/~chiba/oopsla98/proc/oliva.pdf

[Ossher95] Harold Ossher, Matthew Kaplan, William Harrison, Alexander Katz and Vincent Kruskal. *Subject-Oriented composition rules.* Proceedings of the 1995 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications, OOPSLA'95, ACM SIGPLAN Notices, vol. 30, no. 10, 235-250, October 1995.

[Ossher98] Harold Ossher and Peri Tarr. *Operation-Level Composition: A Case in (Join) Point.* In Cristina Lopes, Gail Murphy and Gregor Kiczales. *Proceedings of the Aspect-Oriented Programming Workshop at ICSE'98. 98. April 1998*. And in *Proceedings of the Aspect-Oriented Programming Workshop at ECOOP'98*. In *Workshop Reader of the European Conference*

on Object-Oriented Programming (ECOOP'98), LNCS 1543, Springer-Verlag, June 1998.
http://www.parc.xerox.com/spl/projects/aop/icse98/aop-icse98-proceedings.pdf
http://wwwtrese.cs.utwente.nl/aop-ecoop98/papers/Ossher.pdf

[Park96] Chong-Mok Park, Kyu-Young Whang, Il-Yeol Song and Shamkant Navathe. *Forced inheritance: A new approach for providing orthogonal persistence to C++*. Journal of Object Oriented Programming, 65-71, March/April 1996.

[Porat95] Sara Porat and Paul Fertig. *Class assertions in C++*. Journal of Object Oriented Programming, 30-37, May 1995.

[Pree94] Wolfgang Pree. *Design Patterns for Object-Oriented Software Development*. Addison-Wesley, 1994.

[Proebsting95] Todd A. Proebsting. *BURS automata generation*. ACM Transactions on Programming Languages and Systems, vol. 17, no. 3, 461-486, May 1995.

[Pryor99] Jane Pryor and Natalio Bastán. *A Reflective Architecture for the Support of Aspect Oriented Programming in Smalltalk.* Position paper for *Aspect-Oriented Programming Workshop at the European Conference on Object-Oriented Programming (ECOOP'99),* June 1999.
http://wwwtrese.cs.utwente.nl/aop-ecoop99/papers/pryor.pdf

[Reiss95] Steven P. Reiss and Tony Davis. *Experiences Writing Object-Oriented Compiler Front Ends.* January 1995.
oofront.ps in ftp://ftp.cs.brown.edu/pub/cppp.tar.Z

[Riehle97] Dirk Riehle. *Composite Design Patterns.* Proceedings of the 1997 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications, OOPSLA'97, ACM SIGPLAN Notices, vol. 32, no. 10, 218-228, October 1997.

[Riehle98] Dirk Riehle and Thomas Gross. *Role model based framework design and integration.* Proceedings of the 1998 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications, OOPSLA'98, ACM SIGPLAN Notices, vol. 33, no. 10, 117-133, October 1998.

[Roskind91] Jim Roskind. *A YACC-able C++ 2.1 Grammar, and the Resulting Ambiguities*. Release 2.0. Updated 11 July 1991.
grammar5.txt in http://www.empathy.com/pccts/roskind.zip

[Rumbaugh99] James Rumbaugh, Ivar Jacobson and Grady Booch. *The Unified Modelling Language Reference Manual*. Addison Wesley, 1999.

[Schmidt95] Douglas C. Schmidt. *Reactor: An Object Behavioural Pattern for Concurrent Event Demultiplexing and Event Handler Dispatching*. In [Coplien95b].

[Seinturier99] Lionel Seinturier. *JST: An Object Synchronisation Aspect for Java.* Position paper for *Aspect-Oriented Programming Workshop at the European Conference on Object-Oriented Programming (ECOOP'99),* June 1999.
http://wwwtrese.cs.utwente.nl/aop-ecoop99/papers/seinturier.pdf

[Shilling89] John J. Shilling and Peter F. Sweeney. *Three Steps to Views: Extending the Object-Oriented Paradigm.* Proceedings of the 1989 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications, OOPSLA'89, ACM SIGPLAN Notices, vol. 24, no. 10, 353-361, October 1989.

[Singhal96] Vivek P. Singhal. *A Programming Language for Writing Domain-Specific Software System Generators.* PhD Dissertation, University of Texas at Austin, August 1996.
ftp://ftp.cs.utexas.edu/pub/predator/vivek-thesis.ps.Z

[Smaragdakis98] Yannis Smaragdakis and Don Batory. *Implementing Layered Designs with Mixin Layers.* Proceedings of the 12th European Conference on Object-

Oriented Programming, ECOOP’98, LNCS 1445, 550-570, Springer-Verlag, July 1998.
ftp://ftp.cs.utexas.edu/pub/predator/templates.ps

[Smith84] Brian Cantwell Smith. *Reflection and semantics in Lisp*. Proceedings of the 11th Annual Symposium on Principles of Programming Languages, 23-35, January 1984.

[Sobel96] Jonathan M. Sobel and Daniel P. Friedman. *An Introduction to Reflection-Oriented Programming*. Proceedings of Reflection’96, April 1996.
http://www.cs.indiana.edu/hyplan/jsobel/rop.ps.gz

[Solntseff74] N. Solntseff and A. Yezerski. *A survey of extensible programming languages*. In *Annual Review in Automatic Programming.* vol. 7, part 5, 267-307, Pergamon Press, 1974.

[Soukup94] Jiri Soukup. *Taming C++: Pattern Classes and Persistence for Large Projects*. Addison-Wesley, 1994.

[Soukup95] Jiri Soukup. *Implementing Patterns*. In [Coplien95b].

[Stallman98] Richard M. Stallman. *Using and porting GNU C*. Free Software Foundation, Inc., Cambridge MA, January 1998. Available as part of the gcc-2.8.0 distribution.
ftp://prep.ai.mit.edu/pub/gnu/gcc/gcc-2.8.0.tar.gz

[Stroud94] Robert J. Stroud and Zhixue Wu. *Using Meta-Objects to Adapt a Persistent Object System to Meet Application Needs.* 6th ACM SIGOPS European Workshop on Matching Operating Systems to Application Needs, 35-38, September 1994. University of Newcastle upon Tyne, Computing Science, Technical Report 513, April 1995.

[Stroud95] Robert J. Stroud and Zhixue Wu. *Using Meta-Object Protocols to Implement Atomic Data Types.* Proceedings of the 9th European Conference on Object-Oriented Programming(ECOOP’95), LNCS 952, 168-189, Springer-Verlag, June 1995. University of Newcastle upon Tyne, Computing Science, Technical Report 512, March 1995.

[Stroustrup94] Bjarne Stroustrup. *The Design and Evolution of C++*. Addison-Wesley, 1994.

[Stroustrup97] Bjarne Stroustrup. *The C++ programming language. Third edition*. Addison-Wesley, 1997.

[Tatsubori98] Michiaki Tatsubori and Shigeru Chiba. *Programming Support of Design Patterns with Compile-time Reflection.* Proceedings of the Workshop on Reflective Programming in C++ and Java at the 1998 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications, OOPSLA’98. UTCCP Report 98-4, Center for Computational Physics, University of Tsukuba Japan, ISSN 1344-3135, October 1998.
http://www.softlab.is.tsukuba.ac.jp/~chiba/oopsla98/proc/mich.pdf

[Veldhuizen94] Todd Veldhuizen. *Expression Templates*. C++ World, 1994. Also in: C++ Report, vol. 7, no. 5, 26-31, June 1995. Reprinted in [Lippman98].
http://extreme.indiana.edu/~tveldhui/papers/cppworld.ps

[Veldhuizen95] Todd Veldhuizen. *Using C++ Template Metaprograms*. C++ Report, vol. 7, no. 4, 36-43, May 1995. Reprinted in [Lippman98].
http://extreme.indiana.edu/~tveldhui/papers/meta-art.ps

[Vemulapati95] Murali Vemulapati, Ram D. Sriram and Amar Gupta. *Incremental loading in the persistent C++ language E*. Journal of Object Oriented Programming, 34-35,38-42, July/August 1995.

[Vidart74] J. Vidart. *Extensions syntactiques dans une contexte LL(1)*. Thése pour obtenir le grade de Docteur de troisiéme cycle,1974.

[Vlissides96] John M. Vlissides, James O. Coplien and Norman L. Kerth. *Pattern Languages of Program Design 2*. Addison-Wesley, 1996.

[Vlissides98] John Vlissides. *Pattern hatching: Design patterns applied*. Addison-Wesley, 1998.

[Wand88] Mitchell Wand and Daniel P. Friedman. *The mystery of the tower revealed: A nonreflective description of the reflective tower.* LISP and Symbolic Computation, vol. 1, no. 1, 11-38, June 1988.

[Weise93] Daniel Weise and Roger Crew. *Programmable syntax macros*. Proceedings of the ACM SIGPLAN '93 Conference on Programming Language Design and Implementation, ACM SIGPLAN Notices, vol. 28, no. 6, 156-165, June 1993.

[Welch98] Ian Welch and Robert Stroud. *Dalang - A Reflective Java Extension.* Proceedings of the Workshop on Reflective Programming in C++ and Java at the 1998 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications, OOPSLA'98. UTCCP Report 98-4, Center for Computational Physics, University of Tsukuba Japan, ISSN 1344-3135, October 1998.
http://www.softlab.is.tsukuba.ac.jp/~chiba/oopsla98/proc/welch.pdf

[Welch99] Ian Welch and Robert Stroud. *Load-time Application of Aspects to Java COTS Software.* Position paper for *Aspect-Oriented Programming Workshop at the European Conference on Object-Oriented Programming (ECOOP'99),* June 1999.
http://wwwtrese.cs.utwente.nl/aop-ecoop99/papers/welch.pdf

[Werther96] Ben Werther and Damian Conway. *A Modest Proposal: C++ Resyntaxed*. ACM SIGPLAN Notices, vol. 31, no. 11, 74-82, November 1996.

[Wild96] Fred Wild. *Instantiating code patterns.* Dr. Dobb's Journal, 72,74-76,88-91, June 1996.

[Willink97a] Edward D. Willink, Alexander V. Shafarenko and Vyacheslav B. Muchnick. *Use of F-code as a Very High Level Intermediate Language for DSP.* Proceedings of the Third International Euro-Par Conference, Euro-Par'97, 820-823, August 1997. In Euro-Par'97, Parallel Processing, Christian Lengauer, Martin Griebl and Sergei Gorlatch (Eds.), LNCS 1300, Springer-Verlag, August 1997.
http://www.ee.surrey.ac.uk/Personal/E.Willink/europar97.{gz,pdf,ps,zip}

[Willink97b] Edward D. Willink. *Low level Object Oriented DSP Development in C++*. Proceedings of the 8th International Conference on Signal Processing Applications and Technology, ICSPAT'97, 894-898, San Diego, California, September 1997.
http://www.ee.surrey.ac.uk/Personal/E.Willink/icspat97.{gz,pdf,ps,zip}

[Willink99a] Edward D. Willink and Vyacheslav B. Muchnick. *Preprocessing C++: Substitution and Composition.* Proceedings of the Eastern European Conference on Technology of Object Oriented Languages and Systems, ISBN 954-90484-1-1, 122-135, Blagoevgrad, Bulgaria, June 1999.
http://www.computing.surrey.ac.uk/research/dsrg/fog/FogToolsEE1.{gz,pdf,ps,zip}

[Willink99b] Edward D. Willink and Vyacheslav B. Muchnick. *Preprocessing C++: Meta-Class Aspects.* Proceedings of the Eastern European Conference on Technology of Object Oriented Languages and Systems, ISBN 954-90484-1-1, 136-149, Blagoevgrad, Bulgaria, June 1999.
http://www.computing.surrey.ac.uk/research/dsrg/fog/FogToolsEE2.{gz,pdf,ps,zip}

[Willink99c] Edward D. Willink and Vyacheslav B. Muchnick. *Weaving a Way Past the C++ One Definition Rule.* Position paper for the *Aspect-Oriented Programming Workshop at ECOOP'99*, June 1999.
http://www.computing.surrey.ac.uk/research/dsrg/fog/AopEcoop99.{gz,pdf,ps,zip}

[Wilson96] Gregory V. Wilson and Paul Lu. *Parallel Programming using C++*. MIT Press, 1996.

[Wu98] Zhixue Wu. *Reflective Java and A Reflective Component-Based Transaction Architecture.* Proceedings of the Workshop on Reflective

Programming in C++ and Java at the 1998 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications, OOPSLA'98. UTCCP Report 98-4, Center for Computational Physics, University of Tsukuba Japan, ISSN 1344-3135, October 1998.
http://www.softlab.is.tsukuba.ac.jp/~chiba/oopsla98/proc/wu.pdf

[Yokote92] Yasuhiko Yokote. *The Apertos Reflective Operating System: The Concept and Its Implementation.* Proceedings of the 1992 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications, OOPSLA'92, ACM SIGPLAN Notices, vol. 27, no. 10, 414-434, October 1992.
ftp://ftp.csl.sony.co.jp/CSL/CSL_Papers/SCSL-TR-92-014.ps.Z

A FOG Grammar changes

This summary of the FOG syntax follows the example of Annex A of the [C++98] standard. It is intended to be an aid to comprehension, identifying all changes in a way that is easily compared to C++. Most completely unchanged grammar productions are omitted. Changed rules and terms are indicated by a ~~strike-through~~ for removal and underline for addition.

A.1 Keywords

~~typedef-name~~:
    identifier

~~namespace-name~~:
    original-namespace-name
    namespace-alias

~~original-namespace-name~~:
    identifier

~~namespace-alias~~:
    identifier

~~class-name~~:
    identifier
    template-id

~~enum-name~~:
    identifier

~~template-name~~:
    identifier

punctuation: one of

`{`	`}`	`[`	`]`	`(`	`)`	`;`	`:`	`?`	`::`	`.`	`.*`	
`+`	`-`	`*`	`/`	`%`	`^`	`&`	`|`	`~`	`!`	`=`	`<`	`>`
`+=`	`-=`	`*=`	`/=`	`%=`	`^=`	`&=`	`|=`	`<<=`	`>>=`			
`<<`	`>>`	`==`	`!=`	`<=`	`>=`	`&&`	`||`	`++`	`--`	`,`	`->`	`->*` `...`

*reserved-word*s are not *identifier*s.

reserved-word: one of

`and`	`and_eq`	`asm`	`auto`
`bitand`	`bitor`	`bool`	`break`
`case`	`catch`	`char`	`class`
`compl`	`const`	`const_cast`	`continue`
`default`	`delete`	`do`	`double`
`dynamic_cast`	`else`	`enum`	`explicit`
`export`	`extern`	`false`	`float`
`for`	`friend`	`goto`	`if`
`inline`	`int`	`long`	`mutable`
`namespace`	`new`	`not`	`not_eq`
`operator`	`or`	`or_eq`	`private`
`protected`	`public`	`register`	`reinterpret_cast`
`return`	`short`	`signed`	`sizeof`
`static`	`static_cast`	`struct`	`switch`
`template`	`this`	`throw`	`true`
`try`	`typedef`	`typeid`	`typename`
`unsigned`	`using`	`virtual`	`void`
`volatile`	`wchar_t`	`while`	`xor`
`xor_eq`			

*non-reserved-word*s are *identifier*s.

non-reserved-word: one of

`derived`	`emit`	`entry`	`exit`
`file`	`frozen`	`guard`	`implementation`
`include`	`interface`	`noguard`	`noimplementation`
`path`	`pool`	`post`	`pre`
`prefix`	`pure`	`suffix`	`utility`

*meta-type-name*s are *reserved-word*s or *identifier*s.

meta-type-name:
  intrinsic-meta-type-name
  actual-meta-type-name
  potential-meta-type-name

intrinsic-meta-type-name: one of

```
array_modifier                  assignment_expression
character                       class_key
constant_expression             cv_qualifier
decl_specifier                  expression
function_modifier               handler
identifier                      iterator
keyword                         meta_type
modifier                        name
nil                             number
pointer_modifier                punctuation
reference_modifier              reserved
string                          token
tree_literal                    using_directive
```

potential-meta-type-name: one of

```
base_specifier                  built_in_type_specifier
class_specifier                 declaration
elaborated_type_specifier       enum_specifier
enumerator_definition           exception_specification
filespace_specifier             function_specifier
linkage_specification           meta_class_specifier
meta_function_specifier         meta_parameter_specifier
meta_variable_specifier         namespace_definition
namespace_alias_definition      object_specifier
parameter_specifier             scope_specifier
statement                       template_parameter_specifier
type_id                         type_specifier
typedef_specifier               using_declaration
variable_specifier
```

actual-meta-type-name: one of *//* [1]

```
base                            built_in
class                           entity
enum                            enumerator
exception                       filespace
function                        linkage
meta_class                      meta_function
meta_parameter                  meta_variable
namespace                       namespace_alias
object                          parameter
scope                           template_parameter
type                            typedef
typename                        using
variable
```

A.2 Lexical conventions

A.2.1 Phase 6 Concatenation Grammar

text-literal$_{pp}$:
  character-literal$_{pp}$
  identifier$_{pp}$ *// Including all reserved words*
  number-literal$_{pp}$
  string-literal$_{pp}$
  tree-literal$_{pp}$ *// a* $ *or* @ *expression*

character-literal$_{cat}$:
  character-literal$_{pp}$
  character-literal$_{cat}$ *text-literal*$_{pp}$

identifier$_{cat}$:
  identifier$_{pp}$
  identifier$_{cat}$ *text-literal*$_{pp}$

1. Some meta-type names are also reserved words. The usage as a meta-type name augments usage as a reserved word.

number-literal$_{cat}$:
number-literal$_{pp}$
number-literal$_{cat}$ *text-literal*$_{pp}$

string-literal$_{cat}$:
string-literal$_{pp}$
string-literal$_{cat}$ *text-literal*$_{pp}$
string-literal$_{cat}$ *whitespace*$_{opt}$ *string-literal*$_{pp}$

tree-literal$_{cat}$:
tree-literal$_{pp}$
tree-literal$_{cat}$ *text-literal*$_{pp}$

‘anything-else’$_{cat}$:
‘anything-else’$_{pp}$

tree-literal$_{pp}$:
at-literal
dollar-literal
syntax-macro-literal — *// See Section 3.7*

at-literal:
`@` *tree-expression*
`@ {` *tree-expression* `}`

dollar-literal:
`$` *tree-expression*
`$ {` *tree-expression* `}`
`$` *dollar-literal*

A.2.2 Phase 7 Tokenization Grammar

‘discard’: — *// Token is discarded*
whitespace

‘reserved-word’:
identifier$_{cat}$ — *// If identifier*$_{cat}$ *is a reserved word*

‘punctuation’:
‘punctuation’$_{pp}$

character-literal:
character-literal$_{cat}$

floating-literal:
number-literal$_{cat}$ — *// If number-literal*$_{cat}$ *is floating point*

integer-literal:
number-literal$_{cat}$ — *// If number-literal*$_{cat}$ *is fixed point*

string-literal:
string-literal$_{cat}$

meta-type-name: — *// If identifier*$_{cat}$ *is a meta-type name*
identifier$_{cat}$ — *// (and not a reserved word)*

‘non-reserved-word’:
identifier$_{cat}$ — *// If identifier*$_{cat}$ *is a non-reserved word*

~~*identifier*~~:
other-identifier:
identifier$_{cat}$ — *// If identifier*$_{cat}$ *is anything else*

tree-literal:
tree-literal$_{cat}$

A.3 Basic concepts

A.4 Expressions

identifier:
other-identifier
meta-type-name
non-reserved-word
tree-literal

id:
identifier
identifier < *template-argument-list* > //[2]
`template` *identifier* < *template-argument-list* >

nested-id:
id
id `::` *nested-id*

scoped-id:
`::`$_{opt}$ *nested-id*

special-function-id:
~ *id*
conversion-function-id
operator-function-id

nested-special-function-id:
special-function-id
id `::` *nested-function-special-id*

scoped-special-function-id:
`::`$_{opt}$ *nested-special-function-id*

primary-expression:
literal
`this`
~~`::` *identifier*~~
~~`::` *operator-function-id*~~
~~`::` *qualified-id*~~
(*expression*)
~~*id-expression*~~
declarator-id

A.5 Statements

statement:
control-statement
expression-statement
compound-statement
declaration-statement
try-block
`auto` *control-statement*
`auto` *meta-expression-statement*

control-statement: //[3]
labeled-statement
selection-statement
iteration-statement
jump-statement

2. Resolution of the *identifier* < context-dependency is discussed in Section 4.8.2.

3. New non-terminal but no changed functionality

A.6 Declarations

declaration:
- *block-declaration*
- *function-definition*
- *template-declaration*
- *explicit-instantiation*
- *explicit-specialization*
- *linkage-specification*
- *namespace-definition*
- *namespace-declaration*
- *accessibility-specifier*
- *compound-declaration*
- *meta-control-declaration*
- `auto` *meta-control-declaration*
- *meta-expression-statement*
- `auto` *meta-expression-statement*
- `auto` *meta-class-declaration*
- `auto` *meta-function-definition*
- `auto` *meta-variable-declaration*
- *syntax-macro-definition*
- *include-declaration*
- *file-dependency-declaration*
- *file-placement-declaration*
- *filespace-declaration*

block-declaration:
- *simple-declaration*
- *asm-definition*
- *namespace-alias-definition*
- ~~*using-declaration*~~ // [4]
- *using-directive*

compound-declaration:
- `{` *declaration-seq*$_{opt}$ `}`

simple-declaration:
- *decl-specifier-seq*$_{opt}$ *init-declarator-list*$_{opt}$ `;`

decl-specifier:
- *storage-class-specifier*
- *type-specifier*
- *function-specifier*
- `friend`
- `typedef`
- *access-specifier*
- `using`

storage-class-specifier:
- ~~`auto`~~ // [5]
- `register`
- *static*
- `extern`
- `mutable`

static: // [3]
- `static`
- `! static`

function-specifier:
- `inline`
- `! inline`
- `inline / implementation`
- `inline / interface`
- `virtual`
- `! virtual`
- `virtual / pure`
- `explicit`

4. *using-declaration* is generalised and covered by *simple-declaration*

5. Compatibility can be retained by allowing `auto` within a *function-body*.

type-specifier:
simple-type-specifier
class-specifier
enum-specifier
elaborated-type-specifier
cv-qualifier

simple-type-specifier:
~~`::`$_{opt}$ *nested-name-specifier*$_{opt}$ *type-name*~~ *scoped-id*
built-in-type-specifier // [6]

built-in-type-specifier:
`char`
`wchar_t`
`bool`
`short`
`int`
`long`
`signed`
`unsigned`
`float`
`double`
`void`

elaborated-type-specifier:
class-key ~~`::`$_{opt}$ *nested-name-specifier*$_{opt}$ *identifier*~~ *scoped-id*
`enum` ~~`::`$_{opt}$ *nested-name-specifier*$_{opt}$ *identifier*~~ *scoped-id*
`typename` ~~`::`$_{opt}$ *nested-name-specifier*$_{opt}$ *identifier*~~ *scoped-id*
~~`typename ::`$_{opt}$ *nested-name-specifier*$_{opt}$ *identifier* < *template-argument-list* >~~

enum-specifier:
`enum` ~~*identifier*$_{opt}$~~ *scoped-id*$_{opt}$ { *enumerator-list*$_{opt}$ }

namespace-declaration:
`namespace` *scoped-id* `;`

namespace-definition:
~~*named-namespace-definition*~~
~~*unnamed-namespace-definition*~~
`namespace` *scoped-id*$_{opt}$ { *namespace-body* }

~~*named-namespace-definition*~~*:*
original-namespace-definition
extension-namespace-definition

~~*original-namespace-definition*~~*:*
`namespace` *identifier* { *namespace-body* }

~~*extension-namespace-definition*~~*:*
`namespace` *original-namespace-name* { *namespace-body* }

~~*unnamed-namespace-definition*~~*:*
`namespace` { *namespace-body* }

~~*using-declaration*~~*:* // [4]
`using typename`$_{opt}$ `::`$_{opt}$ *nested-name-specifier unqualified-id* `;`
`using ::` *unqualified-id* `;`

linkage-specification:
`extern` *string-literal* { *declaration-seq*$_{opt}$ }
`extern` *string-literal declaration*

A.7 Declarators

init-declarator-list:
init-declarator
init-declarator-list `,` *init-declarator*

6. The built-in types are split off to a distinct non-terminal.

init-declarator: // [7]
    declarator pure-specifier$_{opt}$ *object-statements-clause*$_{opt}$
    declarator initializer$_{opt}$ *object-statements-clause*$_{opt}$
    identifier$_{opt}$ `:` *constant-expression object-statements-clause*$_{opt}$

cv-qualifier:
    `const`
    `volatile`

declarator-id:
    ~~`::`$_{opt}$ *id-expression*~~
    ~~`::`$_{opt}$ *nested-name-specifier*$_{opt}$ *type-name*~~
    scoped-id
    scoped-special-function-id

function-definition:
    decl-specifier-seq$_{opt}$ *declarator pure-specifier*$_{opt}$ *ctor-initializer*$_{opt}$ *function-body*
    decl-specifier-seq$_{opt}$ *declarator pure-specifier*$_{opt}$ *function-try-block*

initializer:
    `=` *initializer-clause*
    `(` *expression-list* `)`

initializer-clause:
    assignment-expression
    `[` *constant-expression* `]` *assignment-expression*
    `{` *initializer-list* `,`$_{opt}$ `}`
    `{ }`

A.8 Classes

class-head:
    class-key ~~*identifier*$_{opt}$~~ *scoped-id*$_{opt}$ *base-clause*$_{opt}$
    ~~*class-key nested-name-specifier identifier base-clause*$_{opt}$~~

class-specifier:
    class-head `{` ~~*member-specification*$_{opt}$~~ *declaration-seq*$_{opt}$ `}`

accessibility-specifier:
    access-specifier `:`

~~*member-specification*~~*:*
    member-declaration member-specification$_{opt}$
    access-specifier `:` *member-specification*$_{opt}$

~~*member-declaration*~~*:*
    decl-specifier-seq$_{opt}$ *member-declarator-list*$_{opt}$ `;`
    function-definition `;`$_{opt}$
    qualified-id `;`
    using-declaration
    template-declaration

~~*member-declarator-list*~~*:*
    member-declarator
    member-declarator-list `,` *member-declarator*

~~*member-declarator*~~*:* // [7]
    declarator pure-specifier$_{opt}$
    declarator constant-initializer$_{opt}$
    identifier$_{opt}$ `:` *constant-expression*

pure-specifier:
    `= 0`

constant-initializer:
    `=` *constant-expression*

7. The distinction between *init-declarator* and *member-declarator* is removed. The ambiguity of a bit-field with a *labeled-statement* is resolved to the label.

A.9 Derived Classes

base-specifier:
~~`::`$_{opt}$ *nested-name-specifier*$_{opt}$ *class-name*~~
~~`virtual` *access-specifier*$_{opt}$ `::`$_{opt}$ *nested-name-specifier*$_{opt}$ *class-name*~~
~~*access-specifier* `virtual`$_{opt}$ `::`$_{opt}$ *nested-name-specifier*$_{opt}$ *class-name*~~
scoped-id
built-in-type-id
`virtual` *base-specifier* // [8]
`! virtual` *base-specifier*
access-specifier base-specifier // [8]
`auto` *base-specifier*

built-in-type-id:
built-in-type-specifier
built-in-type-id built-in-type-specifier

A.10 Special member functions

mem-initializer-id:
~~`::`$_{opt}$ *nested-name-specifier*$_{opt}$ *class-name*~~
~~*identifier*~~
scoped-id

A.11 Overloading

A.12 Templates

template-declaration:
`export`$_{opt}$ `using`$_{opt}$ `template` < *template-parameter-list* > *declaration*

explicit-specialization:
`using`$_{opt}$ `template` < > *declaration*

A.13 Exception Handling

A.14 Tree Literals

primary-tree-expression:
meta-scoped-id
`(` *tree-expression* `)`

postfix-tree-expression:
primary-tree-expression
postfix-tree-expression `(` *tree-argument-list*$_{opt}$ `)`
postfix-tree-expression `[` *expression* `]`
postfix-tree-expression `.` *scoped-id*
postfix-tree-expression `->` *scoped-id*

tree-expression:
postfix-tree-expression
`*` *tree-expression*

tree-argument-list:
tree-argument // [9]
tree-argument-list `,` *tree-argument*

tree-argument:
tree-statement
unterminated-tree-argument

tree-statement:
terminated-tree-argument
unterminated-tree-argument `;`

compound-tree-statement: // [10]
`{` *tree-statement-seq*$_{opt}$ `}`

8. A more general rather than changed syntax.
9. Ambiguities are resolved semantically by left to right maximisation of the length of each *tree-argument* with respect to its required syntax.

tree-statement-seq:
  tree-statement
  tree-statement-seq tree-statement

terminated-tree-argument: // [11]
  asm-definition
  control-statement
  declaration-statement
  explicit-instantiation
  explicit-specialization
  expression-statement
  file-dependency-declaration
  file-placement-declaration
  filespace-declaration
  include-declaration
  linkage-specification
  namespace-alias-definition
  namespace-declaration
  template-declaration
  using-declaration
  using-directive
  `auto` *meta-class-declaration*
  `auto` *meta-control-declaration*
  `auto` *meta-expression-statement*
  `auto` *meta-variable-declaration*

unterminated-tree-argument: // [11]
  access-specifier
  accessibility-specifier
  base-specifier
  built-in-type-id
  class-specifier
  compound-tree-statement
  condition
  cv-qualifier
  decl-specifier
  enum-specifier
  enumerator-definition
  expression
  filespace-specifier
  function-definition
  function-try-block
  handler-seq
  initializer-clause
  mem-initializer
  namespace-definition
  parameter-declaration
  reserved-word // [12]
  simple-type-parameter
  storage-class-specifier
  template-argument
  template-parameter
  type-parameter
  `auto` *meta-class-specifier*
  `auto` *meta-function-definition*

A.15 Object Statements

object-statements-clause:
  `: {` *object-statement-seq*$_{opt}$ `}`

---

10. An ambiguity arising from a *try-block* at the end of a *statement* followed by a *handler-seq* is resolved by maximising the length of the left-hand (*try-block*) element.
11. The presentation of these productions has huge ambiguities. They demonstrate some of the variety of C++ grammar that can be parsed in a context-free fashion. Very little cannot be. See Appendix C for an actual implementation that avoids the ambiguities.
12. `do;` is presumed to start an *iteration-statement*. `operator,` is presumed to be a *declarator-id*.

object-statement-seq:
  object-statement
  object-statement-seq object-statement

object-statement:
  `;`
  initializer `;`
  function-used-block
  file-dependency-declaration
  file-placement-declaration
  filespace-declaration
  meta-control-declaration
  `auto` *meta-control-declaration*
  `auto` *meta-expression-statement*
  `auto` *meta-function-definition*
  `auto` *meta-variable-declaration*
  derived-clause object-statement
  derived-clause `:` `{` *object-statement-seq*$_{opt}$ `}`

function-used-block:
  ctor-initializer `;`
  ctor-initializer$_{opt}$ *function-body*
  function-try-block
  `using` *file-id-list function-used-block*
  segment function-used-block

segment:
  `entry`
  `pre`
  `body`
  `post`
  `exit`

derived-clause:
  `derived` `(` *meta-conditional-expression* `)`

A.16 Meta-programming

A.16.1 Meta-names

meta-id:
  id
  meta-type
  `auto`

meta-nested-id:
  meta-id
  ~ *meta-id*
  meta-id `::` *meta-nested-id*

meta-scoped-id:
  `::`$_{opt}$ *meta-nested-id*

A.16.2 Meta-classes

meta-class-id:
  meta-id
  meta-id `::` *meta-class-id*

meta-class-specifier:
  meta-class-key meta-class-id
  meta-class-key meta-class-id base-specifier-clause$_{opt}$ `{` *declaration-seq*$_{opt}$ `}`

meta-class-declaration:
  meta-class-specifier `;`

meta-class-key:
  class-key
  `namespace`

A.16.3 Meta-types

meta-type:
meta-type-name
built-in-type-id
meta-class-key
`enum`
`typedef`
`typename`
`using`

exposed-tree:
`[ ]`

A.16.4 Meta-variables

meta-variable-declaration:
static$_{opt}$ `const`$_{opt}$ *meta-type meta-scoped-id exposed-tree*$_{opt}$ = *tree-statement*
static$_{opt}$ `const`$_{opt}$ *meta-type meta-scoped-id exposed-tree*$_{opt}$ *object-statements-clause*
static$_{opt}$ `const`$_{opt}$ *meta-type* `(` *meta-scoped-id* `)` *exposed-tree*$_{opt}$ = *tree-statement*
static$_{opt}$ `const`$_{opt}$ *meta-type* `(` *meta-scoped-id* `)` *exposed-tree*$_{opt}$ *object-statements-clause*

A.16.5 Meta-functions, Meta-constructors and Meta-destructors

meta-nested-constructor-id:
meta-id
meta-id `::` *meta-nested-constructor-id*

meta-scoped-constructor-id:
`::`$_{opt}$ *meta-nested-constructor-id*

meta-nested-destructor-id:
~ *meta-id*
meta-id `::` *meta-nested-destructor-id*

meta-scoped-destructor-id:
`::`$_{opt}$ *meta-nested-destructor-id*

meta-function-definition:
static$_{opt}$ *meta-type meta-scoped-id* `(` *meta-parameter-list*$_{opt}$ `)` *exposed-tree*$_{opt}$
compound-tree-statement
static$_{opt}$ *meta-type meta-scoped-id* `(` *meta-parameter-list*$_{opt}$ `)` *exposed-tree*$_{opt}$
object-statements-clause
meta-scoped-constructor-id `( )` *compound-tree-statement*
meta-scoped-constructor-id `( )` *object-statements-clause*
meta-scoped-destructor-id `( )` *compound-tree-statement*
meta-scoped-destructor-id `( )` *object-statements-clause*

meta-parameter-list:
meta-parameter
meta-parameter-list `,` *meta-parameter*

meta-parameter:
meta-type identifier exposed-tree$_{opt}$
meta-type identifier exposed-tree$_{opt}$ = *tree-argument*

A.16.6 Meta-statements

meta-control-declaration:
`case` *constant-expression* `:` *declaration*
`default` `:` *declaration*
`do` *declaration* `while` `(` *expression* `)` `;`
`for` `(` *for-init-statement condition*$_{opt}$ `;` *expression*$_{opt}$ `)` *declaration*
`if` `(` *condition* `)` *declaration*
`if` `(` *condition* `)` *declaration* `else` *declaration*
`switch` `(` *expression* `)` *declaration*
`while` `(` *condition* `)` *declaration*
jump-statement

A.16.7 Meta-expressions

meta-primary-expression:
    literal
    `this`
    meta-scoped-id
    meta-type meta-nested-id
    `(` *tree-argument-list*$_{opt}$ `)`

meta-postfix-expression:
    meta-primary-expression
    meta-postfix-expression `(` *tree-argument-list*$_{opt}$ `)`
    meta-postfix-expression `[` *expression*$_{opt}$ `]`
    meta-postfix-expression `.` *declarator-id*
    meta-postfix-expression `->` *declarator-id*
    meta-postfix-expression `++`
    meta-postfix-expression `--`

meta-unary-expression:
    meta-postfix-expression
    `++` *meta-unary-expression*
    `--` *meta-unary-expression*
    `*` *meta-unary-expression*
    `+` *meta-unary-expression*
    `-` *meta-unary-expression*
    `!` *meta-unary-expression*
    `~` *meta-unary-expression*
    `sizeof` *unary-expression*

meta-multiplicative-expression:
    meta-unary-expression
    meta-multiplicative-expression `*` *meta-unary-expression*
    meta-multiplicative-expression `/` *meta-unary-expression*
    meta-multiplicative-expression `%` *meta-unary-expression*

meta-additive-expression:
    meta-multiplicative-expression
    meta-additive-expression `+` *meta-multiplicative-expression*
    meta-additive-expression `-` *meta-multiplicative-expression*

meta-shift-expression:
    meta-additive-expression
    meta-shift-expression `<<` *meta-additive-expression*
    meta-shift-expression `>>` *meta-additive-expression*

meta-relational-expression:
    meta-shift-expression
    meta-relational-expression `<` *meta-shift-expression*
    meta-relational-expression `>` *meta-shift-expression*
    meta-relational-expression `<=` *meta-shift-expression*
    meta-relational-expression `>=` *meta-shift-expression*

meta-equality-expression:
    meta-relational-expression
    meta-equality-expression `==` *meta-relational-expression*
    meta-equality-expression `!=` *meta-relational-expression*

meta-and-expression:
    meta-equality-expression
    meta-and-expression `&` *meta-equality-expression*

meta-exclusive-or-expression:
    meta-and-expression
    meta-exclusive-or-expression `^` *meta-and-expression*

meta-inclusive-or-expression:
    meta-exclusive-or-expression
    meta-inclusive-or-expression `|` *meta-exclusive-or-expression*

meta-logical-and-expression:
    meta-inclusive-or-expression
    meta-logical-and-expression `&&` *meta-inclusive-or-expression*

meta-logical-or-expression:
    meta-logical-and-expression
    meta-logical-or-expression `||` *meta-logical-and-expression*

meta-conditional-expression:
    meta-logical-or-expression
    meta-logical-or-expression `?` *meta-conditional-expression* `:` *meta-conditional-expression*

meta-expression-statement:
    meta-conditional-expression `;`
    meta-logical-or-expression assignment-operator tree-statement

A.17 Syntax Macros

syntax-macro-definition:
    `explicit` `auto` *meta-type identifier* `(` *syntax-macro-parameter-list*$_{opt}$ `)` *exposed-tree*$_{opt}$
        compound-tree-statement

syntax-macro-parameter-list:
    syntax-macro-parameter
    syntax-macro-parameter-list `,` *syntax-macro-parameter*

syntax-macro-parameter:
    meta-type identifier exposed-tree$_{opt}$
    identifier
    reserved-word
    punctuation

A.18 Files

include-declaration:
    `using` *slash-include*$_{opt}$ *slash-utility*$_{opt}$ *string-literal* `;`

slash-include:
    `/ include`

slash-utility:
    `/` *utility*

utility:
    `emit`
    `pool`
    `utility`
    `frozen`

file-dependency-declaration:
    `using` `/` `implementation` `=`$_{opt}$ *file-specifier* `;`
    `using` `/` `interface` `=`$_{opt}$ *file-specifier* `;`

file-placement-declaration:
    `export` `/` `implementation` `=`$_{opt}$ *file-specifier* `;`
    `export` `/` `interface` `=`$_{opt}$ *file-specifier* `;`
    `export` `/` `noimplementation` `;`

file-specifier:
    file-name
    file-entity
    file-entity `/` `implementation`
    file-entity `/` `interface`

file-name:
    string-literal
    file-name `/` `interface`
    file-name `/` `implementation`
    file-name `/` `template`
    file-name `/` *utility*
    file-name `/` `guard` `=` *string-literal*
    file-name `/` `noguard`
    file-name `/` `path` `=` *string-literal*
    file-name `/` `prefix` `=` *string-literal*
    file-name `/` `suffix` `=` *string-literal*

file-entity:
    declarator-id
    elaborated-type-specifier
    `namespace` *scoped-id*

filespace-specifier:
    `namespace` `/` `file` *file-name compound-declaration*

filespace-declaration:
    filespace-specifier `;`

B C++ Grammar

The complete C++ grammar implemented using the superset approach outlined for FOG is presented in this appendix.

The presented grammar has been derived automatically from FogParser.y by a sed script to

- remove FOG specific grammar
- remove irrelevant action rules
- simplify relevant action rules
- remove implementation specific $ clutter

The resulting text is acceptable to yacc and has 0 unresolved conflicts.

It is available from

http://www.computing.surrey.ac.uk/research/dsrg/fog/CxxGrammar.y

```
/* This is a yacc-able parser for the entire ANSI C++ grammar with no unresolved conflicts. */
/* The parse is SYNTACTICALLY consistent and requires no template or type name assistance.
 * The grammar in the C++ standard notes that its grammar is a superset of the true
 * grammar requiring semantic constraints to resolve ambiguities. This grammar is a really big
 * superset unifying expressions and declarations, eliminating the type/non-type distinction,
 * and iterating to find a consistent solution to the template/arith,metoic < ambiguity.
 * As a result the grammar is much simpler, but requires the missing semantic constraints to be
 * performed in a subsequent semantic pass, which is of course where they belong. This grammar will
 * support conversion of C++ tokens into an Abstract Syntax Tree. A lot of further work is required to
 * make that tree useful.
 *
 *  Author:         E.D.Willink             Ed.Willink@rrl.co.uk
 *  Date:           19-Nov-1999
 */
/*
 * The lexer (and/or a preprocessor) is expected to identify the following
 *
 *  Punctuation:
 */
%type <keyword> ‘+’ ‘-’ ‘*’ ‘/’ ‘%’ ‘^’ ‘&’ ‘|’ ‘~’ ‘!’ ‘<‘  ‘>’ ‘=’ ‘:’ ‘[‘ ‘]’ ‘{‘ ‘}’ ‘(‘ ‘)’
%type <keyword> ‘?’ ‘.’ ‘\’’ ‘\”’ ‘\\’ ‘@’ ‘$’ ‘;’ ‘,’
/*
 *  Punctuation sequences
 */
%term <keyword> ARROW ARROW_STAR DEC EQ GE INC LE LOG_AND LOG_OR NE SHL SHR
%term <keyword> ASS_ADD ASS_AND ASS_DIV ASS_MOD ASS_MUL ASS_OR ASS_SHL ASS_SHR ASS_SUB ASS_XOR
%term <keyword> DOT_STAR ELLIPSIS SCOPE
/*
 *  Reserved words
 */
%term <access_specifier> PRIVATE PROTECTED PUBLIC
%term <built_in_id> BOOL CHAR DOUBLE FLOAT INT LONG SHORT SIGNED UNSIGNED VOID WCHAR_T
%term <class_key> CLASS ENUM NAMESPACE STRUCT TYPENAME UNION
%term <cv_qualifiers> CONST VOLATILE
%term <decl_specifier_id> AUTO EXPLICIT EXPORT EXTERN FRIEND INLINE MUTABLE REGISTER STATIC TEMPLATE TYPEDEF USING VIRTUAL
%term <keyword> ASM BREAK CASE CATCH CONST_CAST CONTINUE DEFAULT DELETE DO DYNAMIC_CAST
%term <keyword> ELSE FALSE FOR GOTO IF NEW OPERATOR REINTERPRET_CAST RETURN
%term <keyword> SIZEOF STATIC_CAST SWITCH THIS THROW TRUE TRY TYPEID WHILE
/*
 *  Parametric values.
 */
%term <character_literal> CharacterLiteral
%term <floating_literal> FloatingLiteral
%term <identifier> Identifier
%term <integer_literal> IntegerLiteral
%term <number_literal> NumberLiteral
%term <string_literal> StringLiteral
/*
 *  The lexer need not treat ‘0’ as distinct from IntegerLiteral in the hope that pure-specifier can
 *  be distinguished, It isn’t. Semantic rescue from = constant-expression is necessary.
 *
 *  The lexer is not required to distinguish template or type names, although a slight simplification to the
 *  grammar and elaboration of the action rules could make good use of template name information.
 *
 *  In return for not needing to use semantic information, the lexer must support back-tracking, which
 *  is easily achieved by a simple linear buffer, a reference implementation of which may be found in the
 *  accompanying CxxParsing.cxx. Back-tracking is used to support:
 *
 *  Binary search for a consistent parse of the template/arithmetic ambiguity.
 *      start_search() initialises the search
 *      advance_search() iterates the search
 *      end_search() cleans up after a search
 *      template_test() maintains context during a search
 *
 *  Lookahead to resolve the inheritance/anonymous bit-field similarity
 *      mark() saves the starting context
 *      unmark() pops it
 *      rewind_colon() restores the context and forces the missing :
 *
 *  Lookahead to resolve type 1 function parameter ambiguities
 *      mark_type1() potentially marks the starting position
 *      mark() marks the pre { position
 *      remark() rewinds to the starting position
 *      unmark() pops the starting position
 *
 *  Note that lookaheads may nest.
 */

/*
 *  The parsing philosophy is unusual. The major ambiguities are resolved by creating a unified superset
 *  grammar rather than non-overlapping subgrammars. Thus the grammar for parameter-declaration covers an
 *  assignment-expression. Minor ambiguities whose resolution by supersetting would create more
 *  ambiguities are resolved the normal way with partitioned subgrammars.
 *  This eliminates the traditional expression/declaration and constructor/parenthesised declarator
 *  ambiguities at the syntactic level. A subsequent semantic level has to sort the problems out.
 *  The generality introduces four bogus ambiguities and defers the cast ambiguity for resolution
 *  once semantic information is available.
 *
 *  The C++ grammar comprises 558 rules and uses 894 states in yacc, with 0 unresolved conflicts.
 *  24 conflicts from 10 ambiguities are resolved by 8 %prec’s, so that yacc and bison report 0 conflicts.
 *
 *  The ambiguities are:
 *  1) dangling else resolved to inner-most if
 *      1 conflict in 1 state on else
 *  2) < as start-template or less-than
 *      1 conflict in 2 states on <
 *  3) a :: b :: c resolved to favour a::b::c rather than a::b ::c or a ::b::c
 *      1 conflicts in 1 state for ::
 *  4) pointer operators maximised at end of conversion id/new in preference to binary operators
 *      2 conflicts in 4 states on * and &
 *  5a) (a)@b resolved to favour binary a@b rather than cast unary (a)(@b)
 *  5b) (a)(b) resolved to favour cast rather than call
 *      8 conflicts in 1 state for the 8 prefix operators: 6 unaries and ( and [.
 *  6) enum name { resolved to enum-specifier rather than function
 *      1 conflict in 1 state on {
 *  7) class name { resolved to class-specifier rather than function
 *      1 conflict in 1 state on {
 *  8) extern “C” resolved to linkage-specification rather than declaration
 *      1 conflict in 1 state on StringLiteral
 *  9) class X : forced to go through base-clause look-ahead
 *      1 conflict in 1 state on :
 *  10) id : forced to label_statement rather than constructor_head
 *      0 conflicts - but causes a double state for 2)
 *  of which
 *      1 is a fundamental C conflict - always correctly resolved
 *          can be removed - see the Java spec
 *      2, 3, 4 are fundamental C++ conflicts
 *          2 always consistently resolved by iteration
 *          3 always correctly resolved
 *          4 always correctly resolved
 *      5 is a result of not using type information - deferred for semantic repair
 *      6,7 are caused by parsing over-generous superset - always correctly resolved
 *      8 is caused by parsing over-generous superset - always correctly resolved
 *          can be removed at the expense of 7 rules and 5 states.
 *      9 is a look-ahead trick - always correctly resolved
 *          could be removed by marking one token sooner
 *      10 is caused by parsing over-generous superset - always correctly resolved
 *
 *  The hard problem of distinguishing
 *      class A { class B : C, D, E {           -- A::B privately inherits C, D and E
 *      class A { class B : C, D, E ;           -- C is width of anon bit-field
 *  is resolved by using a lookahead that assumes inheritance and rewinds for the bit-field.
 *
 *  The potential shift-reduce conflict on > is resolved by flattening part of the expression grammar
 *  to know when the next > is template end or arithmetic >.
 *
 *  The grammar is SYNTACTICALLY context-free with respect to type. No semantic assistance is required
 *  during syntactic analysis. However the cast ambiguity is deferred and must be recovered
 *  after syntactic analysis of a statement has completed.
 *
 *  The grammar is SYNTACTICALLY context-free with respect to template-names. This is achieved by
 *  organising a binary search over all possible template/arithmetic ambiguities with respect to
 *  the enclosing statement. This is potentially exponentially inefficient but well-behaved in practice.
 *  Approximately 1% of statements trigger a search and approximately 1% of those are misparsed,
 *  requiring the semantic analysis to check and correct once template information is available.
 *  1.5 parse attempts are required on average per ambiguous statement.
 *
 *  The grammar supports type I function declarations at severe impediment to efficiency. A lookahead
 *  has to be performed after almost every non-statement close parenthesis. A one-line plus corollary
 *  change to postfix_expression is commented and strongly recommended to make this grammar as
 *  efficient as the rather large number of reduction levels permits.
 *
 *  Error recovery occurs mostly at the statement/declaration level. Recovery also occurs at
 *  the list-element level where this poses no hazard to statement/declaration level recovery.
 *  Note that since error propagation interacts with the lookaheads for template iteration or
 *  type 1 function arguments, introduction of finer grained error recovery may repair a false
 *  parse and so cause a misparse.
 *
 *  The following syntactic analysis errors occur, but are correctable semantically:
 *  (cast)unary-op expr         is parsed as (parenthesised)binary-op expr
 *      The semantic test should look for a binary/call with a (type) as its left child.
 *  (parenthesised)(arguments)  is parsed as (cast)(parenthesised)
 *      The semantic test should look for a cast with a non-type as its left child.
 *  template < and arithmetic < may be cross-parsed (unless semnatic help is provided)
 *      approximately 0.01% are misparsed, and must be sorted out - not easy.
 *
 *  The syntactic analysis defers the following ambiguities for semantic resolution:
 *  declaration/expression is parsed as a unified concept
 *      Use type and context to complete the parse.
 *  ~class-name                 is parsed as unary~ name
 *      The semantic test should look for ~ with a type as its child.
 *  delete[] expr               is parsed as delete []expr
 *      The semantic test should look for delete with a [] cast of its child.
 *  operator new/delete[]       are parsed as array of operator new/delete
 *      The semantic test should look for array of operator new/delete
 *      or activate the two extra commented rules in operator
 *  template of an explicit_instantiation is buried deep in the tree
 *      dig it out
 *  pure-specifier and constant-initializer are covered by assignment-expression
 *      just another of the deferred declaration/expression ambiguities
 *  sizeof and typeid don't distinguish type/value syntaxes
 *      probably makes life polymorphically easier
 */
%nonassoc SHIFT_THERE
%nonassoc SCOPE ELSE INC DEC '+' '-' '*' '&' '[' '{' '<' ':' StringLiteral
%nonassoc REDUCE_HERE_MOSTLY
%nonassoc '('
/*%nonassoc REDUCE_HERE */

%start translation_unit
%%

/*
 *  The %prec resolves the 14.2-3 ambiguity:
 *  Identifier '<' is forced to go through the is-it-a-template-name test
 *  All names absorb TEMPLATE with the name, so that no template_test is performed for them.
 *  This requires all potential declarations within an expression to perpetuate this policy
 *  and thereby guarantee the ultimate coverage of explicit_instantiation.
 *
 *  The %prec also resolves a conflict in identifier : which is forced to be a shift of a label for
 *  a labeled-statement rather than a reduction for the name of a bit-field or generalised constructor.
 *  This is pretty dubious syntactically but correct for all semantic possibilities.
 *  The shift is only activated when the ambiguity exists at the start of a statement. In this context
 *  a bit-field declaration or constructor definition are not allowed.
 */
identifier:                         Identifier
id:                                 identifier                          %prec SHIFT_THERE       /* Force < through test */
    |                               identifier template_test '+' template_argument_list '>'
    |                               identifier template_test '-'                                /* requeued < follows */
    |                               template_id
template_test:                      '<'             /* Queue '+' or '-' < as follow on */       { template_test(); }
global_scope:                       SCOPE
    |                               TEMPLATE global_scope
id_scope:                           id SCOPE
/*
 *  A :: B :: C; is ambiguous How much is type and how much name ?
 *  The %prec maximises the (type) length which is the 7.1-2 semantic constraint.
 */
nested_id:                          id                                  %prec SHIFT_THERE       /* Maximise length */
    |                               id_scope nested_id
scoped_id:                          nested_id
    |                               global_scope nested_id

/*
```

```
 *  destructor_id has to be held back to avoid a conflict with a one’s complement as per 5.3.1-9,
 *  It gets put back only when scoped or in a declarator_id, which is only used as an explicit member name.
 *  Declarations of an unscoped destructor are always parsed as a one’s complement.
 */
destructor_id:                      ‘~’ id
    |                               TEMPLATE destructor_id
special_function_id:                conversion_function_id
    |                               operator_function_id
    |                               TEMPLATE special_function_id
nested_special_function_id:         special_function_id
    |                               id_scope destructor_id
    |                               id_scope nested_special_function_id
scoped_special_function_id:         nested_special_function_id
    |                               global_scope nested_special_function_id

/* declarator-id is all names in all scopes, except reserved words */
declarator_id:                      scoped_id
    |                               scoped_special_function_id
    |                               destructor_id

/*  The standard defines pseudo-destructors in terms of type-name, which is class/enum/typedef, of which
 *  class-name is covered by a normal destructor. pseudo-destructors are supposed to support ~int() in
 *  templates, so the grammar here covers built-in names. Other names are covered by the lack of
 *  identifier/type discrimination.
 */
built_in_type_id:                   built_in_type_specifier
    |                               built_in_type_id built_in_type_specifier
pseudo_destructor_id:               built_in_type_id SCOPE ‘~’ built_in_type_id
    |                               ‘~’ built_in_type_id
    |                               TEMPLATE pseudo_destructor_id
nested_pseudo_destructor_id:        pseudo_destructor_id
    |                               id_scope nested_pseudo_destructor_id
scoped_pseudo_destructor_id:        nested_pseudo_destructor_id
    |                               global_scope scoped_pseudo_destructor_id

/*---------------------------------------------------------------------------------------------------
 * A.2 Lexical conventions
 *---------------------------------------------------------------------------------------------------*/
/*
 *  String concatenation is a phase 6, not phase 7 activity so does not really belong in the grammar.
 *  However it may be convenient to have it here to make this grammar fully functional.
 *  Unfortunately it introduces a conflict with the generalised parsing of extern “C” which
 *  is correctly resolved to maximise the string length as the token source should do anyway.
 */
string:                             StringLiteral
/*string:                           StringLiteral                          %prec SHIFT_THERE */
/*  |                               StringLiteral string  -- Perverse order avoids conflicts -- */
literal:                            IntegerLiteral
    |                               CharacterLiteral
    |                               FloatingLiteral
    |                               string
    |                               boolean_literal
boolean_literal:                    FALSE
    |                               TRUE

/*---------------------------------------------------------------------------------------------------
```

```
 * A.3 Basic concepts
 *---------------------------------------------------------------------------------------------*/
translation_unit:                   declaration_seq.opt

/*---------------------------------------------------------------------------------------------
 * A.4 Expressions
 *---------------------------------------------------------------------------------------------
 *  primary_expression covers an arbitrary sequence of all names with the exception of an unscoped destructor,
 *  which is parsed as its unary expression which is the correct disambiguation (when ambiguous).
 *  This eliminates the traditional A(B) meaning A B ambiguity, since we never have to tack an A onto
 *  the front of something that might start with (. The name length got maximised ab initio. The downside
 *  is that semantic interpretation must split the names up again.
 *
 *  Unification of the declaration and expression syntax means that unary and binary pointer declarator operators:
 *      int * * name
 *  are parsed as binary and unary arithmetic operators (int) * (*name). Since type information is not used
 *  ambiguities resulting from a cast
 *      (cast)*(value)
 *  are resolved to favour the binary rather than the cast unary to ease AST clean-up.
 *  The cast-call ambiguity must be resolved to the cast to ensure that (a)(b)c can be parsed.
 *
 *  The problem of the functional cast ambiguity
 *      name(arg)
 *  as call or declaration is avoided by maximising the name within the parsing kernel. So
 *  primary_id_expression picks up
 *      extern long int const var = 5;
 *  as an assignment to the syntax parsed as “extern long int const var”. The presence of two names is
 *  parsed so that “extern long into const” is distinguished from “var” considerably simplifying subsequent
 *  semantic resolution.
 *
 *  The generalised name is a concatenation of potential type-names (scoped identifiers or built-in sequences)
 *  plus optionally one of the special names such as an operator-function-id, conversion-function-id or
 *  destructor as the final name.
 */
primary_expression:                 literal
    |                               THIS
    |                               suffix_decl_specified_ids
/*  |                               SCOPE identifier                                  -- covered by suffix_decl_specified_ids */
/*  |                               SCOPE operator_function_id                        -- covered by suffix_decl_specified_ids */
/*  |                               SCOPE qualified_id                                -- covered by suffix_decl_specified_ids */
    |                               abstract_expression               %prec REDUCE_HERE_MOSTLY  /* Prefer binary to unary ops, cast to call */
/*  |                               id_expression                                     -- covered by suffix_decl_specified_ids */
/*
 *  Abstract-expression covers the () and [] of abstract-declarators.
 */
abstract_expression:                parenthesis_clause
    |                               ‘[‘ expression.opt ‘]’
    |                               TEMPLATE abstract_expression

/*  Type I function parameters are ambiguous with respect to the generalised name, so we have to do a lookahead following
 *  any function-like parentheses. This unfortunately hits normal code, so kill the -- lines and add the ++ lines for efficiency.
 *  Supporting Type I code under the superset causes perhaps 25% of lookahead parsing. Sometimes complete class definitions
 *  get traversed since they are valid generalised type I parameters!
 */
type1_parameters:       /*----*/    parameter_declaration_list ‘;’
    |                   /*----*/    type1_parameters parameter_declaration_list ‘;’
```

```
mark_type1:                         /* empty */                                         { mark_type1(); yyclearin; }
postfix_expression:                 primary_expression
/*  |                   /++++++/    postfix_expression parenthesis_clause */
    |                   /*----*/    postfix_expression parenthesis_clause mark_type1 '-'
    |                   /*----*/    postfix_expression parenthesis_clause mark_type1 '+' type1_parameters mark '{' error
                        /*----*/                    { yyerrok; yyclearin; remark_type1(); unmark(); unmark(); }
    |                   /*----*/    postfix_expression parenthesis_clause mark_type1 '+' type1_parameters mark error
                        /*----*/                    { yyerrok; yyclearin; remark_type1(); unmark(); unmark(); }
    |                   /*----*/    postfix_expression parenthesis_clause mark_type1 '+' error
                        /*----*/                    { yyerrok; yyclearin; remark_type1(); unmark(); }
    |                               postfix_expression '[' expression.opt ']'
/*  |                               destructor_id '[' expression.opt ']'                -- not semantically valid */
/*  |                               destructor_id parenthesis_clause                    -- omitted to resolve known ambiguity */
/*  |                               simple_type_specifier '(' expression_list.opt ')'   -- simple_type_specifier is a primary_expression */
    |                               postfix_expression '.' declarator_id
/*  |                               postfix_expression '.' TEMPLATE declarator_id       -- TEMPLATE absorbed into declarator_id. */
    |                               postfix_expression '.' scoped_pseudo_destructor_id
    |                               postfix_expression ARROW declarator_id
/*  |                               postfix_expression ARROW TEMPLATE declarator_id     -- TEMPLATE absorbed into declarator_id. */
    |                               postfix_expression ARROW scoped_pseudo_destructor_id
    |                               postfix_expression INC
    |                               postfix_expression DEC
    |                               DYNAMIC_CAST '<' type_id '>' '(' expression ')'
    |                               STATIC_CAST '<' type_id '>' '(' expression ')'
    |                               REINTERPRET_CAST '<' type_id '>' '(' expression ')'
    |                               CONST_CAST '<' type_id '>' '(' expression ')'
    |                               TYPEID parameters_clause
/*  |                               TYPEID '(' expression ')'                           -- covered by parameters_clause */
/*  |                               TYPEID '(' type_id ')'                              -- covered by parameters_clause */
expression_list.opt:                /* empty */
    |                               expression_list
expression_list:                    assignment_expression
    |                               expression_list ',' assignment_expression

unary_expression:                   postfix_expression
    |                               INC cast_expression
    |                               DEC cast_expression
    |                               ptr_operator cast_expression
/*  |                               '*' cast_expression                                 -- covered by ptr_operator */
/*  |                               '&' cast_expression                                 -- covered by ptr_operator */
/*  |                               decl_specifier_seq '*' cast_expression              -- covered by binary operator */
/*  |                               decl_specifier_seq '&' cast_expression              -- covered by binary operator */
    |                               suffix_decl_specified_scope star_ptr_operator cast_expression   /* covers e.g int ::type::* const t = 4 */

    |                               '+' cast_expression
    |                               '-' cast_expression
    |                               '!' cast_expression
    |                               '~' cast_expression
    |                               SIZEOF unary_expression
/*  |                               SIZEOF '(' type_id ')'                              -- covered by unary_expression */
    |                               new_expression
    |                               global_scope new_expression
    |                               delete_expression
    |                               global_scope delete_expression
/*  |                               DELETE '[' ']' cast_expression       -- covered by DELETE cast_expression since cast_expression covers ... */
/*  |                               SCOPE DELETE '[' ']' cast_expression //  ... abstract_expression cast_expression and so [] cast_expression */
```

```
delete_expression:                  DELETE cast_expression                                  /* also covers DELETE[] cast_expression */

new_expression:                     NEW new_type_id new_initializer.opt
    |                               NEW parameters_clause new_type_id new_initializer.opt
    |                               NEW parameters_clause
/*  |                               NEW '(' type-id ')'                                     -- covered by parameters_clause */
    |                               NEW parameters_clause parameters_clause new_initializer.opt
/*  |                               NEW '(' type-id ')' new_initializer                     -- covered by parameters_clause parameters_clause */
/*  |                               NEW parameters_clause '(' type-id ')'                   -- covered by parameters_clause parameters_clause */
                                                                                /* ptr_operator_seq.opt production reused to save a %prec */
new_type_id:                        type_specifier ptr_operator_seq.opt
    |                               type_specifier new_declarator
    |                               type_specifier new_type_id
new_declarator:                     ptr_operator new_declarator
    |                               direct_new_declarator
direct_new_declarator:              '[' expression ']'
    |                               direct_new_declarator '[' constant_expression ']'
new_initializer.opt:                /* empty */
    |                               '(' expression_list.opt ')'

/*  cast-expression is generalised to support a [] as well as a () prefix. This covers the omission of DELETE[] which when
 *  followed by a parenthesised expression was ambiguous. It also covers the gcc indexed array initialisation for free.
 */
cast_expression:                    unary_expression
    |                               abstract_expression cast_expression
/*  |                               '(' type_id ')' cast_expression                             -- covered by abstract_expression */

pm_expression:                      cast_expression
    |                               pm_expression DOT_STAR cast_expression
    |                               pm_expression ARROW_STAR cast_expression
multiplicative_expression:          pm_expression
    |                               multiplicative_expression star_ptr_operator pm_expression
    |                               multiplicative_expression '/' pm_expression
    |                               multiplicative_expression '%' pm_expression
additive_expression:                multiplicative_expression
    |                               additive_expression '+' multiplicative_expression
    |                               additive_expression '-' multiplicative_expression
shift_expression:                   additive_expression
    |                               shift_expression SHL additive_expression
    |                               shift_expression SHR additive_expression
relational_expression:              shift_expression
    |                               relational_expression '<' shift_expression
    |                               relational_expression '>' shift_expression
    |                               relational_expression LE shift_expression
    |                               relational_expression GE shift_expression
equality_expression:                relational_expression
    |                               equality_expression EQ relational_expression
    |                               equality_expression NE relational_expression
and_expression:                     equality_expression
    |                               and_expression '&' equality_expression
exclusive_or_expression:            and_expression
    |                               exclusive_or_expression '^' and_expression
inclusive_or_expression:            exclusive_or_expression
    |                               inclusive_or_expression '|' exclusive_or_expression
logical_and_expression:             inclusive_or_expression
```

```
    |                               logical_and_expression LOG_AND inclusive_or_expression
logical_or_expression:              logical_and_expression
    |                               logical_or_expression LOG_OR logical_and_expression
conditional_expression:             logical_or_expression
    |                               logical_or_expression '?' expression ':' assignment_expression


/*  assignment-expression is generalised to cover the simple assignment of a braced initializer in order to contribute to the
 *  coverage of parameter-declaration and init-declaration.
 */
assignment_expression:              conditional_expression
    |                               logical_or_expression assignment_operator assignment_expression
    |                               logical_or_expression '=' braced_initializer
    |                               throw_expression
assignment_operator:                '=' | ASS_ADD | ASS_AND | ASS_DIV | ASS_MOD | ASS_MUL | ASS_OR | ASS_SHL | ASS_SHR | ASS_SUB | ASS_XOR

/*  expression is widely used and usually single-element, so the reductions are arranged so that a
 *  single-element expression is returned as is. Multi-element expressions are parsed as a list that
 *  may then behave polymorphically as an element or be compacted to an element. */
expression.opt:                     /* empty */
    |                               expression
expression:                         assignment_expression
    |                               expression_list ',' assignment_expression
constant_expression:                conditional_expression

/*  The grammar is repeated for when the parser stack knows that the next > must end a template.
 */
templated_relational_expression:    shift_expression
    |                               templated_relational_expression '<' shift_expression
    |                               templated_relational_expression LE shift_expression
    |                               templated_relational_expression GE shift_expression
templated_equality_expression:      templated_relational_expression
    |                               templated_equality_expression EQ templated_relational_expression
    |                               templated_equality_expression NE templated_relational_expression
templated_and_expression:           templated_equality_expression
    |                               templated_and_expression '&' templated_equality_expression
templated_exclusive_or_expression:  templated_and_expression
    |                               templated_exclusive_or_expression '^' templated_and_expression

templated_inclusive_or_expression:  templated_exclusive_or_expression
    |                               templated_inclusive_or_expression '|' templated_exclusive_or_expression

templated_logical_and_expression:   templated_inclusive_or_expression
    |                               templated_logical_and_expression LOG_AND templated_inclusive_or_expression

templated_logical_or_expression:    templated_logical_and_expression
    |                               templated_logical_or_expression LOG_OR templated_logical_and_expression

templated_conditional_expression:   templated_logical_or_expression
    |                               templated_logical_or_expression '?' templated_expression ':' templated_assignment_expression

templated_assignment_expression:    templated_conditional_expression
    |                               templated_logical_or_expression assignment_operator templated_assignment_expression

    |                               templated_throw_expression
templated_expression:               templated_assignment_expression
```

```
    |                                   templated_expression_list ',' templated_assignment_expression

templated_expression_list:          templated_assignment_expression
    |                                   templated_expression_list ',' templated_assignment_expression

/*---------------------------------------------------------------------------------------------------
 * A.5 Statements
 *---------------------------------------------------------------------------------------------------
 *  Parsing statements is easy once simple_declaration has been generalised to cover expression_statement.
 */
looping_statement:                  start_search looped_statement                               { end_search(); }
looped_statement:                   statement
    |                               advance_search '+' looped_statement
    |                               advance_search '-'
statement:                          control_statement
/*  |                               expression_statement                                        -- covered by declaration_statement */
    |                               compound_statement
    |                               declaration_statement
    |                               try_block
control_statement:                  labeled_statement
    |                               selection_statement
    |                               iteration_statement
    |                               jump_statement
labeled_statement:                  identifier ':' looping_statement
    |                               CASE constant_expression ':' looping_statement
    |                               DEFAULT ':' looping_statement
/*expression_statement:             expression.opt ';'                                          -- covered by declaration_statement */
compound_statement:                 '{' statement_seq.opt '}'
    |                               '{' statement_seq.opt looping_statement '#' bang error '}'  { UNBANG("Bad statement-seq."); }
statement_seq.opt:                  /* empty */
    |                               statement_seq.opt looping_statement
    |                               statement_seq.opt looping_statement '#' bang error ';'      { UNBANG("Bad statement."); }
/*
 *  The dangling else conflict is resolved to the innermost if.
 */
selection_statement:                IF '(' condition ')' looping_statement    %prec SHIFT_THERE
    |                               IF '(' condition ')' looping_statement ELSE looping_statement
    |                               SWITCH '(' condition ')' looping_statement
condition.opt:                      /* empty */
    |                               condition
condition:                          parameter_declaration_list
/*  |                               expression                                                  -- covered by parameter_declaration_list */
/*  |                               type_specifier_seq declarator '=' assignment_expression     -- covered by parameter_declaration_list */
iteration_statement:                WHILE '(' condition ')' looping_statement
    |                               DO looping_statement WHILE '(' expression ')' ';'
    |                               FOR '(' for_init_statement condition.opt ';' expression.opt ')' looping_statement

for_init_statement:                 simple_declaration
/*  |                               expression_statement                                        -- covered by simple_declaration */
jump_statement:                     BREAK ';'
    |                               CONTINUE ';'
    |                               RETURN expression.opt ';'
    |                               GOTO identifier ';'
declaration_statement:              block_declaration

/*---------------------------------------------------------------------------------------------------
```

```
  * A.6 Declarations
  *---------------------------------------------------------------------------------------*/
compound_declaration:              '{' nest declaration_seq.opt '}'                          { unnest(); }
    |                              '{' nest declaration_seq.opt util looping_declaration '#' bang error '}'
                                                                                             { unnest(); UNBANG("Bad declaration-seq."); }
declaration_seq.opt:               /* empty */
    |                              declaration_seq.opt util looping_declaration
    |                              declaration_seq.opt util looping_declaration '#' bang error ';' { UNBANG("Bad declaration."); }
looping_declaration:               start_search1 looped_declaration                          { end_search(); }
looped_declaration:                declaration
    |                              advance_search '+' looped_declaration
    |                              advance_search '-'
declaration:                       block_declaration
    |                              function_definition
    |                              template_declaration
/*  |                              explicit_instantiation                                    -- covered by relevant declarations */
    |                              explicit_specialization
    |                              specialised_declaration
specialised_declaration:           linkage_specification
    |                              namespace_definition
    |                              TEMPLATE specialised_declaration
block_declaration:                 simple_declaration
    |                              specialised_block_declaration
specialised_block_declaration:     asm_definition
    |                              namespace_alias_definition
    |                              using_declaration
    |                              using_directive
    |                              TEMPLATE specialised_block_declaration
simple_declaration:                ';'
    |                              init_declaration ';'
    |                              init_declarations ';'
    |                              decl_specifier_prefix simple_declaration

/*  A decl-specifier following a ptr_operator provokes a shift-reduce conflict for
 *      * const name
 *  which is resolved in favour of the pointer, and implemented by providing versions
 *  of decl-specifier guaranteed not to start with a cv_qualifier.
 *
 *  decl-specifiers are implemented type-centrically. That is the semantic constraint
 *  that there must be a type is exploited to impose structure, but actually eliminate
 *  very little syntax. built-in types are multi-name and so need a different policy.
 *
 *  non-type decl-specifiers are bound to the left-most type in a decl-specifier-seq,
 *  by parsing from the right and attaching suffixes to the right-hand type. Finally
 *  residual prefixes attach to the left.
 */
suffix_built_in_decl_specifier.raw: built_in_type_specifier
    |                              suffix_built_in_decl_specifier.raw built_in_type_specifier
    |                              suffix_built_in_decl_specifier.raw decl_specifier_suffix
suffix_built_in_decl_specifier:    suffix_built_in_decl_specifier.raw
    |                              TEMPLATE suffix_built_in_decl_specifier
suffix_named_decl_specifier:       scoped_id
    |                              elaborate_type_specifier
    |                              suffix_named_decl_specifier decl_specifier_suffix
suffix_named_decl_specifier.bi:    suffix_named_decl_specifier
    |                              suffix_named_decl_specifier suffix_built_in_decl_specifier.raw
```

```
suffix_named_decl_specifiers:       suffix_named_decl_specifier.bi
    |                               suffix_named_decl_specifiers suffix_named_decl_specifier.bi
suffix_named_decl_specifiers.sf:    scoped_special_function_id          /* operators etc */
    |                               suffix_named_decl_specifiers
    |                               suffix_named_decl_specifiers scoped_special_function_id
suffix_decl_specified_ids:          suffix_built_in_decl_specifier
    |                               suffix_built_in_decl_specifier suffix_named_decl_specifiers.sf
    |                               suffix_named_decl_specifiers.sf
suffix_decl_specified_scope:        suffix_named_decl_specifiers SCOPE
    |                               suffix_built_in_decl_specifier suffix_named_decl_specifiers SCOPE
    |                               suffix_built_in_decl_specifier SCOPE

decl_specifier_affix:               storage_class_specifier
    |                               function_specifier
    |                               FRIEND
    |                               TYPEDEF
    |                               cv_qualifier

decl_specifier_suffix:              decl_specifier_affix

decl_specifier_prefix:              decl_specifier_affix
    |                               TEMPLATE decl_specifier_prefix

storage_class_specifier:            REGISTER | STATIC | MUTABLE
    |                               EXTERN                  %prec SHIFT_THERE                   /* Prefer linkage specification */
    |                               AUTO

function_specifier:                 EXPLICIT
    |                               INLINE
    |                               VIRTUAL

type_specifier:                     simple_type_specifier
    |                               elaborate_type_specifier
    |                               cv_qualifier

elaborate_type_specifier:           class_specifier
    |                               enum_specifier
    |                               elaborated_type_specifier
    |                               TEMPLATE elaborate_type_specifier
simple_type_specifier:              scoped_id
    |                               built_in_type_specifier
built_in_type_specifier:            CHAR | WCHAR_T | BOOL | SHORT | INT | LONG | SIGNED | UNSIGNED | FLOAT | DOUBLE | VOID

/*
 *  The over-general use of declaration_expression to cover decl-specifier-seq.opt declarator in a function-definition means that
 *      class X { };
 *  could be a function-definition or a class-specifier.
 *      enum X { };
 *  could be a function-definition or an enum-specifier.
 *  The function-definition is not syntactically valid so resolving the false conflict in favour of the
 *  elaborated_type_specifier is correct.
 */
elaborated_type_specifier:          elaborated_class_specifier
    |                               elaborated_enum_specifier
    |                               TYPENAME scoped_id
```

```
elaborated_enum_specifier:          ENUM scoped_id               %prec SHIFT_THERE
enum_specifier:                     ENUM scoped_id enumerator_clause
    |                               ENUM enumerator_clause
enumerator_clause:                  '{' enumerator_list_ecarb
    |                               '{' enumerator_list enumerator_list_ecarb
    |                               '{' enumerator_list ',' enumerator_definition_ecarb
enumerator_list_ecarb:              '}'
    |                               bang error '}'                                            { UNBANG("Bad enumerator-list."); }
enumerator_definition_ecarb:        '}'
    |                               bang error '}'                                            { UNBANG("Bad enumerator-definition."); }
enumerator_definition_filler:       /* empty */
    |                               bang error ','                                            { UNBANG("Bad enumerator-definition."); }
enumerator_list_head:               enumerator_definition_filler
    |                               enumerator_list ',' enumerator_definition_filler
enumerator_list:                    enumerator_list_head enumerator_definition
enumerator_definition:              enumerator
    |                               enumerator '=' constant_expression
enumerator:                         identifier

namespace_definition:               NAMESPACE scoped_id compound_declaration
    |                               NAMESPACE compound_declaration
namespace_alias_definition:         NAMESPACE scoped_id '=' scoped_id ';'

using_declaration:                  USING declarator_id ';'
    |                               USING TYPENAME declarator_id ';'

using_directive:                    USING NAMESPACE scoped_id ';'
asm_definition:                     ASM '(' string ')' ';'
linkage_specification:              EXTERN string looping_declaration
    |                               EXTERN string compound_declaration

/*---------------------------------------------------------------------------------------------------
 * A.7 Declarators
 *---------------------------------------------------------------------------------------------------*/
/*init-declarator is named init_declaration to reflect the embedded decl-specifier-seq.opt*/
init_declarations:                  assignment_expression ',' init_declaration
    |                               init_declarations ',' init_declaration
init_declaration:                   assignment_expression
/*  |                               assignment_expression '=' initializer_clause              -- covered by assignment_expression */
/*  |                               assignment_expression '(' expression_list ')'             -- covered by another set of call arguments */

/*declarator:                                                                                 -- covered by assignment_expression */
/*direct_declarator:                                                                          -- covered by postfix_expression */

star_ptr_operator:                  '*'
    |                               star_ptr_operator cv_qualifier
nested_ptr_operator:                star_ptr_operator
    |                               id_scope nested_ptr_operator
ptr_operator:                       '&'
    |                               nested_ptr_operator
    |                               global_scope nested_ptr_operator
ptr_operator_seq:                   ptr_operator
    |                               ptr_operator ptr_operator_seq
/* Independently coded to localise the shift-reduce conflict: sharing just needs another %prec */
ptr_operator_seq.opt:               /* empty */                       %prec SHIFT_THERE       /* Maximise type length */
    |                               ptr_operator ptr_operator_seq.opt
```

```
cv_qualifier_seq.opt:               /* empty */
    |                               cv_qualifier_seq.opt cv_qualifier
cv_qualifier:                       CONST | VOLATILE

/*type_id                                                           -- also covered by parameter declaration */
type_id:                            type_specifier abstract_declarator.opt
    |                               type_specifier type_id

/*abstract_declarator:                                              -- also covered by parameter declaration */
abstract_declarator.opt:            /* empty */
    |                               ptr_operator abstract_declarator.opt
    |                               direct_abstract_declarator
direct_abstract_declarator.opt:     /* empty */
    |                               direct_abstract_declarator
direct_abstract_declarator:         direct_abstract_declarator.opt parenthesis_clause
    |                               direct_abstract_declarator.opt '[' ']'
    |                               direct_abstract_declarator.opt '[' constant_expression ']'
/*  |                               '(' abstract_declarator ')'     -- covered by parenthesis_clause */

parenthesis_clause:                 parameters_clause cv_qualifier_seq.opt
    |                               parameters_clause cv_qualifier_seq.opt exception_specification
parameters_clause:                  '(' parameter_declaration_clause ')'
/* parameter_declaration_clause also covers init_declaration, type_id, declarator and abstract_declarator. */
parameter_declaration_clause:       /* empty */
    |                               parameter_declaration_list
    |                               parameter_declaration_list ELLIPSIS
parameter_declaration_list:         parameter_declaration
    |                               parameter_declaration_list ',' parameter_declaration

/*
 * A typed abstract qualifier such as
 *      Class * ...
 * looks like a multiply, so pointers are parsed as their binary operation equivalents that
 * ultimately terminate with a degenerate right hand term.
 */
abstract_pointer_declaration:       ptr_operator_seq
    |                               multiplicative_expression star_ptr_operator ptr_operator_seq.opt
abstract_parameter_declaration:     abstract_pointer_declaration
    |                               and_expression '&'
    |                               and_expression '&' abstract_pointer_declaration
special_parameter_declaration:      abstract_parameter_declaration
    |                               abstract_parameter_declaration '=' assignment_expression
    |                               ELLIPSIS
parameter_declaration:              assignment_expression
    |                               special_parameter_declaration
    |                               decl_specifier_prefix parameter_declaration

/*  The grammar is repeated for use within template <>
 */
templated_parameter_declaration:    templated_assignment_expression
    |                               templated_abstract_declaration
    |                               templated_abstract_declaration '=' templated_assignment_expression

    |                               decl_specifier_prefix templated_parameter_declaration
templated_abstract_declaration:     abstract_pointer_declaration
```

```
    |                               templated_and_expression '&'
    |                               templated_and_expression '&' abstract_pointer_declaration

/*  function_definition includes constructor, destructor, implicit int definitions too.
 *  A local destructor is successfully parsed as a function-declaration but the ~ was treated as a unary operator.
 *  constructor_head is the prefix ambiguity between a constructor and a member-init-list starting with a bit-field.
 */
function_definition:                ctor_definition
    |                               func_definition
func_definition:                    assignment_expression function_try_block
    |                               assignment_expression function_body
    |                               decl_specifier_prefix func_definition
ctor_definition:                    constructor_head function_try_block
    |                               constructor_head function_body
    |                               decl_specifier_prefix ctor_definition
constructor_head:                   bit_field_init_declaration
    |                               constructor_head ',' assignment_expression
function_try_block:                 TRY function_block handler_seq
function_block:                     ctor_initializer.opt function_body
function_body:                      compound_statement

/*
 *  An = initializer looks like an extended assignment_expression.
 *  An () initializer looks like a function call.
 *  initializer is therefore flattened into its generalised customers.
 *initializer:                      '=' initializer_clause                                      -- flattened into caller
 *  |                               '(' expression_list ')'                                     -- flattened into caller */
initializer_clause:                 assignment_expression
    |                               braced_initializer
braced_initializer:                 '{' initializer_list '}'
    |                               '{' initializer_list ',' '}'
    |                               '{' '}'
    |                               '{' looping_initializer_clause '#' bang error '}'           { UNBANG("Bad initializer_clause."); }
    |                               '{' initializer_list ',' looping_initializer_clause '#' bang error '}'
                                                                                                { UNBANG("Bad initializer_clause."); }
initializer_list:                   looping_initializer_clause
    |                               initializer_list ',' looping_initializer_clause
looping_initializer_clause:         start_search looped_initializer_clause                      { end_search(); }
looped_initializer_clause:          initializer_clause
    |                               advance_search '+' looped_initializer_clause
    |                               advance_search '-'

/*---------------------------------------------------------------------------------------------------
 * A.8 Classes
 *---------------------------------------------------------------------------------------------------
 *
 *  An anonymous bit-field declaration may look very like inheritance:
 *      const int B = 3;
 *      class A : B ;
 *  The two usages are too distant to try to create and enforce a common prefix so we have to resort to
 *  a parser hack by backtracking. Inheritance is much the most likely so we mark the input stream context
 *  and try to parse a base-clause. If we successfully reach a { the base-clause is ok and inheritance was
 *  the correct choice so we unmark and continue. If we fail to find the { an error token causes back-tracking
 *  to the alternative parse in elaborated_type_specifier which regenerates the : and declares unconditional success.
 */
colon_mark:                         ':'                                                         { mark(); }
```

```
elaborated_class_specifier:         class_key scoped_id                   %prec SHIFT_THERE
    |                               class_key scoped_id colon_mark error                    { rewind_colon(); }
class_specifier_head:               class_key scoped_id colon_mark base_specifier_list '{'      { unmark(); }
    |                               class_key ':' base_specifier_list '{'
    |                               class_key scoped_id '{'
    |                               class_key '{'
class_key:                          CLASS | STRUCT | UNION
class_specifier:                    class_specifier_head member_specification.opt '}'
    |                               class_specifier_head member_specification.opt util looping_member_declaration '#' bang error '}'
                                            { UNBANG("Bad member_specification.opt."); }
member_specification.opt:           /* empty */
    |                               member_specification.opt util looping_member_declaration
    |                               member_specification.opt util looping_member_declaration '#' bang error ';'
                                                                                            { UNBANG("Bad member-declaration."); }
looping_member_declaration:         start_search looped_member_declaration                  { end_search(); }
looped_member_declaration:          member_declaration
    |                               advance_search '+' looped_member_declaration
    |                               advance_search '-'
member_declaration:                 accessibility_specifier
    |                               simple_member_declaration
    |                               function_definition
/*  |                               function_definition ';'                                 -- trailing ; covered by null declaration */
/*  |                               qualified_id ';'                                        -- covered by simple_member_declaration */
    |                               using_declaration
    |                               template_declaration

/*  The generality of constructor names (there need be no parenthesised argument list) means that that
 *          name : f(g), h(i)
 *  could be the start of a constructor or the start of an anonymous bit-field. An ambiguity is avoided by
 *  parsing the ctor-initializer of a function_definition as a bit-field.
 */
simple_member_declaration:          ';'
    |                               assignment_expression ';'
    |                               constructor_head ';'
    |                               member_init_declarations ';'
    |                               decl_specifier_prefix simple_member_declaration
member_init_declarations:           assignment_expression ',' member_init_declaration
    |                               constructor_head ',' bit_field_init_declaration
    |                               member_init_declarations ',' member_init_declaration
member_init_declaration:            assignment_expression
/*  |                               assignment_expression '=' initializer_clause            -- covered by assignment_expression */
/*  |                               assignment_expression '(' expression_list ')'           -- covered by another set of call arguments */
    |                               bit_field_init_declaration
accessibility_specifier:            access_specifier ':'
bit_field_declaration:              assignment_expression ':' bit_field_width
    |                               ':' bit_field_width
bit_field_width:                    logical_or_expression
/*  |                               logical_or_expression '?' expression ':' assignment_expression  -- has SR conflict w.r.t later = */
    |                               logical_or_expression '?' bit_field_width ':' bit_field_width
bit_field_init_declaration:         bit_field_declaration
    |                               bit_field_declaration '=' initializer_clause

/*---------------------------------------------------------------------------------------------------
 * A.9 Derived classes
 *---------------------------------------------------------------------------------------------------*/
/*base_clause:                      ':' base_specifier_list                                 -- flattened */
```

```
base_specifier_list:                base_specifier
    |                               base_specifier_list ‘,’ base_specifier
base_specifier:                     scoped_id
    |                               access_specifier base_specifier
    |                               VIRTUAL base_specifier
access_specifier:                   PRIVATE | PROTECTED | PUBLIC

/*----------------------------------------------------------------------------------------------------
 * A.10 Special member functions
 *----------------------------------------------------------------------------------------------------*/
conversion_function_id:             OPERATOR conversion_type_id
conversion_type_id:                 type_specifier ptr_operator_seq.opt
    |                               type_specifier conversion_type_id
/*
 *  Ctor-initialisers can look like a bit field declaration, given the generalisation of names:
 *      Class(Type) : m1(1), m2(2) { }
 *      NonClass(bit_field) : int(2), second_variable, ...
 *  The grammar below is used within a function_try_block or function_definition.
 *  See simple_member_declaration for use in normal member function_definition.
 */
ctor_initializer.opt:               /* empty */
    |                               ctor_initializer
ctor_initializer:                   ‘:’ mem_initializer_list
    |                               ‘:’ mem_initializer_list bang error                          { UNBANG(“Bad ctor-initializer.”); }
mem_initializer_list:               mem_initializer
    |                               mem_initializer_list_head mem_initializer
mem_initializer_list_head:          mem_initializer_list ‘,’
    |                               mem_initializer_list bang error ‘,’                          { UNBANG(“Bad mem-initializer.”); }
mem_initializer:                    mem_initializer_id ‘(‘ expression_list.opt ‘)’
mem_initializer_id:                 scoped_id

/*----------------------------------------------------------------------------------------------------
 * A.11 Overloading
 *----------------------------------------------------------------------------------------------------*/
operator_function_id:               OPERATOR operator
/*
 *  It is not clear from the ANSI standard whether spaces are permitted in delete[]. If not then it can
 *  be recognised and returned as DELETE_ARRAY by the lexer. Assuming spaces are permitted there is an
 *  ambiguity created by the over generalised nature of expressions. operator new is a valid delarator-id
 *  which we may have an undimensioned array of. Semantic rubbish, but syntactically valid. Since the
 *  array form is covered by the declarator consideration we can exclude the operator here. The need
 *  for a semantic rescue can be eliminated at the expense of a couple of shift-reduce conflicts by
 *  removing the comments on the next four lines.
 */
operator:             /*++++*/      NEW
    |                 /*++++*/      DELETE
/*  |                 / ---- /      NEW                 %prec SHIFT_THERE
/*  |                 / ---- /      DELETE              %prec SHIFT_THERE
/*  |                 / ---- /      NEW ‘[‘ ‘]’                                              -- Covered by array of OPERATOR NEW */
/*  |                 / ---- /      DELETE ‘[‘ ‘]’                                           -- Covered by array of OPERATOR DELETE */
    |                               ‘+’
    |                               ‘-’
    |                               ‘*’
    |                               ‘/’
    |                               ‘%’
    |                               ‘^’
    |                                   '&'
    |                                   '|'
    |                                   '~'
    |                                   '!'
    |                                   '='
    |                                   '<'
    |                                   '>'
    |                                   ASS_ADD
    |                                   ASS_SUB
    |                                   ASS_MUL
    |                                   ASS_DIV
    |                                   ASS_MOD
    |                                   ASS_XOR
    |                                   ASS_AND
    |                                   ASS_OR
    |                                   SHL
    |                                   SHR
    |                                   ASS_SHR
    |                                   ASS_SHL
    |                                   EQ
    |                                   NE
    |                                   LE
    |                                   GE
    |                                   LOG_AND
    |                                   LOG_OR
    |                                   INC
    |                                   DEC
    |                                   ','
    |                                   ARROW_STAR
    |                                   ARROW
    |                                   '(' ')'
    |                                   '[' ']'

/*----------------------------------------------------------------------------------------------------
 * A.12 Templates
 *----------------------------------------------------------------------------------------------------*/
template_declaration:               template_parameter_clause declaration
    |                               EXPORT template_declaration
template_parameter_clause:          TEMPLATE '<' template_parameter_list '>'
template_parameter_list:            template_parameter
    |                               template_parameter_list ',' template_parameter
template_parameter:                 simple_type_parameter
    |                               simple_type_parameter '=' type_id
    |                               templated_type_parameter
    |                               templated_type_parameter '=' identifier
    |                               templated_parameter_declaration
    |                               bang error                                             { UNBANG("Bad template-parameter."); }
simple_type_parameter:              CLASS
/*  |                               CLASS identifier                                       -- covered by parameter_declaration */
    |                               TYPENAME
/*  |                               TYPENAME identifier                                    -- covered by parameter_declaration */
templated_type_parameter:           template_parameter_clause CLASS
    |                               template_parameter_clause CLASS identifier
template_id:                        TEMPLATE identifier '<' template_argument_list '>'
    |                               TEMPLATE template_id
/*
```

```
 *  template-argument is evaluated using a templated...expression so that > resolves to end of template.
 */
template_argument_list:             template_argument
    |                               template_argument_list ‘,’ template_argument
template_argument:                  templated_parameter_declaration
/*  |                               type_id                                                 -- covered by templated_parameter_declaration */
/*  |                               template_name                                           -- covered by templated_parameter_declaration */
/*  |                               error                                                   -- must allow template failure to re-search */

/*
 *  Generalised naming makes identifier a valid declaration, so TEMPLATE identifier is too.
 *  The TEMPLATE prefix is therefore folded into all names, parenthesis_clause and decl_specifier_prefix.
 */
/*explicit_instantiation:           TEMPLATE declaration */
explicit_specialization:            TEMPLATE ‘<’ ‘>’ declaration

/*---------------------------------------------------------------------------------------------------
 * A.13 Exception Handling
 *---------------------------------------------------------------------------------------------------*/
try_block:                          TRY compound_statement handler_seq
/*function_try_block:                                                                       -- moved near function_block */
handler_seq:                        handler
    |                               handler handler_seq
handler:                            CATCH ‘(’ exception_declaration ‘)’ compound_statement
exception_declaration:              parameter_declaration
/*                                  ELLIPSIS                                                -- covered by parameter_declaration */
throw_expression:                   THROW
    |                               THROW assignment_expression
templated_throw_expression:         THROW
    |                               THROW templated_assignment_expression
exception_specification:            THROW ‘(’ ‘)’
    |                               THROW ‘(’ type_id_list ‘)’
type_id_list:                       type_id
    |                               type_id_list ‘,’ type_id

/*---------------------------------------------------------------------------------------------------
 * Back-tracking and context support
 *---------------------------------------------------------------------------------------------------*/
advance_search:                     error               { yyerrok; yyclearin; advance_search(); } /* Rewind and queue ‘+’ or ‘-’ ‘#’ */
bang:                               /* empty */         { BANG(); }   /* set flag to suppress “parse error” */
mark:                               /* empty */         { mark(); }        /* Push lookahead and input token stream context onto a stack */
nest:                               /* empty */         { nest(); }        /* Push a declaration nesting depth onto the parse stack */
start_search:                       /* empty */         { start_search(false); }    /* Create/reset binary search context */
start_search1:                      /* empty */         { start_search(true); }     /* Create/reset binary search context */
util:                               /* empty */           /* Get current utility mode */
```

C FOG Grammar

The complete FOG grammar implemented using the superset approach outlined for FOG is presented in this appendix.

The presented grammar has been derived automatically from FogParser.y by a sed script to

- remove C++ specific grammar
- remove irrelevant action rules
- simplify relevant action rules
- remove implementation specific $ clutter

The resulting text is acceptable to yacc and has 0 unresolved conflicts.

It is available from

http://www.computing.surrey.ac.uk/research/dsrg/fog/FogGrammar.y

```
/* This is a yacc-able parser for the entire FOG grammar with no unresolved conflicts. */
/* The parse is SYNTACTICALLY consistent and requires no template or type name assistance.
 * The grammar in the C++ standard notes that its grammar is a superset of the true
 * grammar requiring semantic constraints to resolve ambiguities. This grammar is a really big
 * superset unifying expressions and declarations, eliminating the type/non-type distinction,
 * and iterating to find a consistent solution to the template/arith,metoic < ambiguity.
 * As a result the grammar is much simpler, but requires the missing semantic constraints to be
 * performed in a subsequent semantic pass, which is of course where they belong. This grammar will
 * support conversion of C++ tokens into an Abstract Syntax Tree. A lot of further work is required to
 * make that tree useful.
 *
 *  Author:         E.D.Willink             Ed.Willink@rrl.co.uk
 *  Date:           19-Nov-1999
 */
/*
 * The lexer (and/or a preprocessor) is expected to identify the following
 *
 *  Punctuation:
 */
%type <keyword> ‘+’ ‘-’ ‘*’ ‘/’ ‘%’ ‘^’ ‘&’ ‘|’ ‘~’ ‘!’ ‘<‘  ‘>’ ‘=’ ‘:’ ‘[‘ ‘]’ ‘{‘ ‘}’ ‘(‘ ‘)’
%type <keyword> ‘?’ ‘.’ ‘\’’ ‘\”’ ‘\\’ ‘@’ ‘$’ ‘;’ ‘,’
/*
 *  Punctuation sequences
 */
%term <keyword> ARROW ARROW_STAR DEC EQ GE INC LE LOG_AND LOG_OR NE SHL SHR
%term <keyword> ASS_ADD ASS_AND ASS_DIV ASS_MOD ASS_MUL ASS_OR ASS_SHL ASS_SHR ASS_SUB ASS_XOR
%term <keyword> DOT_STAR ELLIPSIS SCOPE
/*
 *  Reserved words
 */
%term <access_specifier> PRIVATE PROTECTED PUBLIC
%term <built_in_id> BOOL CHAR DOUBLE FLOAT INT LONG SHORT SIGNED UNSIGNED VOID WCHAR_T
%term <class_key> CLASS ENUM NAMESPACE STRUCT TYPENAME UNION
%term <cv_qualifiers> CONST VOLATILE
%term <decl_specifier_id> AUTO EXPLICIT EXPORT EXTERN FRIEND INLINE MUTABLE REGISTER STATIC TEMPLATE TYPEDEF USING VIRTUAL
%term <keyword> ASM BREAK CASE CATCH CONST_CAST CONTINUE DEFAULT DELETE DO DYNAMIC_CAST
%term <keyword> ELSE FALSE FOR GOTO IF NEW OPERATOR REINTERPRET_CAST RETURN
%term <keyword> SIZEOF STATIC_CAST SWITCH THIS THROW TRUE TRY TYPEID WHILE
/*
 *  Parametric values.
 */
%term <character_literal> CharacterLiteral
%term <floating_literal> FloatingLiteral
%term <identifier> Identifier
%term <integer_literal> IntegerLiteral
%term <number_literal> NumberLiteral
%term <string_literal> StringLiteral
/*
 *  FOG non-reserved word identifier extensions
 */
%term <built_in_id> BuiltInTypeSpecifier
%term <meta_type> MetaType
%term <name> TreeLiteral
%term <name> DERIVED EMIT FILE FROZEN GUARD IMPLEMENTATION INCLUDE INTERFACE
%term <name> NOGUARD NOIMPLEMENTATION OVERLOAD PATH POOL
%term <name> PREFIX PURE SUFFIX UTILITY
```

```
%term <segment> BODY ENTRY EXIT POST PRE
/*
 *  The lexer need not treat ‘0’ as distinct from IntegerLiteral in the hope that pure-specifier can
 *  be distinguished, It isn’t. Semantic rescue from = constant-expression is necessary.
 *
 *  The lexer is not required to distinguish template or type names, although a slight simplification to the
 *  grammar and elaboration of the action rules could make good use of template name information.
 *
 *  In return for not needing to use semantic information, the lexer must support back-tracking, which
 *  is easily achieved by a simple linear buffer, a reference implementation of which may be found in the
 *  accompanying CxxParsing.cxx. Back-tracking is used to support:
 *
 *  Binary search for a consistent parse of the template/arithmetic ambiguity.
 *      start_search() initialises the search
 *      advance_search() iterates the search
 *      end_search() cleans up after a search
 *      template_test() maintains context during a search
 *
 *  Lookahead to resolve the inheritance/anonymous bit-field similarity
 *      mark() saves the starting context
 *      unmark() pops it
 *      rewind_colon() restores the context and forces the missing :
 *
 *  Lookahead to resolve type 1 function parameter ambiguities
 *      mark_type1() potentially marks the starting position
 *      mark() marks the pre { position
 *      remark() rewinds to the starting position
 *      unmark() pops the starting position
 *
 *  Note that lookaheads may nest.
 */

/*
 *  The parsing philosophy is unusual. The major ambiguities are resolved by creating a unified superset
 *  grammar rather than non-overlapping subgrammars. Thus the grammar for parameter-declaration covers an
 *  assignment-expression. Minor ambiguities whose resolution by supersetting would create more
 *  ambiguities are resolved the normal way with partitioned subgrammars.
 *  This eliminates the traditional expression/declaration and constructor/parenthesised declarator
 *  ambiguities at the syntactic level. A subsequent semantic level has to sort the problems out.
 *  The generality introduces four bogus ambiguities and defers the cast ambiguity for resolution
 *  once semantic information is available.
 *
 *  The FOG grammar comprises 939 rules and uses 1550 states in yacc, with 0 unresolved conflicts.
 *  32 conflicts from 15 ambiguities are resolved by 15 %prec’s, so that yacc and bison report 0 conflicts.
 *
 *  The ambiguities are:
 *  1) dangling else resolved to inner-most if
 *      1 in 2 states on else
 *  2) < as start-template or less-than
 *      1 conflict in 2 states on <
 *  3) a :: b :: c resolved to favour a::b::c rather than a::b ::c or a ::b::c
 *      1 conflicts in 1 state for ::
 *  4) pointer operators maximised at end of conversion id/new in preference to binary operators
 *      2 conflicts in 4 states on * and &
 *  5a) (a)@b resolved to favour binary a@b rather than cast unary (a)(@b)
 *  5b) (a)(b) resolved to favour cast rather than call
 *      8 conflicts in 1 state for the 8 prefix operators: 6 unaries and ( and [.
 *  6) enum name { resolved to enum-specifier rather than function
 *      1 conflict in 1 state on {
 *  7) class name { resolved to class-specifier rather than function
 *      1 conflict in 1 state on {
 *  8) extern “C” resolved to linkage-specification rather than declaration
 *      1 conflict in 1 state on StringLiteral
 *  9) class X : forced to go through base-clause look-ahead
 *      1 conflict in 1 state on :
 *  10) id : forced to label_statement rather than constructor_head
 *      1 conflict in 1 state on :
 *  11) access-specifier : forced to access-declaration rather than anon bit-field
 *      1 conflict in 1 state on :
 *  12) inline/ and virtual/ forced to switch rather than divide treatment
 *      1 conflict in 2 states on /
 *  13) using StringLiteral forced to include_declaration not simple_declaration
 *      1 conflict in 1 states on StringLiteral
 *  14) handler_seq maximised avoiding ambiguity in compound_tree_statement
 *      1 conflict in 1 states on catch
 *  15) built_in_type_id maximised resolving ambiguity for auto unsigned int :: a
 *      1 conflict in 1 states on BuiltInTypeSpecifier
 *  of which
 *      1 is a fundamental C conflict - always correctly resolved
 *          can be removed - see the Java spec
 *      2, 3, 4 are fundamental C++ conflicts
 *          2 always consistently resolved by iteration
 *          3 always correctly resolved
 *          4 always correctly resolved
 *      5 is a result of not using type information - deferred for semantic repair
 *      6,7 are caused by parsing over-generous superset - always correctly resolved
 *      8 is caused by parsing over-generous superset - always correctly resolved
 *          can be removed at the expense of 7 rules and 5 states.
 *      9 is a look-ahead trick - always correctly resolved
 *          could be removed by marking one token sooner
 *      10 is caused by parsing over-generous superset - always correctly resolved
 *      11 is caused by parsing over-generous superset - always correctly resolved
 *      12 is caused by parsing over-generous superset - always correctly resolved
 *      13 is caused by parsing over-generous superset - always correctly resolved
 *      14 is a genuine conflict - always correctly resolved by definition
 *          more enthusiastic parsing of the } or ; statement end could fix this
 *      15 is a fundamental FOG conflict comparable to 3
 *          always correctly resolved
 *
 *  The hard problem of distinguishing
 *      class A { class B : C, D, E {           -- A::B privately inherits C, D and E
 *      class A { class B : C, D, E ;           -- C is width of anon bit-field
 *  is resolved by using a lookahead that assumes inheritance and rewinds for the bit-field.
 *
 *  The potential shift-reduce conflict on > is resolved by flattening part of the expression grammar
 *  to know when the next > is template end or arithmetic >.
 *
 *  The grammar is SYNTACTICALLY context-free with respect to type. No semantic assistance is required
 *  during syntactic analysis. However the cast ambiguity is deferred and must be recovered
 *  after syntactic analysis of a statement has completed.
 *
 *  The grammar is SYNTACTICALLY context-free with respect to template-names. This is achieved by
 *  organising a binary search over all possible template/arithmetic ambiguities with respect to
 *  the enclosing statement. This is potentially exponentially inefficient but well-behaved in practice.
 *  Approximately 1% of statements trigger a search and approximately 1% of those are misparsed,
 *  requiring the semantic analysis to check and correct once template information is available.
 *  1.5 parse attempts are required on average per ambiguous statement.
 *
 *  The grammar supports type I function declarations at severe impediment to efficiency. A lookahead
 *  has to be performed after almost every non-statement close parenthesis. A one-line plus corollary
 *  change to postfix_expression is commented and strongly recommended to make this grammar as
 *  efficient as the rather large number of reduction levels permits.
 *
 *  Error recovery occurs mostly at the statement/declaration level. Recovery also occurs at
 *  the list-element level where this poses no hazard to statement/declaration level recovery.
 *  Note that since error propagation interacts with the lookaheads for template iteration or
 *  type 1 function arguments, introduction of finer grained error recovery may repair a false
 *  parse and so cause a misparse.
 *
 *  The following syntactic analysis errors occur, but are correctable semantically:
 *  (cast)unary-op expr         is parsed as (parenthesised)binary-op expr
 *      The semantic test should look for a binary/call with a (type) as its left child.
 *  (parenthesised)(arguments)  is parsed as (cast)(parenthesised)
 *      The semantic test should look for a cast with a non-type as its left child.
 *  template < and arithmetic < may be cross-parsed (unless semnatic help is provided)
 *      approximately 0.01% are misparsed, and must be sorted out - not easy.
 *
 *  The syntactic analysis defers the following ambiguities for semantic resolution:
 *  declaration/expression is parsed as a unified concept
 *      Use type and context to complete the parse.
 *  ~class-name                 is parsed as unary~ name
 *      The semantic test should look for ~ with a type as its child.
 *  delete[] expr               is parsed as delete []expr
 *      The semantic test should look for delete with a [] cast of its child.
 *  operator new/delete[]       are parsed as array of operator new/delete
 *      The semantic test should look for array of operator new/delete
 *      or activate the two extra commented rules in operator
 *  template of an explicit_instantiation is buried deep in the tree
 *      dig it out
 *  pure-specifier and constant-initializer are covered by assignment-expression
 *      just another of the deferred declaration/expression ambiguities
 *  sizeof and typeid don’t distinguish type/value syntaxes
 *      probably makes life polymorphically easier
 */
%nonassoc SHIFT_THERE
%nonassoc SCOPE ELSE INC DEC ‘+’ ‘-’ ‘*’ ‘&’ ‘[‘ ‘{‘ ‘<‘ ‘:’ StringLiteral
 ‘/’ CATCH BuiltInTypeSpecifier
%nonassoc REDUCE_HERE_MOSTLY
%nonassoc ‘(‘
/*%nonassoc REDUCE_HERE */

%start translation_unit
%%

/*
 *  The %prec resolves the 14.2-3 ambiguity:
 *  Identifier ‘<‘ is forced to go through the is-it-a-template-name test
 *  All names absorb TEMPLATE with the name, so that no template_test is performed for them.
 *  This requires all potential declarations within an expression to perpetuate this policy
 *  and thereby guarantee the ultimate coverage of explicit_instantiation.
 *
 *  The %prec also resolves a conflict in identifier : which is forced to be a shift of a label for
 *  a labeled-statement rather than a reduction for the name of a bit-field or generalised constructor.
 *  This is pretty dubious syntactically but correct for all semantic possibilities.
 *  The shift is only activated when the ambiguity exists at the start of a statement. In this context
 *  a bit-field declaration or constructor definition are not allowed.
 */
identifier:                         Identifier
    |                               MetaType
    |                               TreeLiteral
    |                               DERIVED | EMIT | FILE | FROZEN | GUARD | IMPLEMENTATION
    |                               INCLUDE | INTERFACE | NOGUARD | NOIMPLEMENTATION
    |                               OVERLOAD | PATH | POOL | PREFIX | PURE | SUFFIX | UTILITY
    |                               segment
id:                                 identifier                          %prec SHIFT_THERE       /* Force < through test */
    |                               identifier template_test ‘+’ template_argument_list ‘>’
    |                               identifier template_test ‘-’                                /* requeued < follows */
    |                               template_id
template_test:                      ‘<’             /* Queue ‘+’ or ‘-’ < as follow on */       { template_test(); }
global_scope:                       SCOPE
    |                               TEMPLATE global_scope
id_scope:                           id SCOPE
/*
 *  A :: B :: C; is ambiguous How much is type and how much name ?
 *  The %prec maximises the (type) length which is the 7.1-2 semantic constraint.
 */
nested_id:                          id                                  %prec SHIFT_THERE       /* Maximise length */
    |                               id_scope nested_id
scoped_id:                          nested_id
    |                               global_scope nested_id

/*
 *  destructor_id has to be held back to avoid a conflict with a one’s complement as per 5.3.1-9,
 *  It gets put back only when scoped or in a declarator_id, which is only used as an explicit member name.
 *  Declarations of an unscoped destructor are always parsed as a one’s complement.
 */
destructor_id:                      ‘~’ id
    |                               TEMPLATE destructor_id
special_function_id:                conversion_function_id
    |                               operator_function_id
    |                               TEMPLATE special_function_id
nested_special_function_id:         special_function_id
    |                               id_scope destructor_id
    |                               id_scope nested_special_function_id
scoped_special_function_id:         nested_special_function_id
    |                               global_scope nested_special_function_id

/* declarator-id is all names in all scopes, except reserved words */
declarator_id:                      scoped_id
    |                               scoped_special_function_id
    |                               destructor_id

/*  The standard defines pseudo-destructors in terms of type-name, which is class/enum/typedef, of which
 *  class-name is covered by a normal destructor. pseudo-destructors are supposed to support ~int() in
 *  templates, so the grammar here covers built-in names. Other names are covered by the lack of
 *  identifier/type discrimination.
 */
built_in_type_id:                   built_in_type_specifier
    |                               built_in_type_id built_in_type_specifier
pseudo_destructor_id:               built_in_type_id SCOPE ‘~’ built_in_type_id
    |                               ‘~’ built_in_type_id
    |                               TEMPLATE pseudo_destructor_id
nested_pseudo_destructor_id:        pseudo_destructor_id
    |                               id_scope nested_pseudo_destructor_id
scoped_pseudo_destructor_id:        nested_pseudo_destructor_id
    |                               global_scope scoped_pseudo_destructor_id

/*---------------------------------------------------------------------------------------------------
 * A.2 Lexical conventions
 *---------------------------------------------------------------------------------------------------*/
/*
 *  String concatenation is a phase 6, not phase 7 activity so does not really belong in the grammar.
 *  However it may be convenient to have it here to make this grammar fully functional.
 *  Unfortunately it introduces a conflict with the generalised parsing of extern “C” which
 *  is correctly resolved to maximise the string length as the token source should do anyway.
 */
string:                             StringLiteral
/*string:                           StringLiteral                           %prec SHIFT_THERE */
/*  |                               StringLiteral string  -- Perverse order avoids conflicts -- */
literal:                            IntegerLiteral
    |                               CharacterLiteral
    |                               FloatingLiteral
    |                               string
    |                               boolean_literal
    |                               NumberLiteral
boolean_literal:                    FALSE
    |                               TRUE

/*---------------------------------------------------------------------------------------------------
 * A.3 Basic concepts
 *---------------------------------------------------------------------------------------------------*/
translation_unit:                   declaration_seq.opt
/* expression grammar */
    |                               ‘$’ tree_expression                                    { YYACCEPT; }
    |                               ‘$’ ‘{‘ tree_expression ecarb                          { YYACCEPT; }
    |                               ‘$’ bang error                                         { UNBANG(“Bad tree-expression.”); YYABORT; }
    |                               ‘$’ ‘{‘ bang error ecarb                               { UNBANG(“Bad tree-expression}.”); YYABORT; }

/*---------------------------------------------------------------------------------------------------
 * A.4 Expressions
 *---------------------------------------------------------------------------------------------------
 *  primary_expression covers an arbitrary sequence of all names with the exception of an unscoped destructor,
 *  which is parsed as its unary expression which is the correct disambiguation (when ambiguous).
 *  This eliminates the traditional A(B) meaning A B ambiguity, since we never have to tack an A onto
 *  the front of something that might start with (. The name length got maximised ab initio. The downside
 *  is that semantic interpretation must split the names up again.
 *
 *  Unification of the declaration and expression syntax means that unary and binary pointer declarator operators:
 *      int * * name
 *  are parsed as binary and unary arithmetic operators (int) * (*name). Since type information is not used
 *  ambiguities resulting from a cast
 *      (cast)*(value)
 *  are resolved to favour the binary rather than the cast unary to ease AST clean-up.
 *  The cast-call ambiguity must be resolved to the cast to ensure that (a)(b)c can be parsed.
 *
 *  The problem of the functional cast ambiguity
 *      name(arg)
 *  as call or declaration is avoided by maximising the name within the parsing kernel. So
 *  primary_id_expression picks up
 *      extern long int const var = 5;
 *  as an assignment to the syntax parsed as “extern long int const var”. The presence of two names is
 *  parsed so that “extern long into const” is distinguished from “var” considerably simplifying subsequent
 *  semantic resolution.
 *
 *  The generalised name is a concatenation of potential type-names (scoped identifiers or built-in sequences)
 *  plus optionally one of the special names such as an operator-function-id, conversion-function-id or
 *  destructor as the final name.
 */
primary_expression:                 literal
    |                               THIS
    |                               suffix_decl_specified_ids
/*  |                               SCOPE identifier                            -- covered by suffix_decl_specified_ids */
/*  |                               SCOPE operator_function_id                  -- covered by suffix_decl_specified_ids */
/*  |                               SCOPE qualified_id                          -- covered by suffix_decl_specified_ids */
    |                               abstract_expression            %prec REDUCE_HERE_MOSTLY  /* Prefer binary to unary ops, cast to call */
/*  |                               id_expression                               -- covered by suffix_decl_specified_ids */
/*
 *  Abstract-expression covers the () and [] of abstract-declarators.
 */
abstract_expression:                parenthesis_clause
    |                               ‘[‘ expression.opt ‘]’
    |                               TEMPLATE abstract_expression

/*  Type I function parameters are ambiguous with respect to the generalised name, so we have to do a lookahead following
 *  any function-like parentheses. This unfortunately hits normal code, so kill the -- lines and add the ++ lines for efficiency.
 *  Supporting Type I code under the superset causes perhaps 25% of lookahead parsing. Sometimes complete class definitions
 *  get traversed since they are valid generalised type I parameters!
 */
type1_parameters:       /*----*/    parameter_declaration_list ‘;’
    |                   /*----*/    type1_parameters parameter_declaration_list ‘;’
mark_type1:                         /* empty */                             { mark_type1(); yyclearin; }
postfix_expression:                 primary_expression
/*  |                   /++++++/    postfix_expression parenthesis_clause */
    |                   /*----*/    postfix_expression parenthesis_clause mark_type1 ‘-’
    |                   /*----*/    postfix_expression parenthesis_clause mark_type1 ‘+’ type1_parameters mark ‘{‘ error
                        /*----*/                    { yyerrok; yyclearin; remark_type1(); unmark(); unmark(); }
    |                   /*----*/    postfix_expression parenthesis_clause mark_type1 ‘+’ type1_parameters mark error
                        /*----*/                    { yyerrok; yyclearin; remark_type1(); unmark(); unmark(); }
    |                   /*----*/    postfix_expression parenthesis_clause mark_type1 ‘+’ error
                        /*----*/                    { yyerrok; yyclearin; remark_type1(); unmark(); }
    |                               postfix_expression ‘[‘ expression.opt ‘]’
/*  |                               destructor_id ‘[‘ expression.opt ‘]’                    -- not semantically valid */
/*  |                               destructor_id parenthesis_clause                        -- omitted to resolve known ambiguity */
/*  |                               simple_type_specifier ‘(‘ expression_list.opt ‘)’       -- simple_type_specifier is a primary_expression */
    |                               postfix_expression ‘.’ declarator_id
/*  |                               postfix_expression ‘.’ TEMPLATE declarator_id           -- TEMPLATE absorbed into declarator_id. */
```

```
    |                               postfix_expression ‘.’ scoped_pseudo_destructor_id
    |                               postfix_expression ARROW declarator_id
/*  |                               postfix_expression ARROW TEMPLATE declarator_id         -- TEMPLATE absorbed into declarator_id. */
    |                               postfix_expression ARROW scoped_pseudo_destructor_id
    |                               postfix_expression INC
    |                               postfix_expression DEC
    |                               DYNAMIC_CAST ‘<‘ type_id ‘>’ ‘(‘ expression ‘)’
    |                               STATIC_CAST ‘<‘ type_id ‘>’ ‘(‘ expression ‘)’
    |                               REINTERPRET_CAST ‘<‘ type_id ‘>’ ‘(‘ expression ‘)’
    |                               CONST_CAST ‘<‘ type_id ‘>’ ‘(‘ expression ‘)’
    |                               TYPEID parameters_clause
/*  |                               TYPEID ‘(‘ expression ‘)’                               -- covered by parameters_clause */
/*  |                               TYPEID ‘(‘ type_id ‘)’                                  -- covered by parameters_clause */
expression_list.opt:                /* empty */
    |                               expression_list
expression_list:                    assignment_expression
    |                               expression_list ‘,’ assignment_expression

unary_expression:                   postfix_expression
    |                               INC cast_expression
    |                               DEC cast_expression
    |                               ptr_operator cast_expression
/*  |                               ‘*’ cast_expression                                     -- covered by ptr_operator */
/*  |                               ‘&’ cast_expression                                     -- covered by ptr_operator */
/*  |                               decl_specifier_seq ‘*’ cast_expression                  -- covered by binary operator */
/*  |                               decl_specifier_seq ‘&’ cast_expression                  -- covered by binary operator */
    |                               suffix_decl_specified_scope star_ptr_operator cast_expression   /* covers e.g int ::type::* const t = 4 */

    |                               ‘+’ cast_expression
    |                               ‘-’ cast_expression
    |                               ‘!’ cast_expression
    |                               ‘~’ cast_expression
    |                               SIZEOF unary_expression
/*  |                               SIZEOF ‘(‘ type_id ‘)’                                  -- covered by unary_expression */
    |                               new_expression
    |                               global_scope new_expression
    |                               delete_expression
    |                               global_scope delete_expression
/*  |                              DELETE ‘[‘ ‘]’ cast_expression       -- covered by DELETE cast_expression since cast_expression covers ... */
/*  |                              SCOPE DELETE ‘[‘ ‘]’ cast_expression //  ... abstract_expression cast_expression and so [] cast_expression */

delete_expression:                  DELETE cast_expression                                  /* also covers DELETE[] cast_expression */

new_expression:                     NEW new_type_id new_initializer.opt
    |                               NEW parameters_clause new_type_id new_initializer.opt
    |                               NEW parameters_clause
/*  |                               NEW ‘(‘ type-id ‘)’                                     -- covered by parameters_clause */
    |                               NEW parameters_clause parameters_clause new_initializer.opt
/*  |                               NEW ‘(‘ type-id ‘)’ new_initializer                     -- covered by parameters_clause parameters_clause */
/*  |                               NEW parameters_clause ‘(‘ type-id ‘)’                   -- covered by parameters_clause parameters_clause */
                                                                            /* ptr_operator_seq.opt production reused to save a %prec */
new_type_id:                        type_specifier ptr_operator_seq.opt
    |                               type_specifier new_declarator
    |                               type_specifier new_type_id
new_declarator:                     ptr_operator new_declarator
    |                               direct_new_declarator
```

```
direct_new_declarator:              ‘[‘ expression ‘]’
    |                               direct_new_declarator ‘[‘ constant_expression ‘]’
new_initializer.opt:                /* empty */
    |                               ‘(‘ expression_list.opt ‘)’

/*  cast-expression is generalised to support a [] as well as a () prefix. This covers the omission of DELETE[] which when
 *  followed by a parenthesised expression was ambiguous. It also covers the gcc indexed array initialisation for free.
 */
cast_expression:                    unary_expression
    |                               abstract_expression cast_expression
/*  |                               ‘(‘ type_id ‘)’ cast_expression                             -- covered by abstract_expression */

pm_expression:                      cast_expression
    |                               pm_expression DOT_STAR cast_expression
    |                               pm_expression ARROW_STAR cast_expression
multiplicative_expression:          pm_expression
    |                               multiplicative_expression star_ptr_operator pm_expression
    |                               multiplicative_expression ‘/’ pm_expression
    |                               multiplicative_expression ‘%’ pm_expression
additive_expression:                multiplicative_expression
    |                               additive_expression ‘+’ multiplicative_expression
    |                               additive_expression ‘-’ multiplicative_expression
shift_expression:                   additive_expression
    |                               shift_expression SHL additive_expression
    |                               shift_expression SHR additive_expression
relational_expression:              shift_expression
    |                               relational_expression ‘<‘ shift_expression
    |                               relational_expression ‘>’ shift_expression
    |                               relational_expression LE shift_expression
    |                               relational_expression GE shift_expression
equality_expression:                relational_expression
    |                               equality_expression EQ relational_expression
    |                               equality_expression NE relational_expression
and_expression:                     equality_expression
    |                               and_expression ‘&’ equality_expression
exclusive_or_expression:            and_expression
    |                               exclusive_or_expression ‘^’ and_expression
inclusive_or_expression:            exclusive_or_expression
    |                               inclusive_or_expression ‘|’ exclusive_or_expression
logical_and_expression:             inclusive_or_expression
    |                               logical_and_expression LOG_AND inclusive_or_expression
logical_or_expression:              logical_and_expression
    |                               logical_or_expression LOG_OR logical_and_expression
conditional_expression:             logical_or_expression
    |                               logical_or_expression ‘?’ expression ‘:’ assignment_expression


/*  assignment-expression is generalised to cover the simple assignment of a braced initializer in order to contribute to the
 *  coverage of parameter-declaration and init-declaration.
 */
assignment_expression:              conditional_expression
    |                               logical_or_expression assignment_operator assignment_expression
    |                               logical_or_expression ‘=’ braced_initializer
    |                               throw_expression
assignment_operator:                ‘=’ | ASS_ADD | ASS_AND | ASS_DIV | ASS_MOD | ASS_MUL | ASS_OR | ASS_SHL | ASS_SHR | ASS_SUB | ASS_XOR
```

```
/*  expression is widely used and usually single-element, so the reductions are arranged so that a
 *  single-element expression is returned as is. Multi-element expressions are parsed as a list that
 *  may then behave polymorphically as an element or be compacted to an element. */
expression.opt:                     /* empty */
    |                               expression
expression:                         assignment_expression
    |                               expression_list ‘,’ assignment_expression
constant_expression:                conditional_expression

/*  The grammar is repeated for when the parser stack knows that the next > must end a template.
 */
templated_relational_expression:    shift_expression
    |                               templated_relational_expression ‘<‘ shift_expression
    |                               templated_relational_expression LE shift_expression
    |                               templated_relational_expression GE shift_expression
templated_equality_expression:      templated_relational_expression
    |                               templated_equality_expression EQ templated_relational_expression
    |                               templated_equality_expression NE templated_relational_expression
templated_and_expression:           templated_equality_expression
    |                               templated_and_expression ‘&’ templated_equality_expression
templated_exclusive_or_expression:  templated_and_expression
    |                               templated_exclusive_or_expression ‘^’ templated_and_expression

templated_inclusive_or_expression:  templated_exclusive_or_expression
    |                               templated_inclusive_or_expression ‘|’ templated_exclusive_or_expression

templated_logical_and_expression:   templated_inclusive_or_expression
    |                               templated_logical_and_expression LOG_AND templated_inclusive_or_expression

templated_logical_or_expression:    templated_logical_and_expression
    |                               templated_logical_or_expression LOG_OR templated_logical_and_expression

templated_conditional_expression:   templated_logical_or_expression
    |                               templated_logical_or_expression ‘?’ templated_expression ‘:’ templated_assignment_expression

templated_assignment_expression:    templated_conditional_expression
    |                               templated_logical_or_expression assignment_operator templated_assignment_expression

    |                               templated_throw_expression
templated_expression:               templated_assignment_expression
    |                               templated_expression_list ‘,’ templated_assignment_expression

templated_expression_list:          templated_assignment_expression
    |                               templated_expression_list ‘,’ templated_assignment_expression

/*---------------------------------------------------------------------------------------------------
 * A.5 Statements
 *---------------------------------------------------------------------------------------------------
 *  Parsing statements is easy once simple_declaration has been generalised to cover expression_statement.
 */
looping_statement:                  start_search looped_statement                               { end_search(); }
looped_statement:                   statement
    |                               advance_search ‘+’ looped_statement
    |                               advance_search ‘-’
statement:                          control_statement
/*  |                               expression_statement                                        -- covered by declaration_statement */
```

```
    |                               compound_statement
    |                               declaration_statement
    |                               try_block
    |                               AUTO control_statement
    |                               AUTO meta_expression_statement
control_statement:                  labeled_statement
    |                               selection_statement
    |                               iteration_statement
    |                               jump_statement
labeled_statement:                  identifier ‘:’ looping_statement
    |                               CASE constant_expression ‘:’ looping_statement
    |                               DEFAULT ‘:’ looping_statement
/*expression_statement:             expression.opt ‘;’                                          -- covered by declaration_statement */
compound_statement:                 ‘{‘ statement_seq.opt ‘}’
    |                               ‘{‘ statement_seq.opt looping_statement ‘#’ bang error ‘}’  { UNBANG(“Bad statement-seq.”); }
statement_seq.opt:                  /* empty */
    |                               statement_seq.opt looping_statement
    |                               statement_seq.opt looping_statement ‘#’ bang error ‘;’      { UNBANG(“Bad statement.”); }
/*
 *  The dangling else conflict is resolved to the innermost if.
 */
selection_statement:                IF ‘(‘ condition ‘)’ looping_statement    %prec SHIFT_THERE
    |                               IF ‘(‘ condition ‘)’ looping_statement ELSE looping_statement
    |                               SWITCH ‘(‘ condition ‘)’ looping_statement
condition.opt:                      /* empty */
    |                               condition
condition:                          parameter_declaration_list
/*  |                               expression                                                  -- covered by parameter_declaration_list */
/*  |                               type_specifier_seq declarator ‘=’ assignment_expression     -- covered by parameter_declaration_list */
iteration_statement:                WHILE ‘(‘ condition ‘)’ looping_statement
    |                               DO looping_statement WHILE ‘(‘ expression ‘)’ ‘;’
    |                               FOR ‘(‘ for_init_statement condition.opt ‘;’ expression.opt ‘)’ looping_statement

for_init_statement:                 simple_declaration
/*  |                               expression_statement                                        -- covered by simple_declaration */
jump_statement:                     BREAK ‘;’
    |                               CONTINUE ‘;’
    |                               RETURN expression.opt ‘;’
    |                               GOTO identifier ‘;’
declaration_statement:              block_declaration

/*----------------------------------------------------------------------------------------------------
 * A.6 Declarations
 *----------------------------------------------------------------------------------------------------*/
compound_declaration:               ‘{‘ nest declaration_seq.opt ‘}’                            { unnest(); }
    |                               ‘{‘ nest declaration_seq.opt util looping_declaration ‘#’ bang error ‘}’
                                                                                                { unnest(); UNBANG(“Bad declaration-seq.”); }
declaration_seq.opt:                /* empty */
    |                               declaration_seq.opt util looping_declaration
    |                               declaration_seq.opt util looping_declaration ‘#’ bang error ‘;’ { UNBANG(“Bad declaration.”); }
looping_declaration:                start_search1 looped_declaration                            { end_search(); }
looped_declaration:                 declaration
    |                               advance_search ‘+’ looped_declaration
    |                               advance_search ‘-’
declaration:                        block_declaration
    |                               function_definition
    |                                template_declaration
/*  |                                explicit_instantiation                                        -- covered by relevant declarations */
    |                                explicit_specialization
    |                                specialised_declaration
    |                                accessibility_specifier
    |                                compound_declaration
    |                                meta_control_statement
    |                                AUTO meta_control_statement
    |                                AUTO meta_class_specifier semi
    |                                AUTO meta_expression_statement
    |                                AUTO meta_function_definition
    |                                syntax_macro_definition
    |                                include_declaration semi
    |                                file_dependency_declaration
    |                                file_placement_declaration
    |                                filespace_specifier semi
specialised_declaration:             linkage_specification
    |                                namespace_declaration
    |                                namespace_definition
    |                                TEMPLATE specialised_declaration
block_declaration:                   simple_declaration
    |                                specialised_block_declaration
specialised_block_declaration:       asm_definition
    |                                namespace_alias_definition
/*  |                                using_declaration                                             -- covered by simple_declaration */
    |                                using_directive
    |                                TEMPLATE specialised_block_declaration
simple_declaration:                  ‘;’
    |                                init_declaration ‘;’
    |                                constructor_head ‘,’ assignment_expression ‘;’
    |                                init_declarations ‘;’
    |                                decl_specifier_prefix simple_declaration

/*  A decl-specifier following a ptr_operator provokes a shift-reduce conflict for
 *      * const name
 *  which is resolved in favour of the pointer, and implemented by providing versions
 *  of decl-specifier guaranteed not to start with a cv_qualifier.
 *
 *  decl-specifiers are implemented type-centrically. That is the semantic constraint
 *  that there must be a type is exploited to impose structure, but actually eliminate
 *  very little syntax. built-in types are multi-name and so need a different policy.
 *
 *  non-type decl-specifiers are bound to the left-most type in a decl-specifier-seq,
 *  by parsing from the right and attaching suffixes to the right-hand type. Finally
 *  residual prefixes attach to the left.
 */
suffix_built_in_decl_specifier.raw: built_in_type_specifier
    |                                suffix_built_in_decl_specifier.raw built_in_type_specifier
    |                                suffix_built_in_decl_specifier.raw decl_specifier_suffix
suffix_built_in_decl_specifier:      suffix_built_in_decl_specifier.raw
    |                                TEMPLATE suffix_built_in_decl_specifier
suffix_named_decl_specifier:         scoped_id
    |                                elaborate_type_specifier
    |                                suffix_named_decl_specifier decl_specifier_suffix
suffix_named_decl_specifier.bi:      suffix_named_decl_specifier
    |                                suffix_named_decl_specifier suffix_built_in_decl_specifier.raw
```

```
suffix_named_decl_specifiers:       suffix_named_decl_specifier.bi
    |                               suffix_named_decl_specifiers suffix_named_decl_specifier.bi
suffix_named_decl_specifiers.sf:    scoped_special_function_id          /* operators etc */
    |                               suffix_named_decl_specifiers
    |                               suffix_named_decl_specifiers scoped_special_function_id
suffix_decl_specified_ids:          suffix_built_in_decl_specifier
    |                               suffix_built_in_decl_specifier suffix_named_decl_specifiers.sf
    |                               suffix_named_decl_specifiers.sf
suffix_decl_specified_scope:        suffix_named_decl_specifiers SCOPE
    |                               suffix_built_in_decl_specifier suffix_named_decl_specifiers SCOPE
    |                               suffix_built_in_decl_specifier SCOPE

decl_specifier_affix:               storage_class_specifier
    |                               function_specifier
    |                               FRIEND
    |                               TYPEDEF
    |                               cv_qualifier
/*  The bogus conflict between public: as an anonymous bit-field and member-specification is resolved to the member-specification.*/
    |                               access_specifier            %prec SHIFT_THERE
/*  using-declaration is generalised to cover a much more general concept of re-use, so using treated like typedef.
 *  Unfortunately this gives the same conflict on string as for linkage_specification, so the %prec forces using followed
 *  by a string to be treated as an include rather than a declaration. */
    |                               USING                       %prec SHIFT_THERE

decl_specifier_suffix:              decl_specifier_affix
    |                               AUTO

decl_specifier_prefix:              decl_specifier_affix
    |                               TEMPLATE decl_specifier_prefix

storage_class_specifier:            REGISTER | STATIC | MUTABLE
    |                               EXTERN                  %prec SHIFT_THERE                   /* Prefer linkage specification */
    |                               '!' STATIC

function_specifier:                 EXPLICIT
    |                               INLINE                  %prec SHIFT_THERE                   /* Prefer INLINE / IMPLEMENTATION */
    |                               VIRTUAL                 %prec SHIFT_THERE                   /* Prefer VIRTUAL / PURE */
    |                               '!' INLINE
    |                               INLINE '/' IMPLEMENTATION
    |                               INLINE '/' INTERFACE
    |                               '!' VIRTUAL
    |                               VIRTUAL '/' PURE

type_specifier:                     simple_type_specifier
    |                               elaborate_type_specifier
    |                               cv_qualifier
/* The following augment type_specifier rather than cv_qualifier to avoid a conflict on ! between
 *      a * ! const b     and    a * ! b    which requires a 2-token lookahead to resolve. */
    |                               '!' CONST
    |                               '!' VOLATILE

elaborate_type_specifier:           class_specifier
    |                               enum_specifier
    |                               elaborated_type_specifier
    |                               TEMPLATE elaborate_type_specifier
simple_type_specifier:              scoped_id
```

```
    |                                   built_in_type_specifier
built_in_type_specifier:            BuiltInTypeSpecifier

/*
 *  The over-general use of declaration_expression to cover decl-specifier-seq.opt declarator in a function-definition means that
 *      class X { };
 *  could be a function-definition or a class-specifier.
 *      enum X { };
 *  could be a function-definition or an enum-specifier.
 *  The function-definition is not syntactically valid so resolving the false conflict in favour of the
 *  elaborated_type_specifier is correct.
 */
elaborated_type_specifier:          elaborated_class_specifier
    |                               elaborated_enum_specifier
    |                               TYPENAME scoped_id

elaborated_enum_specifier:          ENUM scoped_id              %prec SHIFT_THERE
enum_specifier:                     ENUM scoped_id enumerator_clause
    |                               ENUM enumerator_clause
enumerator_clause:                  '{' enumerator_list_ecarb
    |                               '{' enumerator_list enumerator_list_ecarb
    |                               '{' enumerator_list ',' enumerator_definition_ecarb
enumerator_list_ecarb:              '}'
    |                               bang error '}'                                          { UNBANG("Bad enumerator-list."); }
enumerator_definition_ecarb:        '}'
    |                               bang error '}'                                          { UNBANG("Bad enumerator-definition."); }
enumerator_definition_filler:       /* empty */
    |                               bang error ','                                          { UNBANG("Bad enumerator-definition."); }
enumerator_list_head:               enumerator_definition_filler
    |                               enumerator_list ',' enumerator_definition_filler
enumerator_list:                    enumerator_list_head enumerator_definition
enumerator_definition:              enumerator
    |                               enumerator '=' constant_expression
enumerator:                         identifier

namespace_definition:               NAMESPACE scoped_id compound_declaration
    |                               NAMESPACE compound_declaration
namespace_alias_definition:         NAMESPACE scoped_id '=' scoped_id ';'
namespace_declaration:              NAMESPACE scoped_id ';'


using_directive:                    USING NAMESPACE scoped_id ';'
asm_definition:                     ASM '(' string ')' ';'
linkage_specification:              EXTERN string looping_declaration
/*   |                              EXTERN string compound_declaration                      -- covered by declaration */

/*---------------------------------------------------------------------------------------------------
 * A.7 Declarators
 *---------------------------------------------------------------------------------------------------*/
/*init-declarator is named init_declaration to reflect the embedded decl-specifier-seq.opt*/
init_declarations:                  assignment_expression ',' init_declaration
    |                               init_declarations ',' init_declaration
    |                               init_object_declaration ',' init_declaration
    |                               constructor_head ',' bit_field_init_declaration
    |                               constructor_head ',' init_object_declaration
init_declaration:                   assignment_expression
```

```
/*  |                             assignment_expression ‘=’ initializer_clause               -- covered by assignment_expression */
/*  |                             assignment_expression ‘(‘ expression_list ‘)’              -- covered by another set of call arguments */
    |                             bit_field_init_declaration
    |                             init_object_declaration
init_object_declaration:          assignment_expression object_statements_clause
    |                             bit_field_init_declaration object_statements_clause

/*declarator:                                                                                -- covered by assignment_expression */
/*direct_declarator:                                                                         -- covered by postfix_expression */

star_ptr_operator:                ‘*’
    |                             star_ptr_operator cv_qualifier
nested_ptr_operator:              star_ptr_operator
    |                             id_scope nested_ptr_operator
ptr_operator:                     ‘&’
    |                             nested_ptr_operator
    |                             global_scope nested_ptr_operator
ptr_operator_seq:                 ptr_operator
    |                             ptr_operator ptr_operator_seq
/* Independently coded to localise the shift-reduce conflict: sharing just needs another %prec */
ptr_operator_seq.opt:             /* empty */                          %prec SHIFT_THERE       /* Maximise type length */
    |                             ptr_operator ptr_operator_seq.opt

cv_qualifier_seq.opt:             /* empty */
    |                             cv_qualifier_seq.opt cv_qualifier
cv_qualifier:                     CONST | VOLATILE

/*type_id                                                                                    -- also covered by parameter declaration */
type_id:                          type_specifier abstract_declarator.opt
    |                             type_specifier type_id

/*abstract_declarator:                                                                       -- also covered by parameter declaration */
abstract_declarator.opt:          /* empty */
    |                             ptr_operator abstract_declarator.opt
    |                             direct_abstract_declarator
direct_abstract_declarator.opt:   /* empty */
    |                             direct_abstract_declarator
direct_abstract_declarator:       direct_abstract_declarator.opt parenthesis_clause
    |                             direct_abstract_declarator.opt ‘[‘ ‘]’
    |                             direct_abstract_declarator.opt ‘[‘ constant_expression ‘]’
/*  |                             ‘(‘ abstract_declarator ‘)’                                -- covered by parenthesis_clause */

parenthesis_clause:               parameters_clause cv_qualifier_seq.opt
    |                             parameters_clause cv_qualifier_seq.opt exception_specification
parameters_clause:                ‘(‘ parameter_declaration_clause ‘)’
/* parameter_declaration_clause also covers init_declaration, type_id, declarator and abstract_declarator. */
parameter_declaration_clause:     /* empty */
    |                             parameter_declaration_list
    |                             parameter_declaration_list ELLIPSIS
parameter_declaration_list:       parameter_declaration
    |                             parameter_declaration_list ‘,’ parameter_declaration

/*
 * A typed abstract qualifier such as
 *      Class * ...
 * looks like a multiply, so pointers are parsed as their binary operation equivalents that
 * ultimately terminate with a degenerate right hand term.
 */
abstract_pointer_declaration:       ptr_operator_seq
    |                               multiplicative_expression star_ptr_operator ptr_operator_seq.opt
abstract_parameter_declaration:     abstract_pointer_declaration
    |                               and_expression ‘&’
    |                               and_expression ‘&’ abstract_pointer_declaration
special_parameter_declaration:      abstract_parameter_declaration
    |                               abstract_parameter_declaration ‘=’ assignment_expression
    |                               ELLIPSIS
parameter_declaration:              assignment_expression
    |                               special_parameter_declaration
    |                               decl_specifier_prefix parameter_declaration

/*  The grammar is repeated for use within template <>
 */
templated_parameter_declaration:    templated_assignment_expression
    |                               templated_abstract_declaration
    |                               templated_abstract_declaration ‘=’ templated_assignment_expression

    |                               decl_specifier_prefix templated_parameter_declaration
templated_abstract_declaration:     abstract_pointer_declaration
    |                               templated_and_expression ‘&’
    |                               templated_and_expression ‘&’ abstract_pointer_declaration

/*  function_definition includes constructor, destructor, implicit int definitions too.
 *  A local destructor is successfully parsed as a function-declaration but the ~ was treated as a unary operator.
 *  constructor_head is the prefix ambiguity between a constructor and a member-init-list starting with a bit-field.
 */
function_definition:                ctor_definition
    |                               func_definition
func_definition:                    assignment_expression function_try_block
    |                               assignment_expression function_body
    |                               decl_specifier_prefix func_definition
ctor_definition:                    constructor_head function_try_block
    |                               constructor_head function_body
    |                               decl_specifier_prefix ctor_definition
constructor_head:                   bit_field_init_declaration
    |                               constructor_head ‘,’ assignment_expression
function_try_block:                 TRY function_block handler_seq
function_block:                     ctor_initializer.opt function_body
function_body:                      compound_statement

/*
 *  An = initializer looks like an extended assignment_expression.
 *  An () initializer looks like a function call.
 *  initializer is therefore flattened into its generalised customers.
 *initializer:                      ‘=’ initializer_clause                                  -- flattened into caller
 *  |                               ‘(‘ expression_list ‘)’                                 -- flattened into caller */
initializer_clause:                 assignment_expression
    |                               braced_initializer
braced_initializer:                 ‘{‘ initializer_list ‘}’
    |                               ‘{‘ initializer_list ‘,’ ‘}’
    |                               ‘{‘ ‘}’
    |                               ‘{‘ looping_initializer_clause ‘#’ bang error ‘}’           { UNBANG(“Bad initializer_clause.”); }
    |                               ‘{‘ initializer_list ‘,’ looping_initializer_clause ‘#’ bang error ‘}’
```

```
                                                                                                  { UNBANG("Bad initializer_clause."); }
initializer_list:                    looping_initializer_clause
    |                                initializer_list ',' looping_initializer_clause
looping_initializer_clause:          start_search looped_initializer_clause                      { end_search(); }
looped_initializer_clause:           initializer_clause
    |                                advance_search '+' looped_initializer_clause
    |                                advance_search '-'

/*---------------------------------------------------------------------------------------------------
 * A.8 Classes
 *---------------------------------------------------------------------------------------------------
 *
 *  An anonymous bit-field declaration may look very like inheritance:
 *      const int B = 3;
 *      class A : B ;
 *  The two usages are too distant to try to create and enforce a common prefix so we have to resort to
 *  a parser hack by backtracking. Inheritance is much the most likely so we mark the input stream context
 *  and try to parse a base-clause. If we successfully reach a { the base-clause is ok and inheritance was
 *  the correct choice so we unmark and continue. If we fail to find the { an error token causes back-tracking
 *  to the alternative parse in elaborated_type_specifier which regenerates the : and declares unconditional success.
 */
colon_mark:                          ':'                                                          { mark(); }
elaborated_class_specifier:          class_key scoped_id                   %prec SHIFT_THERE
    |                                class_key scoped_id colon_mark error                         { rewind_colon(); }
class_specifier_head:                class_key scoped_id colon_mark base_specifier_list '{'      { unmark(); }
    |                                class_key ':' base_specifier_list '{'
    |                                class_key scoped_id '{'
    |                                class_key '{'
class_key:                           CLASS | STRUCT | UNION
class_specifier:                     class_specifier_head nest declaration_seq.opt '}'           { unnest(); }
    |                                class_specifier_head nest declaration_seq.opt util looping_declaration '#' bang error '}'
                                             { unnest(); UNBANG("Bad member_specification.opt."); }
accessibility_specifier:             access_specifier ':'
bit_field_declaration:               assignment_expression ':' bit_field_width
    |                                ':' bit_field_width
bit_field_width:                     logical_or_expression
/*  |                                logical_or_expression '?' expression ':' assignment_expression  -- has SR conflict w.r.t later = */
    |                                logical_or_expression '?' bit_field_width ':' bit_field_width
bit_field_init_declaration:          bit_field_declaration
    |                                bit_field_declaration '=' initializer_clause

/*---------------------------------------------------------------------------------------------------
 * A.9 Derived classes
 *---------------------------------------------------------------------------------------------------*/
/*base_clause:                       ':' base_specifier_list                                      -- flattened */
base_specifier_list:                 base_specifier
    |                                base_specifier_list ',' base_specifier
base_specifier:                      scoped_id
    |                                access_specifier base_specifier
    |                                VIRTUAL base_specifier
    |                                '!' VIRTUAL base_specifier
    |                                AUTO base_specifier
    |                                built_in_type_id
access_specifier:                    PRIVATE | PROTECTED | PUBLIC

/*---------------------------------------------------------------------------------------------------
```

```
 * A.10 Special member functions
 *---------------------------------------------------------------------------------------------------*/
conversion_function_id:            OPERATOR conversion_type_id
conversion_type_id:                type_specifier ptr_operator_seq.opt
    |                              type_specifier conversion_type_id
/*
 *  Ctor-initialisers can look like a bit field declaration, given the generalisation of names:
 *      Class(Type) : m1(1), m2(2) { }
 *      NonClass(bit_field) : int(2), second_variable, ...
 *  The grammar below is used within a function_try_block or function_definition.
 *  See simple_member_declaration for use in normal member function_definition.
 */
ctor_initializer.opt:              /* empty */
    |                              ctor_initializer
ctor_initializer:                  ‘:’ mem_initializer_list
    |                              ‘:’ mem_initializer_list bang error                          { UNBANG(“Bad ctor-initializer.”); }
mem_initializer_list:              mem_initializer
    |                              mem_initializer_list_head mem_initializer
mem_initializer_list_head:         mem_initializer_list ‘,’
    |                              mem_initializer_list bang error ‘,’                          { UNBANG(“Bad mem-initializer.”); }
mem_initializer:                   mem_initializer_id ‘(‘ expression_list.opt ‘)’
mem_initializer_id:                scoped_id

/*---------------------------------------------------------------------------------------------------
 * A.11 Overloading
 *---------------------------------------------------------------------------------------------------*/
operator_function_id:              OPERATOR operator
/*
 *  It is not clear from the ANSI standard whether spaces are permitted in delete[]. If not then it can
 *  be recognised and returned as DELETE_ARRAY by the lexer. Assuming spaces are permitted there is an
 *  ambiguity created by the over generalised nature of expressions. operator new is a valid delarator-id
 *  which we may have an undimensioned array of. Semantic rubbish, but syntactically valid. Since the
 *  array form is covered by the declarator consideration we can exclude the operator here. The need
 *  for a semantic rescue can be eliminated at the expense of a couple of shift-reduce conflicts by
 *  removing the comments on the next four lines.
 */
operator:              /*++++*/    NEW
    |                  /*++++*/    DELETE
/*  |                  / ---- /    NEW                 %prec SHIFT_THERE
/*  |                  / ---- /    DELETE              %prec SHIFT_THERE
/*  |                  / ---- /    NEW ‘[‘ ‘]’                                                  -- Covered by array of OPERATOR NEW */
/*  |                  / ---- /    DELETE ‘[‘ ‘]’                                               -- Covered by array of OPERATOR DELETE */
    |                              ‘+’
    |                              ‘-’
    |                              ‘*’
    |                              ‘/’
    |                              ‘%’
    |                              ‘^’
    |                              ‘&’
    |                              ‘|’
    |                              ‘~’
    |                              ‘!’
    |                              ‘=’
    |                              ‘<‘
    |                              ‘>’
    |                              ASS_ADD
    |                                   ASS_SUB
    |                                   ASS_MUL
    |                                   ASS_DIV
    |                                   ASS_MOD
    |                                   ASS_XOR
    |                                   ASS_AND
    |                                   ASS_OR
    |                                   SHL
    |                                   SHR
    |                                   ASS_SHR
    |                                   ASS_SHL
    |                                   EQ
    |                                   NE
    |                                   LE
    |                                   GE
    |                                   LOG_AND
    |                                   LOG_OR
    |                                   INC
    |                                   DEC
    |                                   ‘,’
    |                                   ARROW_STAR
    |                                   ARROW
    |                                   ‘(‘ ‘)’
    |                                   ‘[‘ ‘]’

/*----------------------------------------------------------------------------------------------------
 * A.12 Templates
 *----------------------------------------------------------------------------------------------------*/
template_declaration:               template_parameter_clause declaration
    |                               EXPORT template_declaration
/*  This extension is only defined for USING, but we need to use decl_specifier_prefix to avoid conflicts. */
    |                               decl_specifier_prefix template_declaration
template_parameter_clause:          TEMPLATE ‘<‘ template_parameter_list ‘>’
template_parameter_list:            template_parameter
    |                               template_parameter_list ‘,’ template_parameter
template_parameter:                 simple_type_parameter
    |                               simple_type_parameter ‘=’ type_id
    |                               templated_type_parameter
    |                               templated_type_parameter ‘=’ identifier
    |                               templated_parameter_declaration
    |                               bang error                                            { UNBANG(“Bad template-parameter.”); }
simple_type_parameter:              CLASS
/*  |                               CLASS identifier                                      -- covered by parameter_declaration */
    |                               TYPENAME
/*  |                               TYPENAME identifier                                   -- covered by parameter_declaration */
templated_type_parameter:           template_parameter_clause CLASS
    |                               template_parameter_clause CLASS identifier
template_id:                        TEMPLATE identifier ‘<‘ template_argument_list ‘>’
    |                               TEMPLATE template_id
/*
 *  template-argument is evaluated using a templated...expression so that > resolves to end of template.
 */
template_argument_list:             template_argument
    |                               template_argument_list ‘,’ template_argument
template_argument:                  templated_parameter_declaration
/*  |                               type_id                                               -- covered by templated_parameter_declaration */
/*  |                               template_name                                                   -- covered by templated_parameter_declaration */
/*  |                               error                                                           -- must allow template failure to re-search */

/*
 *  Generalised naming makes identifier a valid declaration, so TEMPLATE identifier is too.
 *  The TEMPLATE prefix is therefore folded into all names, parenthesis_clause and decl_specifier_prefix.
 */
/*explicit_instantiation:           TEMPLATE declaration */
explicit_specialization:            TEMPLATE ‘<‘ ‘>’ declaration
/*  This extension is only defined for USING, but we need to use decl_specifier_prefix to avoid conflicts. */
    |                               decl_specifier_prefix explicit_specialization

/*---------------------------------------------------------------------------------------------------
 * A.13 Exception Handling
 *---------------------------------------------------------------------------------------------------*/
try_block:                          TRY compound_statement handler_seq
/*function_try_block:                                                                               -- moved near function_block */
/* A handler_seq may follow a try_block in a compound_tree_statement such as:
 *      if (a) try { } catch(a) {} catch(b) {} catch(c) {} ...
 *  we resolve the conflict by maximising the handler sequence. */
handler_seq:                        handler                                 %prec SHIFT_THERE   /* Maximise length */
    |                               handler handler_seq
handler:                            CATCH ‘(‘ exception_declaration ‘)’ compound_statement
exception_declaration:              parameter_declaration
/*                                  ELLIPSIS                                                        -- covered by parameter_declaration */
throw_expression:                   THROW
    |                               THROW assignment_expression
templated_throw_expression:         THROW
    |                               THROW templated_assignment_expression
exception_specification:            THROW ‘(‘ ‘)’
    |                               THROW ‘(‘ type_id_list ‘)’
type_id_list:                       type_id
    |                               type_id_list ‘,’ type_id


/*---------------------------------------------------------------------------------------------------
 * A.14 Tree literals
 *---------------------------------------------------------------------------------------------------*/
primary_tree_expression:            meta_scoped_id
    |                               ‘(‘ tree_expression ‘)’
postfix_tree_expression:            primary_tree_expression
    |                               postfix_tree_expression ‘[‘ ‘]’
    |                               postfix_tree_expression ‘[‘ constant_expression ‘]’
    |                               postfix_tree_expression ‘(‘ tree_argument_list.opt ‘)’
    |                               postfix_tree_expression ‘.’ scoped_id
    |                               postfix_tree_expression ARROW scoped_id
tree_expression:                    postfix_tree_expression
    |                               ‘*’ tree_expression

/* tree_argument_list.opt are carefully coded to avoid conflicts between the components of a constructor_head at the start of a function_definition
 * and the equivalent discrete elements. There is no need to resolve a conflict on “,”!, which is fortunate because it couldn’t work. */
tree_argument_list.opt:             tree_arguments.head
    |                               tree_arguments.head ‘,’ tree_argument_list.opt
    |                               tree_argument.ctors
tree_argument.ctors:                constructor_head
    |                               decl_specifier_prefix tree_argument.ctors
```

```
tree_arguments.head:                /* empty */
    |                               tree_argument.most
    |                               assignment_expression
    |                               func_definition
    |                               tree_argument.ctors_comma_most
tree_argument.ctors_comma_most:     constructor_head ‘,’ tree_argument.most
    |                               decl_specifier_prefix tree_argument.ctors_comma_most
tree_argument.most:                 terminated_tree_argument
    |                               ctor_definition
    |                               unterminated_tree_argument.most
    |                               unterminated_tree_argument.most ‘;’
    |                               tree_argument.misc
    |                               tree_argument.misc ‘;’
tree_argument.misc:                 decl_specifier_prefix
/*  |                               assignment_expression                                       -- separated out */
/*  |                               bit_field_init_declaration                                  -- separated out into tree_argument.ctors */
/*  |                               function_definition                                         -- split into ctor/func_definition */
    |                               init_object_declaration
    |                               special_parameter_declaration
    |                               decl_specifier_prefix assignment_expression
    |                               decl_specifier_prefix tree_argument.misc

looping_tree_statement:             start_search looped_tree_statement                          { end_search(); }
looped_tree_statement:              tree_statement
    |                               advance_search ‘+’ looped_tree_statement
    |                               advance_search ‘-’
tree_statement:                     terminated_tree_argument
    |                               unterminated_tree_argument ‘;’
compound_tree_statement:            ‘{‘ tree_statement_seq.opt ‘}’
    |                               ‘{‘ tree_statement_seq.opt looping_tree_statement ‘#’ bang error ‘}’
                                                                                { UNBANG(“Bad compound-tree-statement.”);}
tree_statement_seq.opt:             /* empty */
    |                               tree_statement_seq.opt looping_tree_statement
    |                               tree_statement_seq.opt looping_tree_statement ‘#’ bang error ‘;’
                                                                                { UNBANG(“Bad tree-statement.”); }

/*  Terminated syntax has an unambiguous end and does not need a ; as a meta-variable initializer. */
terminated_tree_argument:           asm_definition
/*  |                               declaration_statement ‘;’                   -- covered by simple_tree_declaration ; */
/*  |                               explicit_instantiation                      -- covered by simple_tree_declaration ; */
    |                               explicit_specialization
/*  |                               expression_statement ‘;’                    -- covered by simple_tree_declaration ; */
    |                               file_dependency_declaration
    |                               file_placement_declaration
    |                               include_declaration semi
    |                               iteration_statement
/*  |                               jump_statement                              -- covered by BREAK ; */
    |                               labeled_statement
    |                               linkage_specification
    |                               namespace_alias_definition
    |                               namespace_declaration
/*  |                               parameter_declaration ‘;’                   -- covered by simple_tree_declaration ; */
    |                               selection_statement
    |                               template_declaration
    |                               using_directive
    |                               AUTO meta_control_statement
    |                               AUTO meta_expression_statement
    |                               OPERATOR ';'

/*  Unterminated syntax has no obvious end and/or must have a ; as a meta-variable initializer. */
unterminated_tree_argument:         unterminated_tree_argument.most
    |                               simple_tree_declaration
    |                               function_definition
unterminated_tree_argument.most:    accessibility_specifier
/*  |                               access_specifier                           -- covered by decl_specifier_affix */
/*  |                               base_specifier                             -- covered by simple_tree_declaration */
/*  |                               built_in_type_specifier                    -- covered by simple_tree_declaration */
/*  |                               class_specifier                            -- covered by simple_tree_declaration */
                                    compound_tree_statement
/*  |                               condition                                  -- covered by simple_tree_declaration */
/*  |                               cv_qualifier                               -- covered by simple_tree_declaration */
/*  |                               decl_specifier                             -- covered by simple_tree_declaration */
/*  |                               enum_specifier                             -- covered by simple_tree_declaration */
/*  |                               enumerator_definition                      -- covered by simple_tree_declaration */
/*  |                               exception_declaration                      -- covered by simple_tree_declaration */
/*  |                               exception_specification                    -- covered by simple_tree_declaration */
    |                               filespace_specifier
/*  |                               function_definition                        -- not part of .most */
    |                               function_try_block
    |                               handler_seq
/*  |                               initializer_clause                         -- covered by simple_tree_declaration, compound_statement */
/*  |                               mem_initializer                            -- covered by simple_tree_declaration */
    |                               AUTO meta_class_specifier
    |                               AUTO meta_function_definition
    |                               namespace_definition
/*  |                               operator                                   -- mostly covered by token.punct */
/*  |                               parameter_declaration                      -- not part of .most */
/*  |                               simple_tree_declaration                    -- not part of .most */
    |                               simple_type_parameter
/*  |                               storage_class_specifier                    -- covered by simple_tree_declaration */
/*  |                               template_argument                          -- covered by simple_tree_declaration */
/*  |                               template_parameter                         -- covered by simple_tree_declaration */
/*  |                               try_block                                  -- covered by function_try_block */
/*  |                               type_id                                    -- covered by simple_tree_declaration */
/*  |                               type_parameter                             -- covered by simple_tree_declaration, template_declaration */
    |                               reserved_id
    |                               token.punct
    |                               AUTO
/*  |                               CATCH                                      -- awkward function-definition at end of terminated  */
/*  |                               CLASS                                      -- covered by simple_type_parameter */
/*  |                               DO                                         -- DO ; awkward */
    |                               ENUM
    |                               NAMESPACE
/*  |                               OPERATOR                                   -- OPERATOR , awkward */
    |                               STRUCT
    |                               TEMPLATE
/*  |                               THROW                                      -- covered by throw-expression */
/*  |                               TYPENAME                                   -- covered by simple_type_parameter */
    |                               UNION
/*  |                               '*' | '&' | ELLIPSIS                       -- covered by simple_tree_declaration */
/*  |                               '#'                                        -- used as error iteration flag */
/*  |                               ',' | '{' | '}' | '(' | ')' | ';'          -- awkward - major punctuation */
/*  |                               '/'                                         -- awkward looks like switch */
reserved_id:                        ASM | BREAK | CASE | CONST_CAST | CONTINUE | DEFAULT | DELETE | DYNAMIC_CAST | ELSE | FOR
    |                               GOTO | IF | NEW | REINTERPRET_CAST | RETURN | SIZEOF | STATIC_CAST | SWITCH | TRY | TYPEID | WHILE
    |                               EXPORT
token.punct:                        SCOPE | SHL | SHR | EQ | NE | LE | GE | LOG_AND | LOG_OR | INC | DEC | ARROW | ARROW_STAR | DOT_STAR
    |                               ASS_ADD | ASS_AND | ASS_DIV | ASS_MOD | ASS_MUL | ASS_OR | ASS_SHL | ASS_SHR | ASS_SUB | ASS_XOR
    |                               '[' | ']' | ':' | '?' | '.'
    |                               '+' | '-' | '%' | '^' | '|' | '~' | '!' | '=' | '<' | '>'
    |                               '\'' | '\"' | '\\'
    |                               '@' | '$'

simple_tree_declaration:            /* empty */
    |                               assignment_expression
    |                               special_parameter_declaration
    |                               decl_specifier_prefix simple_tree_declaration

/*---------------------------------------------------------------------------------------------------
 * A.15 Object statements
 *---------------------------------------------------------------------------------------------------*/
object_statements_clause:           ':' '{' object_statement_seq.opt '}'
    |                               ':' '{' object_statement_seq.opt looping_object_statement '#' bang error '}'
                                                                            { UNBANG("Bad object-statements-clause.");}
object_statement_seq.opt:           /* empty */
    |                               object_statement_seq.opt looping_object_statement
    |                               object_statement_seq.opt looping_object_statement '#' bang error ';'
                                                                            { UNBANG("Bad object-statement."); }
looping_object_statement:           start_search looped_object_statement                    { end_search(); }
looped_object_statement:            object_statement
    |                               advance_search '+' looped_object_statement
    |                               advance_search '-'
object_statement:                   ';'
    |                               function_used_block
    |                               '=' initializer_clause ';'
    |                               '(' expression_list ')' ';'
    |                               file_dependency_declaration
    |                               file_placement_declaration
    |                               filespace_specifier semi
    |                               meta_control_statement
    |                               AUTO meta_control_statement
    |                               AUTO meta_expression_statement
    |                               AUTO meta_function_definition
    |                               derived_clause object_statement
    |                               derived_clause ':' '{' object_statement_seq.opt '}'
function_used_block:                function_block
    |                               function_try_block
    |                               ctor_initializer ';'
    |                               USING file_id_list function_used_block
    |                               segment function_used_block
segment:                            BODY
    |                               ENTRY
    |                               EXIT
    |                               POST
    |                               PRE

/*---------------------------------------------------------------------------------------------------
 * A.16 Derivation rules
 *---------------------------------------------------------------------------------------------------*/
derived_clause:                    DERIVED ‘(‘ meta_conditional_expression ‘)’

/*---------------------------------------------------------------------------------------------------
 * A 17.1 meta-names
 *---------------------------------------------------------------------------------------------------*/
meta_id:                           id
    |                              meta_simple_type
    |                              AUTO
meta_scope:                        meta_id SCOPE
meta_nested_id:                    meta_id
    |                              meta_scope ‘~’ meta_id
    |                              meta_scope meta_nested_id
meta_scoped_id:                    meta_nested_id
    |                              global_scope meta_nested_id

/*---------------------------------------------------------------------------------------------------
 * A 17.2 meta-classes
 *---------------------------------------------------------------------------------------------------*/
meta_class_head:                   meta_class_key meta_nested_id
meta_class_specifier:              meta_class_head compound_declaration
    |                              meta_class_head ‘:’ base_specifier_list compound_declaration

/*---------------------------------------------------------------------------------------------------
 * A 17.3 meta-types
 *---------------------------------------------------------------------------------------------------
 *  The MetaType names are not reserved words so form part of identifier and consequently scoped_id */
/* The %prec maximises the length of e.g. unsigned int when followed by e.g int::a */
meta_class_key:                    class_key
    |                              NAMESPACE
meta_non_class_key:                ENUM
    |                              TYPEDEF
    |                              TYPENAME
    |                              USING
    |                              built_in_type_id                %prec SHIFT_THERE
meta_simple_type:                  meta_class_key
    |                              meta_non_class_key
meta_type:                         MetaType
    |                              meta_simple_type

/*---------------------------------------------------------------------------------------------------
 * A 17.4 meta-variables
 *---------------------------------------------------------------------------------------------------*/
/*meta_variable_declaration:                                                          -- covered by meta_expression_statement */

/*---------------------------------------------------------------------------------------------------
 * A 17.5 meta-functions, meta-constructors and meta-destructors
 *---------------------------------------------------------------------------------------------------*/
/*          meta_postfix_expression covers the function name, tree_argument_list.opt covers the paramerer list */
/*          meta_postfix_expression covers the function name, tree_argument_list.opt covers the paramerer list */
meta_function_definition:          meta_postfix_expression ‘(‘ tree_argument_list.opt ‘)’ compound_tree_statement
    |                              STATIC meta_function_definition
    |                              ‘!’ STATIC meta_function_definition
/*  |                              meta_postfix_expression ‘(‘ tree_argument_list.opt ‘)’ object_statements_clause
                                                                                      -- covered by meta_expression_statement */
```

```
/*---------------------------------------------------------------------------------------------------
 * A 17.6 meta-statements
 *---------------------------------------------------------------------------------------------------*/
meta_control_statement:             CASE constant_expression ‘:’ declaration
    |                               DEFAULT ‘:’ declaration
    |                               DO line declaration WHILE ‘(‘ expression ‘)’ semi
    |                               IF line ‘(‘ condition ‘)’ declaration     %prec SHIFT_THERE
    |                               IF line ‘(‘ condition ‘)’ declaration ELSE declaration
    |                               SWITCH line ‘(‘ expression ‘)’ declaration
    |                               WHILE line ‘(‘ condition ‘)’ declaration
    |                               FOR line ‘(‘ for_init_statement condition.opt ‘;’ expression.opt ‘)’ declaration

    |                               jump_statement

/*---------------------------------------------------------------------------------------------------
 * A 17.7 meta-expressions
 *---------------------------------------------------------------------------------------------------*/
meta_primary_head:                  meta_scoped_id
    |                               MetaType meta_nested_id
    |                               meta_non_class_key meta_nested_id
    |                               CONST meta_primary_id
meta_primary_id:                    meta_primary_head
    |                               meta_class_head
meta_primary_expression:            literal
    |                               THIS
    |                               meta_primary_id
    |                               ‘(‘ tree_argument_list.opt ‘)’
meta_postfix_expression:            meta_primary_expression
    |                               meta_postfix_expression ‘(‘ tree_argument_list.opt ‘)’
    |                               meta_postfix_expression ‘[‘ expression.opt ‘]’
    |                               meta_postfix_expression ‘.’ declarator_id
    |                               meta_postfix_expression ARROW declarator_id
    |                               meta_postfix_expression INC
    |                               meta_postfix_expression DEC
meta_unary_expression:              meta_postfix_expression
    |                               INC meta_cast_expression
    |                               DEC meta_cast_expression
    |                               ‘*’ meta_cast_expression
    |                               ‘+’ meta_cast_expression
    |                               ‘-’ meta_cast_expression
    |                               ‘!’ meta_cast_expression
    |                               ‘~’ meta_cast_expression
    |                               SIZEOF unary_expression
meta_cast_expression:               meta_unary_expression
meta_pm_expression:                 meta_cast_expression
meta_multiplicative_expression:     meta_pm_expression
    |                               meta_multiplicative_expression ‘*’ meta_pm_expression
    |                               meta_multiplicative_expression ‘/’ meta_pm_expression
    |                               meta_multiplicative_expression ‘%’ meta_pm_expression
meta_additive_expression:           meta_multiplicative_expression
    |                               meta_additive_expression ‘+’ meta_multiplicative_expression
    |                               meta_additive_expression ‘-’ meta_multiplicative_expression
meta_shift_expression:              meta_additive_expression
    |                               meta_shift_expression SHL meta_additive_expression
    |                               meta_shift_expression SHR meta_additive_expression
meta_relational_expression:         meta_shift_expression
```

```
    |                               meta_relational_expression ‘<‘ meta_shift_expression
    |                               meta_relational_expression ‘>’ meta_shift_expression
    |                               meta_relational_expression LE meta_shift_expression
    |                               meta_relational_expression GE meta_shift_expression
meta_equality_expression:           meta_relational_expression
    |                               meta_equality_expression EQ meta_relational_expression
    |                               meta_equality_expression NE meta_relational_expression
meta_and_expression:                meta_equality_expression
    |                               meta_and_expression ‘&’ meta_equality_expression
meta_exclusive_or_expression:       meta_and_expression
    |                               meta_exclusive_or_expression ‘^’ meta_and_expression
meta_inclusive_or_expression:       meta_exclusive_or_expression
    |                               meta_inclusive_or_expression ‘|’ meta_exclusive_or_expression
meta_logical_and_expression:        meta_inclusive_or_expression
    |                               meta_logical_and_expression LOG_AND meta_inclusive_or_expression
meta_logical_or_expression:         meta_logical_and_expression
    |                               meta_logical_or_expression LOG_OR meta_logical_and_expression
meta_conditional_expression:        meta_logical_or_expression
    |                               meta_logical_or_expression ‘?’ meta_conditional_expression ‘:’ meta_conditional_expression

meta_expression_statement:          meta_conditional_expression semi
    |                               meta_primary_head object_statements_clause semi
    |                               meta_class_head object_statements_clause semi
    |                               meta_postfix_expression ‘(‘ tree_argument_list.opt ‘)’ object_statements_clause semi
    |                               meta_postfix_expression ‘[‘ expression.opt ‘]’ object_statements_clause semi

    |                               meta_logical_or_expression assignment_operator tree_statement
    |                               STATIC meta_expression_statement
    |                               ‘!’ STATIC meta_expression_statement

/*---------------------------------------------------------------------------------------------------
 * A 18 Syntax macros
 *---------------------------------------------------------------------------------------------------*/
syntax_macro_definition:            EXPLICIT AUTO meta_type identifier ‘(‘ syntax_macro_parameter_list ‘)’ compound_tree_statement

    |                               EXPLICIT AUTO meta_type identifier ‘(‘ syntax_macro_parameter_list ‘)’ ‘[‘ ‘]’ compound_tree_statement

    |                               EXPLICIT AUTO meta_type identifier ‘(‘ ‘)’ compound_tree_statement

    |                               EXPLICIT AUTO meta_type identifier  ‘(‘ ‘)’ ‘[‘ ‘]’ compound_tree_statement

syntax_macro_parameter_list:        syntax_macro_parameter
    |                               syntax_macro_parameter_list ‘,’ syntax_macro_parameter
syntax_macro_parameter:             meta_type identifier
    |                               meta_type identifier ‘[‘ ‘]’
    |                               identifier
    |                               reserved_id
    |                               token.punct
    |                               ‘;’
    |                               ‘,’
    |                               ‘{‘
    |                               ‘}’
    |                               ‘(‘
    |                               ‘)’
    |                               bang error                                                  { UNBANG(“bad syntax-macro-parameter.”); }
```

```
/*--------------------------------------------------------------------------------------------------
 * A 19 files
 *--------------------------------------------------------------------------------------------------*/
include_declaration:                USING string
    |                               USING ‘/’ INCLUDE string
    |                               USING ‘/’ INCLUDE ‘/’ utility string
    |                               USING ‘/’ utility string
utility:                            EMIT
    |                               FROZEN
    |                               POOL
    |                               UTILITY

file_dependency_declaration:        using_implementation semi
    |                               using_interface semi
using_implementation:               USING ‘/’ IMPLEMENTATION file_use
    |                               USING ‘/’ IMPLEMENTATION ‘=’ file_use
using_interface:                    USING ‘/’ INTERFACE file_use
    |                               USING ‘/’ INTERFACE ‘=’ file_use
file_use:                           file_id
    |                               file_entity

file_placement_declaration:         export_implementation semi
    |                               export_interface semi
    |                               EXPORT ‘/’ NOIMPLEMENTATION semi
export_implementation:              EXPORT ‘/’ IMPLEMENTATION implementation_file
    |                               EXPORT ‘/’ IMPLEMENTATION ‘=’ implementation_file
export_interface:                   EXPORT ‘/’ INTERFACE interface_file
    |                               EXPORT ‘/’ INTERFACE ‘=’ interface_file
implementation_file:                file_id
    |                               file_entity
interface_file:                     file_id
    |                               file_entity

file_name:                          string
    |                               file_name ‘/’ INTERFACE
    |                               file_name ‘/’ IMPLEMENTATION
    |                               file_name ‘/’ TEMPLATE
    |                               file_name ‘/’ utility
    |                               file_name ‘/’ GUARD ‘=’ string
    |                               file_name ‘/’ NOGUARD
    |                               file_name ‘/’ PATH ‘=’ string
    |                               file_name ‘/’ PREFIX ‘=’ string
    |                               file_name ‘/’ SUFFIX ‘=’ string
file_entity:                        declarator_id
    |                               elaborated_type_specifier
    |                               NAMESPACE scoped_id
file_id:                            file_name
    |                               file_entity ‘/’ IMPLEMENTATION
    |                               file_entity ‘/’ INTERFACE
file_id_list:                       file_id
    |                               file_id_list ‘,’ file_id

filespace_specifier:                NAMESPACE ‘/’ FILE file_name compound_declaration

/*--------------------------------------------------------------------------------------------------
 * Error hnadling aids
 *---------------------------------------------------------------------------------------------------*/
ecarb:                              '}'
    |                               bang error '}'                                          { UNBANG("Extra text ignored before '}'."); }
    |                               bang error ')'                                          { UNBANG("Missing '}'."); }
semi:                               ';'
    |                               bang error ';'                                          { UNBANG("Extra text ignored before ';'.");}

/*---------------------------------------------------------------------------------------------------
 * Back-tracking and context support
 *---------------------------------------------------------------------------------------------------*/
advance_search:                     error               { yyerrok; yyclearin; advance_search(); } /* Rewind and queue '+' or '-' '#' */
bang:                               /* empty */         { BANG(); }   /* set flag to suppress "parse error" */
line:                               /* empty */                   /* Get current line context */
mark:                               /* empty */         { mark(); }        /* Push lookahead and input token stream context onto a stack */
nest:                               /* empty */         { nest(); }        /* Push a declaration nesting depth onto the parse stack */
start_search:                       /* empty */         { start_search(false); }    /* Create/reset binary search context */
start_search1:                      /* empty */         { start_search(true); }     /* Create/reset binary search context */
util:                               /* empty */           /* Get current utility mode */
```

D Command Line

The FOG command line is

```
fog <tokens> <files>
```

`<files>`	is one or more input files, conventionally using the extension `.fog`.
`-`	may be used to indicate that the standard input be used as an input file.
`<tokens>`	is any combination of the following, with or without spacing between a token such as `-I` and a subsequent text argument denoted as `*`.

D.1 Miscellaneous options

`-help`	Display usage help.
`-q`	Suppress the program identification message.

D.2 Preprocessor options

`-D*`	Define a preprocessor macro value.
`-I*`	
`-i*`	Source include file path(s) (defaults to current directory).

D.3 Variant C++ options

`-long_long_type`	Treat `long long` as built-in (Sun C++ language extension).
`-mbc#`	Bytes in a multi-byte character(4).
`-no_access`	Diagnose access declarations (ARM C++ compatibility).
`-no_bool_type`	Do not treat `bool` as built-in (ARM C++ compatibility).
`-no_namespace`	Treat namespace as a synonym for `class`.
`-no_specialisation_prefix`	Do not require `template<>` for specialisation (ARM C++ compatibility).
`-no_using`	Emit *using-declaration*s as *access-declaration*s (ARM C++ compatibility).
`-no_wchar_t_type`	Do not treat `wchar_t` as built-in (ARM C++ compatibility).

D.4 Generated C++ options

`-anon_prefix*`	Prefix for “anonymous” names (default is `_anon_`).
`-c++`	Behave more like a C++ compiler.
`-comment_line_numbers`	Enclose `#line` numbers as comments in emitted files.
`-extern_prefix*`	Prefix for `extern` linkage names (default is `_extern_`).
`-no_line_numbers`	Omit `#line` numbers from emitted files.
`-nobanner`	Suppress emitted comment banners (to ease regression testing).
`-t#`	Columns per tab in source files (default 8).

D.5 Output file options

`-cd*`	Emitted implementation file directory path.
`-cp*`	Emitted implementation file prefix.
`-cs*`	Emitted implementation file suffix.
`-ctd*`	Emitted template implementation file directory path.
`-ctp*`	Emitted template implementation file prefix.

-cts*	Emitted template implementation file suffix.
-f	Force file emission (bypass redundancy comparisons).
-global*	(File)name of the global namespace.
-hd*	Emitted interface file directory path.
-hp*	Emitted interface file prefix.
-hs*	Emitted interface file suffix.
-htd*	Emitted template interface file directory path.
-htp*	Emitted template interface file prefix.
-hts*	Emitted template interface file suffix.
-log*	Log file name (duplicates standard error).
-max_errors#	maximum number of errors before program termination (100).
-max_warnings#	maximum number of warnings before program termination (0).
-nc	Notify emitted file names that are created.
-ne	Notify emitted file names that are suppressed through equivalence.
-o*	File name for make dependencies between source and emitted files.

D.6 Diagnostic options

-readonly	Just read source files to gather token count statistics.
-statistics	Emit program performance statistics.
-z2h	Display each token passed between lexer and hash parser.
-z2l	Display each token passed between lexer and locate parser.
-z2m	Display each token passed between lexer and main parser.
-z2r	Display each token passed between lexer and replace parser.
-z2s	Display each token passed between lexer and substitute parser.
-zd	Delete all objects rigorously on exit (for testing with purify).
-zf	Display file name as each entity is (re)positioned.
-zi	Display each input and macro line.
-zl	Display the behaviour of the lexer.
-zm	Display each main object made by the parser.
-zs	Display changes to the default parser scope.
-zt	Display each token passed between lexer and main parser.
-zu	Display changes to actual declaration utility.
-zx	Display full hex address of each object in diagnostics.
-zy	Display yacc parser progress.
-zz	Repeat certain failed invocations after generating an error to aid debug.

D.7 Predefined macros

```
#define __STDC__ 0
#define __cplusplus 0
```

E Built-In Functionality

This appendix describes the built-in functionality, or more accurately the potential built-in functionality of FOG, since only about half of what is described has actually been implemented and because usage of FOG for a variety of practical applications will probably reveal requirements for further built-in support.

E.1 Built-in meta-classes

E.1.1 `auto`

All meta-classes (and meta-namespaces) ultimately inherit from the `::auto` meta-class, which has no functionality. Its positioning at the root enables meta-program code to affect all classes by composition with its initially empty meta-constructor or meta-destructor. For instance:

```
auto auto::~auto()
{
    $std::diagnostic(@This);
}
```

generates a diagnostic message for every meta-class in an application, since the meta-destructor of the root meta-class is inherited by and consequently executed during meta-destruction of every meta-class including those for built-in types.

E.2 Built-in meta-namespaces

E.2.1 `std`

A variety of generic support facilities are provided by built-in meta-functions. These built-in functions are incorporated as part of the `std` meta-namespace. This avoids cluttering the global meta-namespace directly, or indirectly through introduction of a new namespace. The `std` meta-namespace already exists and is otherwise empty since C++ reserves `std` for language support but has no meta-functionality.

```
auto nil std::diagnostic(string aString)
auto nil std::error(string aString)
auto nil std::warning(string aString)
```

These meta-functions provide the only method for communication between a meta-program and the programmer. The string argument is emitted to standard out (and any log-file) classified as either a diagnostic warning or error message.

Warning messages are prefixed by “WARNING --” and increment the overall warning count.

Error messages are prefixed by “ERROR --” and increment the overall error count.

Obviously an extension to support meta-streams and `std::cerr` would be more powerful.

```
auto string std::get_cpp(string aString)
```

An almost universally available preprocessor extension is the ability to pass macro definitions with an invoking command line such as

```
    cc -DDEBUG_LEVEL=4 ...
```

FOG also supports `-D` as a command line option (see Appendix D).

FOG provides access to the preprocessor definition name-space via the `get_cpp` built-in meta-function, which takes the name to be looked up in the Cpp name-space as an argument and returns its value.

```
    auto int debugLevel = $std::get_cpp("DEBUG_LEVEL");
    auto if (debugLevel > 4) /* ... */;
```

`auto string std::get_env(string aString)`

Definitions may be acquired from the programming environment by using `std::get_env`, which just invokes the POSIX `getenv` routine.

```
const char *logName = $std::get_env("LOGNAME");
```

```
auto string std::date()
auto string std::file()
auto string std::time()
```

These functions return the current date, file and time and replace the ANSI C preprocessor symbols `__DATE__`, `__FILE__` and `__TIME__`.

```
auto token std::parse(string aString)
auto token std::parse_tokens(token[] someTokens)
auto token[] std::tokenize(string aString)
```

The `parse` meta-function provides necessary support for character- and token-level substitution. It performs lexical and syntactical analysis of `aString` to return the equivalent syntax tree.

`parse` is equivalent to successive calls of `tokenize` and then `parse_tokens`, to perform lexical and syntactic analysis respectively. It is not clear how much, if any of this functionality is necessary or even desirable.

E.3 Built-in meta-variables

The following meta-variables are built-in (to the `token` meta-type and so inherited by all meta-types).

`Namespace`

Identifies the current namespace, which is necessary to ensure that declarations occurring nominally within one class can be rescoped to be placed elsewhere.

```
class ThisClass
{
    class NestedClass {};
    class ${Namespace}::SiblingClass {};
};
```

This works whether `ThisClass` is a class in the traditional unnamed global namespace or a class in a named namespace.

When invoked directly within a namespace, `Namespace` identifies the namespace, not its enclosing namespace.

`OuterNamespace`

`OuterNamespace` differs from `Namespace` when invoked for a namespace by returning the immediately enclosing namespace, if there is one or the global namespace otherwise.

Thus `${Namespace}::${OuterNamespace}` first locates the current namespace and then locates its immediately enclosing namespace. Eventually after sufficient iterations `OuterNamespace` always returns the global namespace.

It is an error to traverse potential declarations beyond their defined ancestry.

`OuterScope`

Similarly `OuterScope` differs from `Scope` when invoked for a scope (class, struct, union) by returning the immediately enclosing scope, if there is one or the namespace otherwise.

Thus `${Scope}::${OuterScope}` first locates the current scope and then locates its immediately enclosing scope. Eventually after sufficient iterations `OuterScope` always returns the namespace.

It is an error to traverse potential declarations beyond their defined ancestry.

`Scope`

Identifies the current scope, which may be a class, filespace, linkage, namespace, struct or union.

When invoked directly within a scope, Scope identifies the scope, not its enclosing scope.

`Super`

Super identifies the primary base class. It is a short form for (and much more efficient than) `Scope::bases()[0]`.

Use of `Super` for a class without a base-class is an error. The base-class determined by Super ignores meta-base classes, and so is not exactly equivalent to `Scope::bases()[0]`, which could resolve a meta-base class.

`This`

This identifies the current declarative region which is the same as Scope, when invoked within the context of a class or namespace. However, when invoked within the context of a variable or function, This refers to the variable or function, and provides access to object-scoped meta-declarations in preference to occluded meta-declarations from the class scope.

E.4 Built-in meta-functions

The inheritance relationships between the built-in meta-types are described in Section 3.1.2 and shown below using indentation.

```
token
    declaration/statement
        expression
            assignment_expression
                constant_expression
            character
            number
            string
            name

                entity
                    base
                    enumerator
                    namespace_alias
                    object
                        exception
                        function
                        meta_function
                        meta_parameter
                        meta_variable
                        parameter
                        template_parameter
                        typedef
                        using
                        variable
                    scope
                        filespace
                        linkage
                        meta_class
                        namespace
                        type
                            built_in
                            class
                            enum
                            typename
                keyword
                    decl_specifier
                        cv_qualifier
                    identifier
                    meta_type
                        class_key
                    reserved
                specifier
                    base_specifier
                    enumerator_definition
                    namespace_alias_definition
                    object_specifier
                        exception_declaration
                        function_specifier
                        meta_function_specifier
                        meta_parameter_specifier
                        meta_variable_specifier
                        parameter_specifier
                        template_parameter_specifier
                        typedef_specifier
                        using_declaration
                        variable_specifier
                    scope_specifier
                        filespace_specifier
                        linkage_specification
                        meta_class_specifier
                        namespace_definition
                        type_specifier
                            built_in_type_specifier
                            class_specifier
                            elaborated_type_specifier
                            enum_specifier
                using_directive
    exception_specification
    handler
    iterator
    modifier
        array_modifier
        function_modifier
        pointer_modifier
        reference_modifier
    nil
    punctuation
```

All meta-types inherit from `token`, where

- polymorphism between scalars and lists is established
- default implementations of all built-in functions are provided
- meta-type type testing predicates are implemented

The built-in meta-functions are described in the following sections, with the description at the lowest level in the inheritance hierarchy at which use of the meta-function is meaningful. Use below that level for a predicate testing meta-function just returns false. For other meta-functions an error message is generated.

E.4.1 array_modifier

E.4.2 assignment_expression

E.4.3 base and base_specifier

```
auto class base::class()
auto class_specifier base_specifier::class()
auto bool base::is_auto()
auto bool base_specifier::is_auto()
auto bool base::is_private()
auto bool base_specifier::is_private()
auto bool base::is_protected()
auto bool base_specifier::is_public()
auto bool base::is_public()
auto bool base_specifier::is_public()
auto bool base::is_virtual()
auto bool base_specifier::is_virtual()
```

E.4.4 built_in and built_in_type_specifier

E.4.5 character

```
auto character::operator character()
auto character::operator identifier()
auto character::operator number()
auto character::operator string()
```

E.4.6 class and class_specifier

E.4.7 class_key

E.4.8 constant_expression

E.4.9 cv_qualifier

E.4.10 decl_specifier

E.4.11 declaration

see statement

E.4.12 elaborated_type_specifier

```
auto class_key elaborated_type_specifier::class_key() []
auto bool elaborated_type_specifier::is_auto()
auto bool elaborated_type_specifier::is_class()
auto bool elaborated_type_specifier::is_namespace()
auto bool elaborated_type_specifier::is_struct()
auto bool elaborated_type_specifier::is_typename()
auto bool elaborated_type_specifier::is_union()
```

E.4.13 entity and specifier

```
auto string entity::implementation_file()
auto string specifier::implementation_file()
auto string entity::interface_file()
auto string specifier::interface_file()
```

E.4.14 enum and enum_specifier

```
auto enumerator enum::enumerators() []
auto enumerator_definition enum_specifier::enumerators() []
```

E.4.15 enumerator and enumerator_definition

```
auto number enumerator::value()
auto expression enumerator_definition::value()
```

E.4.16 exception and exception_declaration

E.4.17 exception_specification

E.4.18 expression

```
auto expression expression::value()
```

E.4.19 filespace and filespace_specifier

E.4.20 function, function_modifier and function_specifier

```
auto exception function::exceptions() []
auto exception_declaration function_modifier::exceptions() []
auto exception_declaration function_specifier::exceptions() []
auto parameter function::parameters() []
auto parameter_specifier function_modifier::parameters() []
auto parameter_specifier function_specifier::parameters() []
auto function_specifier function::signature() []
auto function_specifier function_modifier::signature() []
auto function_specifier function_specifier::signature() []
```

E.4.21 handler

E.4.22 identifier

```
auto identifier::operator character()
auto identifier::operator identifier()
auto identifier::operator number()
```

returns the result of a text to numeric conversion, which usually involves an error message.

```
auto identifier::operator string()
```

E.4.23 iterator

A typical idiomatic use of an iterator is shown in the following example

```
auto for (iterator i = $MyClass::variables(); i; ++i)
    auto if (i->is_static())
        const char *MyClass::names[] = { ""$i->name() };
const char *MyClass::names[] = { 0 };
```

in which composition of array elements is used to build a null terminated list of the names of the member variables of `MyClass`.

Note that an iterator maintains a copy of the identities of the elements of the iteration domain, but not of their contents. Therefore addition of an addition base class during a traversal of base classes will not be detected by the iteration. Modification of an iteration element prior to traversal does affect the iteration.

`auto iterator::iterator()`

constructs an iterator already out-of-domain.

`auto iterator::iterator(token [])`

constructs an iterator to iterate over and from the start of the exposed list. The identities of the elements in the list are copied.

`auto iterator::iterator(iterator)`

constructs a copy of an iterator, which involves a copy of the identities of the elements in the iteration domain and of the position within the domain.

`auto nil iterator::operator=(token [])`

assigns an iterator to iterate over and from the start of the exposed list. The identities of the elements in the list are copied.

`auto nil iterator::operator=(iterator)`

assigns a copy of an iterator, which involves a copy of the identities of the elements in the iteration domain and of the position within the domain.

`auto iterator::operator number()`

returns true as long as the iterator remains within the iteration domain

`auto nil iterator::operator++()`

advances the iterator through the iteration domain, setting out-of-domain once the top edge passed.

`auto nil iterator::operator--()`

rewinds an iterator one step back through the iteration domain, setting out-of-domain once the bottom edge passed.

`auto token iterator::operator->()`

returns the current element in the iteration domain, generating an error if out of domain.

`auto token iterator::operator*()`

returns the current element in the iteration domain, generating an error if out of domain.

E.4.24 keyword

E.4.25 linkage and linkage_specification

```
auto string linkage::value()
auto string linkage_specification::value()
```

E.4.26 meta_class and meta_class_specifier

E.4.27 meta_function and meta_function_specifier

```
auto meta_parameter meta_function::meta_parameters() []
auto meta_parameter_specifier
   meta_function_specifier::meta_parameters() []
```

E.4.28 meta_parameter and meta_parameter_specifier

E.4.29 meta_type

E.4.30 meta_variable and meta_variable_specifier

```
auto token meta_variable::value()
auto token meta_variable::value()
```

E.4.31 modifier

```
auto bool modifier::is_array_modifier()
auto bool modifier::is_const()
auto bool modifier::is_function_modifier()
auto bool modifier::is_pointer_modifier()
auto bool modifier::is_reference_modifier()
auto bool modifier::is_volatile()
```

E.4.32 name

```
auto name name::full_name()
auto name name::name()
```

E.4.33 namespace and namespace_definition

E.4.34 namespace_alias and namespace_alias_definition

```
auto namespace namespace_alias::value()
auto name namespace_alias_definition::value()
```

E.4.35 nil

The `nil` meta-type is used for invalid and zero values.

E.4.36 number

```
auto number::operator character()
auto number::operator identifier()
auto number::operator number()
auto number::operator string()
```

E.4.37 object and object_specifier

```
auto bool object::is_const()
auto bool object_specifier::is_const()
auto bool object::is_static()
auto bool object_specifier::is_static()
auto bool object::is_volatile()
auto bool object_specifier::is_volatile()
auto meta_function object::meta_functions() []
auto meta_function_specifier object_specifier::meta_functions() []
```

returns the immediate list of member meta-functions

```
auto meta_variable object::meta_variables() []
auto meta_variable_specifier object_specifier::meta_variables() []
```

returns the immediate list of member meta-variables

```
auto modifier object::modifiers() []
auto modifier object_specifier::modifiers() []
```

returns the list of declarator modifiers

```
auto type object::type()
auto type_specifier object_specifier::type()
```

E.4.38 parameter and parameter_specifier

E.4.39 pointer_modifier

E.4.40 punctuation

E.4.41 reference_modifier

E.4.42 reserved

E.4.43 scope and scope_specifier

```
auto base scope::all_bases() []
auto base_specifier scope_specifier::all_bases() []
```

returns the transitive list of base-specifiers (including meta-bases)

```
auto scope scope::all_classes() []
auto scope_specifier scope_specifier::all_classes() []
```

returns the transitive list of nested classes

```
auto function scope::all_functions() []
auto function_specifier scope_specifier::all_functions() []
```

returns the transitive list of member functions

```
auto type scope::all_types() []
auto type_specifier scope_specifier::all_types() []
```

returns the transitive list of member types

```
auto typedef scope::all_typedefs() []
auto typedef_specifier scope_specifier::all_typedefs() []
```

returns the transitive list of member typedefs

```
auto variable scope::all_variables() []
auto variable_specifier scope_specifier::all_variables() []
```

returns the transitive list of member variables

```
auto base scope::bases() []
auto base_specifier scope_specifier::bases() []
```

returns the immediate list of base-specifiers (including meta-bases)

```
auto class_key scope::class_key() []
auto class_key scope_specifier::class_key() []
auto scope scope::classes() []
auto scope_specifier scope_specifier::classes() []
```

returns the immediate list of nested classes

```
auto entity scope::friends() []
auto specifier scope_specifier::friends() []
auto function scope::functions() []
auto function_specifier scope_specifier::functions() []
```

returns the immediate list of member functions

```
auto bool scope::is_auto()
auto bool scope_specifier::is_auto()
auto bool scope::is_boundary()
```

returns true if there is a pure-virtual function in an immediate base class but no pure-virtual in this class

```
auto bool scope::is_class()
auto bool scope_specifier::is_class()
auto bool scope::is_leaf()
```

returns true if there are no derived classes

```
auto bool scope::is_pure()
```

returns true if there is a pure-virtual function

```
auto bool scope::is_struct()
auto bool scope_specifier::is_struct()
auto bool scope::is_union()
auto bool scope_specifier::is_union()
auto typedef scope::typedefs() []
auto typedef_specifier scope_specifier::typedefs() []
```

returns the immediate list of member typedefs

```
auto type scope::types() []
auto type_specifier scope_specifier::types() []
```

returns the immediate list of member types

```
auto variable scope::variables() []
auto variable_specifier scope_specifier::variables() []
```

returns the immediate list of member variables

E.4.44 statement

E.4.45 string

Identical strings are represented by the same meta-object, so content comparison is performed by `operator==` and `operator!=`.

```
auto string::operator character()
auto string::operator identifier()
auto string::operator number()
auto string::operator string()
```

E.4.46 template_parameter and template_parameter_specifier

E.4.47 token

The token meta-type provides a default implementation for all built-in meta-functions. Predicate meta-functions such as `is_virtual()` that return a `bool` value are implemented to return `false`. All other meta-functions generate an error message.

```
auto bool token::ambiguous(expression anExpression)
auto bool token::defined(expression anExpression)
```

These two meta-functions test for the presence and multi-presence of declarations. They return non-zero if the expression is ambiguous (has multiple definitions) or is defined (has at least one definition).

These meta-functions may be used as predicates to avoid errors in subsequent code. They take no account of whether a declaration is enabled or not. Thus a declaration for use only in leaf classes is regarded as defined at the root class and all its derived classes, even though the declaration is disabled at non-leaves.

```
auto token token::find(expression anExpression) []
```

returns all meta-declarations visible in the meta-name-space

`auto entity token::find_entity(expression anExpression) []`
: returns all visible entities (names or types)

`auto object token::find_name(expression anExpression) []`
: returns all visible names

`auto type token::find_type(expression anExpression) []`
: returns all visible types)

`auto token token::get(expression anExpression)`
: returns the meta-declaration visible in the meta-name-space, generating an error if undefined or ambiguous.

`auto entity token::get_entity(expression anExpression)`
: returns the visible entity (name or type), generating an error if undefined or ambiguous.

`auto object token::get_name(expression anExpression)`
: returns the visible name, generating an error if undefined or ambiguous.

`auto type token::get_type(expression anExpression)`
: returns the visible type, generating an error if undefined or ambiguous.

`auto bool token::is_exposed()`
: returns true if token comprises an exposed list.

`auto bool token::is_list()`
: returns true if token comprises an encapsulated or exposed list.

`auto bool token::is_`*`meta_type`*`()`
: returns true if token is at least as specialised as *`meta_type`*.

`auto unsigned token::length()`
: returns the number of elements in an encapsulated or exposed list.

`auto meta_type token::meta_type()`
: returns the describing meta-type.

`auto token token::sub_list(unsigned start, signed length) []`
: returns an exposed list comprising the `abs(length)` elements from `start` to `start+length` exclusive, generating an error if any element out-of range.

`auto token token::operator[](unsigned index)`
: returns the `index` element, generating an error if out-of range.

`auto token token::operator+(token tokens[]) []`
: return a new list comprising tokens appended to this list.

`auto nil token::operator+=(token tokens[]) []`
: appends tokens.

E.4.48 type and type_specifier

E.4.49 typedef and typedef_specifier

```
auto type typedef::value()
auto type_specifier typedef_specifier::value()
```

E.4.50 typename

E.4.51 using and using_declaration

E.4.52 using_directive

E.4.53 variable and variable_specifier

```
auto expression variable::value()
auto expression variable_specifier::value()
```

F Implementation

The presentations of the FOG extensions to C++ in Chapter 2, their semantics in Chapter 3, and a novel parsing approach in Chapter 4 are all fairly substantial and so a number secondary issues are relegated to this appendix.

A brief discussion of the difficulties of enhancing C++ syntax is followed by a description of some syntax extensions that were considered and why they were not implemented.

We then describe how the superset grammar approach resolves specific C++ parsing difficulties and outline the activities needed during the semantic processing to recover lost syntactic resolution.

F.1 Syntax Implementation

Providing additional syntax in C++ without introducing new reserved words or totally esoteric meanings for punctuation is rather difficult, since most simple syntax using non-reserved words is covered by a *simple-declaration*.

The multi-pass implementation used `set` rather than `export`, and use rather than `using` for the file syntaxes. This caused no ambiguity at the declaration level since `/` cannot appear except as an initializer in a declaration.

```
    set / implementation = "file" ;
```

The above of course is a syntactically valid expression, and a little provocative given the presence of `set` as a template name in the Standard Template Library.

Migration to the superset grammar resulted in a generalised parser being used for declarations and so `set` became difficult to disambiguate syntactically. The syntax was therefore changed to its current form. `export` and `using` are not so very far away from the intended meanings of specify-output-file, and specify-input-dependency.

Overloading reserved words is undesirable and confusing as exemplified by the many meanings of `static`. The reuse of `auto` to mean *meta* is equally unsatisfactory. Other new overloadings are relatively clear, since the reserved word is followed by a switch.

F.1.1 `!const` and `!volatile`

Provision of `!static` for more explicit control of composition suggests that `!const` and `!volatile` should also be provided. However the situation is not quite the same. `const` and `volatile` form part of a function signature and so there is no possibility that composition should ever interpret a missing `const` as `const`. A missing `const` always means `!const`. This reasoning makes `!const` unnecessary but does not preclude its provision as a documentation aid.

Provision of `!const` causes implementation problems too, since an ambiguity arises in a generalised parse between

`(type) ! a     // Cast of complement`
`(type) ! const // Very degenerate` *parameter-declaration-clause* and *cv-qualifier*s

Two tokens of lookahead are required to resolve the ambiguity.

Supporting `!const` as an extension of *type-specifier* rather than *cv-qualifier* solves this problem allowing usage everywhere except following *parameter-declaration-clause*s. However this support was a little irregular and since the sole purpose was as a documentation aid, it was decided to omit `!const` and `!volatile`.

F.1.2 Member variable delegation

Larger objects may be built from smaller objects using inheritance or aggregation. Inheritance has the convenient property that the entire interface of the base object is visible as part of the larger object, whereas aggregation makes none of the interface available. The implementor is faced with an all or nothing choice for delegation.

A *re-using-declaration* that mentions a member-variable could be interpreted as a directive to support delegation so that:

```
class Proxy
{
    Client *_member_variable;
    using _member_variable;
};
```

automatically synthesises delegation routines such as

```
int f(double b) { return _member_variable->f(b); }
```

for every accessible function of `_member_variable`. More selective synthesis could be achieved by naming functions

```
using _member_variable->f;
```

A further extension was considered whereby the client could group a number of functions to establish a view:

```
class Client
{
    namespace/view ProxyView
    {
        int f(double b);
    };
};
```

so that all functions identified as part of the `ProxyView` would automatically be delegated by:

```
using _member_variable->ProxyView;
```

This extension then ensures that addition of a further function to `Client::ProxyView` automatically adds a delegating function to the `Proxy`.

This is useful, but vulnerable to practical considerations:

- it may be desirable to handle null tests in the delegation routines
- it may be necessary to add `*this` as an extra argument during delegation

Customized formatting of the synthesised routine is not easily handled by a standardised approach. Customized formatting is available via meta-programming and so the concept of using a member variable and delimiting part of an interface through a view is no longer supported. The effect can be achieved by:

```
class Client
{
    int f(double b);
    int g(double b);
    void g();
    auto declaration ProxyView[] = { f, g(double) };
};
class Proxy : auto Client
{
    Client *_member_variable;
};
auto Proxy::Proxy()
{
    $delegate(_member_variable, Client::ProxyView);
};
```

An appropriate implementation of delegate can then be written or accessed from a meta-library to iterate over the declarations in Client::ProxyView and to synthesize delegation routines via _member_variable.

[The meta-inheritance of Proxy from Client establishes a meta-compilation order dependency so that meta-construction of Client occurs before Proxy, ensuring that any member-functions declared by Client's meta-constructor exist before Proxy's meta-constructor synthesises its delegation functions.]

This example is far from bomb-proof; functions added to Client during meta-main execution or by later meta-constructors will not receive delegate treatment. An implementation proof against arbitrary composition requires Client to provide a meta-function to register contributions to ProxyView, and to register classes interested in viewing the contents of ProxyView. The registration function can notify registered viewers when any change occurs, and the meta-destructor for the client can verify that no functions have appeared without passing through the registration function. This approach is reliable but slow and it is readily supported by standard meta-functions from a meta-library. Alternative meta-object protocols such as those available with CLOS or OpenC++ provide more direct hooks. FOG could support the option for a user-definable meta-function:

```
    auto declaration Client::add_function(declaration);
```

which would be invoked for each addition and return a possibly modified declaration or even a completely different set of declarations. The modest complexity and inefficiency of detecting, maintaining and invoking such functions does not seem to be justified for simple applications.

F.1.3 Pattern names

AspectJ [Lopes98] supports the addition of code to all functions whose name matches some pattern, which is very useful for adding tracing code, since an "entering function x" diagnostic is easily attached to everything. It is not clear whether a more partial pattern match is useful without imposing a potentially awkward lexical convention on function names just to satisfy the pattern match.

Support for pattern matching in AspectJ is relatively easy since it is Java-based and so there is no overloading and the * character is free for use in patterns. Direct adoption of the same policy in FOG would not be possible since overloads need resolution and * is used for pointers. The problem of pattern syntax is soluble by expressing pattern names as strings. Thus

```
    void "print*"(ostream&, "*")
```

might select all functions whose name starts with print, that return void and have an ostream& as a first parameter.

An alternative solution is available by meta-programming. A pattern-matching routine filters the set of all member function names and invokes a customised meta-function for each of the filtered names. Much of this functionality can be provided by a meta-library, and would be considerably assisted by a built-in meta-function to support an elemental pattern match.

```
    auto::~auto()
    {
        $std::map(customMetaFunction,
                    $std::filter($functions(), "print*"));
    }
```

for which std::map() and std::filter() represent to-be-implemented components of a meta-library, functions() is built-in to FOG. customMetaFunction performs the per-function meta-programming, and might also be a standard meta-library component. Invocation from the meta-destructor of auto ensures that the custom functionality is applied to all classes.

This an area for further research. It would appear that most of the solution lies in the domain of a meta-library rather than fundamental FOG functionality.

F.1.4 #line directive

The #line directive is not used in source programs generated by human beings. It provides a very simple but useful mechanism for automatic source code generators to ensure that compilers and debuggers refer to the original source lines rather than some scrambled intermediate. #line performs this role adequately and needs no replacement, although an extension with a more cryptic free format spelling could be considered to free the # token once Cpp has been discontinued. The line-literal would be discarded along with whitespace in translation phase 7.

line-literal:
  ~ { *line-context-seq* }

line-context-seq:
  line-context
  line-context-seq line-context

line-context:
  domain$_{opt}$ *file-line*$_{opt}$ *line-number*

domain:
  identifier

file-line:
  string-literal

line-number:
  decimal-literal

The optional *domain* supports definition of line numbers for more than one source domain, with the list of contexts supporting multiple contexts. For instance code passed first through yacc++ and then cfront needs to report both yacc input and yacc output line numbers, so that an enhanced cfront might include a *line-literal* such as

```
~{"Grammar.y" 21 cxx "yacc.tab.c" 127}
```

The first domain is unspecified and defaults to source. With this information, enhanced debugging systems and their users can select the appropriate file upon which to perform source-level debugging.

F.2 Resolution of parsing difficulties

This section reviews the specific problems that arise in parsing C++ and shows how they are resolved using the multi-pass or superset grammar approach.

F.2.1 Context-free problems

The C++ grammar violates the requirement for a context-free grammar, since “New context-dependent keywords are introduced into a program by typedef, namespace, class, enumeration and template declarations” (§A.1-1).

When an identifier is encountered, semantic information is needed, since there is a context-dependency on type names and on template names.

typedef-name:	*identifier*
namespace-name:	*original-namespace-name* *namespace-alias*
original-namespace-name:	*identifier*
namespace-alias:	*identifier*
class-name:	*identifier* *template-id*
enum-name:	*identifier*

template-name:	*identifier*
template-id:	*template-name* < *template-argument-list* >

If the grammar defined by the standard is to be followed very closely, semantic information is apparently needed to classify identifiers into one of

- *class-name*
- *enum-name*
- *identifier* (anything else)
- *namespace-alias*
- *original-namespace-name*
- *template-name*
- *typedef-name*

However quite what constitutes context-dependency in the grammar depends on how much the grammar is intended to specify. A complete grammar could include all the language constraints on definition / reference ordering, template instantiation and function overloading. From such a strict perspective almost all languages are context-dependent (at least when implemented using a first order grammar).

When a grammar is solely concerned with the conversion of a token stream into an Abstract Syntax Tree to support a subsequent semantic analysis, the requirements on the grammar are much less stringent. In Section 4.7.1 it was shown that type information was not necessary. In Section 4.8.2 it was shown that a lack of template name information could also be accommodated, although the consequent complexity might not justify that approach.

The context-dependencies are therefore reviewed from the less ambitious standpoint of AST creation.

F.2.1.1 `#include` anomaly

The arguments of the Cpp `#include` directive use non-standard forms. In the

```
#include "string"
```

form, there is no recognition of escape sequences. And in the

```
#include <file>
```

form, the angle brackets act as string delimiters, rather than template delimiters or arithmetic operators.

Resolution

This context-dependence is readily resolved within the lexical analysis processing by switching the lexer into an alternate state starting at the recognition of a `#include` and continuing to the end of the line. While in this alternate state a different tokenization policy is adopted,

F.2.1.2 Type information

A full semantic interpretation of a C++ program obviously requires a knowledge of the types. Unfortunately this information is also needed for a complete syntactic disambiguation of

- declaration / expression ambiguity (Section F.2.5.1)
- parenthesised-call / cast-parenthesis ambiguity (Section F.2.5.2)
- parenthesised-binary / cast-unary ambiguity (Section F.2.5.2)
- call / functional-cast ambiguity (Section F.2.5.3)
- new placement / initializer ambiguity (Section F.2.5.5)
- `sizeof` type / value ambiguity (Section F.2.5.6)
- `typeid` type / value ambiguity (Section F.2.5.7)
- template argument type / value ambiguity (Section F.2.5.8)

F.2.1.3 < as template-end or less than (§14.2-3)

A misparse resulting from the lack of template context is difficult to resolve because the two meanings of < and > do not result in localised errors to the tree structure. The arithmetic operators are infix binary operators and have no requirements for associated punctuation, whereas the template brackets must be paired.

A problem arises for an expression such as

```
V < W < X > ( Y ) > ( Z )
```

which, in C, would be four unambiguous comparisons. However in C++, there are alternate meanings depending upon which of V or W are template names.

Correct determination of template names requires that names be resolved in the correct scope and may require a template to be instantiated.

```
a->b<int>::c<...
```

a in current scope.

b in scope of a.

c in scope of b`<int>`.

Resolution

Correcting a template misparse is an inconvenient but not a particularly difficult AST rearrangement. Although a back-tracking search for a syntactically consistent interpretation is of exponential complexity, the implementation and results presented in Section 4.8.2 show that this does not arise in practice.

F.2.1.4 > as template-end or greater than (§14.2-3)

Within a template, the meaning of an unnested > changes to close the template rather than perform an arithmetic operation. This is not a context-dependency, since the interpretation is dependent on the preceding parse context. The parser knows whether it is in a template and so the grammar can be written to resolve the conflict.

Resolution

The expression rules with higher precedence than > are duplicated to omit the arithmetic > behaviour. This duplicated behaviour is used whenever parsing within a template. As a result the parser keeps track of in / out of template context and distinguishes between > nested within parentheses or brackets as part of its normal operation. The cost is about 25 extra rules.

F.2.1.5 Meta-types

The use of meta-types in Section 2.1.5.5 introduces a form of context dependency. In the declaration

```
auto expression e = a + b & c;
```

the syntax used following the = is determined by the `expression` meta-type. This dependency could be eliminated by flattening the grammar to support each meta-type individually in every relevant rule.

However when the same dependency exists in parsing a function argument

```
auto bool meta_function(expression e) { ... }
if (meta_function(a + b & c))
```

it appears that the semantic knowledge of the meta-type of the meta-function parameter must influence the syntactic parsing.

Resolution (multi-pass)

The multi-pass implementation of FOG was syntax-driven, using the meta-type of a meta-function or meta-variable to guide the parse. This only required loose coupling between syntactic and semantic processing, since meta-function and meta-variable definitions can change only at the end of a statement or declaration.

Resolution (superset)

The superset implementation of FOG parses for generic syntax elements, and so syntactic and semantic processing are isolated and there is no meta-type context dependency.

F.2.2 Trivial Ambiguities

The C++ grammar “accepts a superset of valid C++ constructs” (§A-1). Two simple examples of the ambiguities that arise from the overlap between subgrammars are described below.

F.2.2.1 Empty statement

In the syntax for a statement:

statement:
    expression-statement
    simple-declaration   *// As part of a declaration-statement*
    ...

expression-statement:
    expression$_{opt}$ `;`

simple-declaration:
    decl-specifier-seq$_{opt}$ *init-declarator-list*$_{opt}$ `;`

both *expression-statement* and *simple-declaration* provide a cover for the empty statement comprising just a semicolon. The ambiguity is trivial, but must be eliminated in order to create an unambiguous grammar for an automated parser tool.

F.2.2.2 Template parameter

In the syntax for template parameters

template-parameter:
  type-parameter
  parameter-declaration

```
class name
```

is valid as both a *type-parameter* and a *parameter-declaration*.

F.2.3 Syntactic Ambiguities

[Roskind91] identified the major C++ ambiguities that existed prior to templates. This section provides an update to the list and shows how each can be resolved using a superset parse. This section describes only those ambiguities that are syntactically ambiguous. The next two sections discuss further apparent ambiguities: one that is not ambiguous at all, and others that may be deferred for resolution after syntactic analysis.

F.2.3.1 Dangling `else` (§6.4.1)

The dangling `else` ambiguity arises in languages with no end of `if` marker:

```
if (a)
    if (b)
        ... ;
else                // else (!a) or (!b) ?
    ... ;
```

The ambiguity is resolved by definition to the inner-most `if`, requiring the parser to shift the `else` on to the parser stack and to continue parsing the `if (b)` statement, rather than reducing the stack, completing the `if (b)` statement, and continuing the `if (a)` statement.

Resolution

Resolution of the dangling `else` ambiguity is implemented by using a `%prec` rule.

```
selection_statement:
        "if" '(' condition ')' statement        %prec SHIFT_THERE
    |   "if" '(' condition ')' statement "else" statement
    |   "switch" '(' condition ')' statement
```

The `SHIFT_THERE` precedence specified for the reduction of the shorter rule is lower than that of the `else` token and so the shift of the `else` is favoured.

F.2.3.2 < as template-start or less than (§14.2-3)

[Roskind91] does not report this ambiguity because he did not implement templates.

The template-name problem is a context-dependency and has been described in Section F.2.1.3.

F.2.3.3 Multiply nested scope (§7.1-2)

[Roskind91] does not report this ambiguity because he did not implement arbitrary scope nesting.

There is an ambiguity between

```
A::B ::C
```

and

```
A ::B::C
```

and

```
A::B::C
```

which is resolved by language definition to favour the longest possible *decl-specifier-seq* as the type.

Resolution

A single `%prec` rule in the FOG grammar resolves the conflict that arises from this ambiguity.

```
id_scope:       id "::"

nested_id:      id                      %prec SHIFT_THERE
    |           id_scope nested_id

scoped_id:      nested_id
    |           "::" nested_id
```

F.2.3.4 *new-type-id* (§5.3.4-2), *conversion-function-id* (§12.3.2-4)

The name of a *new-type-id* in a *new-expression*

new-expression:
  `::`$_{opt}$ `new` *new-placement*$_{opt}$ *new-type-id new-initializer*$_{opt}$
  `::`$_{opt}$ `new` *new-placement*$_{opt}$ (*type-id*) *new-initializer*$_{opt}$

new-type-id:
  type-specifier-seq new-declarator$_{opt}$

new-declarator:
  ptr-operator new-declarator$_{opt}$
  direct-new-declarator

direct-new-declarator:
  [*expression*]
  direct-new-declarator [*constant-expression*]

and of a *conversion-function-id*

conversion-function-id:
  `operator` *conversion-type-id*

conversion-type-id:
  type-specifier-seq conversion-declarator$_{opt}$

conversion-declarator:
  ptr-operator conversion-declarator$_{opt}$

may each end in a `*` or `&`, which can cause an ambiguity with respect to a subsequent expression.

```
    new int ** * 0;
    &operator int ** + 0;
```

Each is resolved by definition to maximise the length of the type name.

Resolution

A `%prec` resolves the ambiguity in accordance with the language specification, using the one production `ptr_operator_seq.opt` to implement *conversion-declarator*$_{opt}$ and part of *new-declarator*$_{opt}$.

```
ptr_operator_seq.opt:
        /* empty */   %prec SHIFT_THERE  /* Maximise type length */
    |   ptr_operator ptr_operator_seq.opt
```

F.2.3.5 Array of operator ambiguity

`operator new` and `operator new[]` (and `operator delete` and `operator delete[]`) are valid *declarator-id*s. It is unclear whether

```
    int operator new[];
```

declares an array or a scalar.

Resolution

The ambiguity is removed by

- excluding `operator new[]` and `operator delete[]` from the grammar
- accepting a missing array dimension in an expression

Semantic processing identifies the array form from the parsed array of scalar form.

Alternatively, unnecessary semantic effort can be avoided by retaining the array forms and using two `%prec`s to resolve the two consequent shift-reduce conflicts.

F.2.4 Deep Ambiguities

There are some C++ constructs that require a significant amount of lookahead to determine which of two alternative syntaxes is in use.

F.2.4.1 Bit-field or Inheritance

[Roskind91] identifies an ambiguity following

```
class A { class B :
```

which could form part of an anonymous bit field

```
const int C = 3;
class A { class B : C, D, E = 5; };
                  // C is a bit-field width, D,E are variables
```

or a base class

```
class C {};
class D {};
class E {};
class A { class B : C, D, E {}; };
                  // A::B inherits privately from C, D and E
```

`class A` is not really part of the ambiguity. It just serves to avoid the semantic quibble that there are no bit-fields at global scope. There is in fact no ambiguity anyway, because the inheritance declaration must eventually lead on to an open brace whereas the bit-field can never be followed by an open brace.

The problem is the need to lookahead through an arbitrary long comma-separated list of names until some keyword (such as `public`) or punctuation (such as `*`) clarifies the name list, or until eventually the trailing punctuation resolves the ambiguity.

Resolution

The superset grammar assumes that `class A :` is the start of a class declaration, and back-tracks to the `:` if the *base-specifier-list* is not terminated by a `{`. This incurs only a very minor performance loss, since the use of redundantly qualified anonymous bit-fields is surely rare, and so back-tracking may never occur in practice.

```
class_specifier:
        class_key scoped_id '{' member_specification.opt '}'
    |   class_key scoped_id ':' mark base_specifier_list
                   '{' unmark member_specification.opt '}'
    |   class_key '{' member_specification.opt '}'
    |   class_key ':' base_specifier_list
                   '{' member_specification.opt '}'
    |   "template" class_specifier
```

```
elaborated_type_specifier:
        class_key scoped_id ':' mark error { rewind_colon(); }
    |   class_key scoped_id      %prec SHIFT_THERE
    |   "enum" scoped_id         %prec SHIFT_THERE
    |   "typename" scoped_id
    |   "template" elaborated_type_specifier
```

The %prec on class_key scoped_id resolves two conflicts. It forces a following : to go through the inheritance lookahead test, and suppresses the spurious interpretation as a function-name when followed by a { (Appendix F.2.6.2).

Attempting to resolve this problem by parsing for a shared prefix with a more generalised expression syntax proves to be rather difficult, since the generalisation to share a prefix allows the constructor initializer list to provide a third alternative. This construct also ends in a { and so

```
class A : name {};
```

would satisfy the generalised syntax of both constructor and class inheritance. Generalising the syntax further is not possible since a constructor takes a list of statements whereas a class takes a list of declarations. A statement and a declaration cannot be unified since the syntax for the label of a goto statement is highly ambiguous with respect to an anonymous bit field (see Appendix F.2.6.2).

F.2.4.2 Type I functions

The original form of C function declarations is not normally supported by C++ compilers. The syntax presents challenges in avoiding conflicts, and does not tie in well with the generalised name solution of the superset.

Resolution

The grammars in Appendix B and Appendix C implement Type I function declarations but only at a severe (25%) cost to the parsing efficiency. To avoid ambiguities, a lookahead parse is performed following almost any closing parenthesis not at statement level. The severe inefficiencies most commonly follow typedefed pointers to functions:

```
typedef A (*B)(C);
```

for which (C) is a valid generalised first parameter, and it is only after parsing many subsequent declarations, which may include complete class definitions, that the missing function-body is eventually detected.

Initiating a lookahead search after every close parenthesis interacts very badly with the initiation of binary tree searches to resolve template ambiguities. It is advisable to constrain the generality by maintaining a type I enabled flag and type I active flag so that type I lookahead is initiated less often and so that incompatible syntaxes (such as templates) terminate the lookahead more rapidly.

F.2.5 Semantic Ambiguities

The traditional parsing approach needs to resolve semantics during syntactic analysis and so encounters ambiguities that need type information. These ambiguities are all deferred until the semantic analysis by the superset parse.

F.2.5.1 Declaration / Expression ambiguity (§6.8)

Section 4.5.3.2 discussed the ambiguity whereby

```
T(a);
```

could be:

- an *expression-statement* invoking the function or constructor T with argument a
- a *declaration* of a variable of type T and redundantly parenthesised name a

Resolution

The superset grammar eliminates the ambiguity but requires semantic processing to resolve type-dependent problems identified in Section F.2.1.2 and Section F.2.6.

[The multi-pass FOG parser resolved the ambiguity by parsing declarations in the first pass and expressions in the second pass. Statements within functions were not parsed and so no semantic corrections were required.]

F.2.5.2 Parenthesised-call / cast-parenthesis, parenthesised-binary / cast-unary ambiguity

A full semantic interpretation of a C++ program obviously requires a knowledge of the types. Unfortunately this information is also needed for a correct syntactic interpretation of an expression using a C cast followed by a unary operator or call.

```
(T)-5              // This is a cast if T is a type
(t)-5              // This is a subtraction if t is not a type

(T)(5)             // This is a cast if T is a type
(t)(5)             // This is a function call if t is not a type
```

Resolution for binary operator

Without type information, the above cases cannot be distinguished. At most one of the two possibilities can be parsed correctly. The misparse must be detected later and corrected. The superset grammar misparses the unary operator as the binary operator, since the subsequent semantic correction to change a binary operator into a cast is simpler than changing a unary into a binary:

- The change from binary requires replacing the binary node by a cast node and inserting a unary operator on the leading child node. This involves only the erroneous node and its children.
- A change from unary to cast would require inserting a cast node in the parent hierarchy of the erroneous node, and parent traversal is not normally supported by tree algorithms.

Preferring the binary operation is probably slightly more efficient. The use of C casts is discouraged in C++ and so the need to make a correction to a cast should be rare.

The misparse resulting from the lack of type information is tractable because the incorrect parse results in a small easily resolved error in the parse tree.

Resolution for parenthesis

When a possible cast is followed by a parenthesised expression, the resolution has to be in the opposite direction, favouring the cast, since the presumption of a function-call would preclude the possibility of a subsequent non-parenthesised term:

```
(a)(b)(c)(d)e(f)(g)(h);
```

[The multi-pass implementation of FOG used back-tracking to resolve ambiguities. The cast ambiguity was resolved by establishing a mark following any open parenthesis and then attempting to parse a cast. If the cast failed a nested expression was parsed. This use of back-tracking was inefficient, required a syntax rearrangement to isolate the leading parenthesis and resolved the ambiguity in the opposite direction. Since expressions were only used within the context of declarations, inadequacies were not significant.]

F.2.5.3 Call / functional-cast ambiguity

As described in Section 4.7.1.4, the functional-cast is totally subsumed by a call and so has been eliminated from the superset grammar.

F.2.5.4 Destructor name / one’s complement ambiguity (§5.3.1-9)

An unqualified destructor name cannot appear in an expression because of the ambiguity with a complement operator.

```
    ~ X ();    // This is ~(X()) even if X is a class
```

Unqualified destructors are therefore omitted from the superset parse.

Resolution

The missing unqualified destructor name in a declaration must be recovered from its complement expression form. The semantic processing must check all complement nodes to detect the misparse.

F.2.5.5 *new-placement* / *new-initializer* ambiguity

Type information is needed to distinguish between an omitted *new-placement* and an omitted *new-initializer* when

```
    new (a)(b)
```

is parsed against

new-expression:
    `::`$_{opt}$ `new` *new-placement*$_{opt}$ *new-type-id* *new-initializer*$_{opt}$
    `::`$_{opt}$ `new` *new-placement*$_{opt}$ `(` *type-id* `)` *new-initializer*$_{opt}$

new-placement:
    `(` *expression-list* `)`

new-initializer:
    `(` *expression-list*$_{opt}$ `)`

Resolution

The parser creates an AST node with two child expressions for the ambiguous case. Semantic processing determines that if the first child is a type, then the *new-placement* has been omitted, or alternately if non-type that the *new-initializer* defaults.

[The multi-pass implementation of FOG parsed for *new-placement* present on a first pass and absent on a second.]

F.2.5.6 `sizeof` ambiguity

Type information is required to distinguish the overlap between the `sizeof` a value and of a type. The syntactic ambiguity resulting from the lack of type information may be resolved by recognising that the generalised syntax accepts a parenthesised *parameter-declaration* as a *unary-expression*. The syntax for parenthesised *type-id* is therefore covered by the syntax accepted as a *unary-expression*.

unary-expression:
    ...
    `sizeof` *unary-expression*
    ~~`sizeof (` *type-id* `)`~~
    ...

Resolution

The parser creates an AST node with a child expression. Semantic processing determines which syntax is in use.

[The multi-pass implementation of FOG used two passes to parse the two alternatives without changing the grammar.]

F.2.5.7 `typeid` ambiguity

Type information is required to distinguish the overlap between the `typeid` of a value and that of a type. In the absence of type information, the superset parse of *expression* and *type-id* is available as a *parameter-declaration-clause*.

postfix-expression:
  ...
  ~~`typeid` (*expression*)~~
  ~~`typeid` (*type-id*)~~
  `typeid` (*parameter-declaration-clause*)
  ...

Resolution

The parser creates an AST node with a child expression. Semantic processing determines which syntax is in use.

[The multi-pass implementation of FOG used two passes to parse the two alternatives without changing the grammar.]

F.2.5.8 Template argument type / value ambiguity

The appropriate template specialisation cannot be selected without knowledge of inheritance.

Successful parsing of the arguments of a template require type information to distinguish type and value parameters.

Resolution

Templates do not need to be specialised during the primary parse. An AST node describing the *template-argument-list* is created which the subsequent semantic processing elaborates.

A superset grammar that covers

template-argument:
  assignment-expression
  type-id
  template-id

can be used to create AST nodes during the syntax analysis that can be interpreted later once type information is available. The generalised *parameter-declaration* covers *template-argument*.

[The multi-pass implementation of FOG used two passes to attempt value and type parsing. Value and type were tried independently for each argument of a multi-argument template resulting in exponential complexity.]

F.2.6 New C++ ambiguities

A strict superset of the declaration and expression syntaxes should introduce no new ambiguities. However parsing is eased by taking a rather larger superset. In particular a very general policy is adopted for names. As a result some new ambiguities are created.

F.2.6.1 *ctor-initializer* or named bit-field

The generalised name makes

```
name
```

valid as the *declarator* in a *function-definition* with the result that there is deep ambiguity between

```
name : m(i), n(j), p(l) {}   // A constructor
name : m(i), n(j), p(l);     // Anon bit-field, and variables
```

Resolution

The superset grammar uses a shared prefix for the two cases, so that no decision is made until the disambiguating punctuation is reached. Use of a shared prefix requires considerable generalisation, tolerating *assignment-expression*s as each *ctor-initializer*, and an *assignment-expression* rather than an *identifier* as the bit-field name.

All valid syntaxes are parsed correctly. Many invalid syntaxes are accepted and need diagnosis at the semantic level.

The shared prefix appears as `constructor_head` in the grammar of Appendix B. The prefix causes greater problems in the implementation of `tree_statement`, since, as described at the end of Section 2.1.1.6, there is an ambiguity between the use of a comma to separate multiple components of a single element, and its use to separate multiple elements. The implementation in Appendix C is carefully structured to avoid shift-reduce conflicts.

F.2.6.2 *class-specifier* or *enum-specifier* as *function-definition* name

The generalised name makes

```
enum X
```

valid as the *declarator* in a *function-definition* with the result that

```
enum X {};
```

is accepted as both an *enum-specifier* and *function-definition*.

Resolution

The resolution of the bit-field or inheritance ambiguity (Appendix F.2.4.1) has the beneficial side effect of solving this ambiguity for

```
class X {};
```

The ambiguity for `enum` produces a shift-reduce conflict, which is resolved in favour of the only valid possibility: the *enum-specifier*. No semantic repair is necessary.

Elaboration of the grammar to be more restrictive on function names might be possible for C++, although there is a risk of introducing conflicts between *function-definition*s and *simple-declaration*s. Such elaboration is not possible in FOG where the extended *re-using-declaration* syntax accepts function names without parentheses.

F.2.6.3 `delete[]` ambiguity

Introduction of the abstract array declarator `[]` as part of a *primary-expression* creates an ambiguity in

```
delete[](p)
```

between

- the intended array operation
- deletion of an abstract array of functions.

Resolution

The array form is removed from the grammar and supported by generalising a cast expression to accept a bracketed as well as a parenthesised prefix. The resulting bracketed cast ambiguities are resolved in exactly the same way as parenthesised cast ambiguities, save for the benefit that the resolution is correct except for the array delete which must be detected semantically.

```
cast_expression:      unary_expression
    |                 abstract_expression cast_expression
abstract_expression:  parenthesis_clause
    |                 '[' expression.opt ']'
parenthesis_clause:   parameters_clause cv_qualifier_seq.opt
                                  exception_specification.opt
```

Generalisation of the cast also covers the gcc indexed array initializer extension described in Section 2.1.4.2.

F.2.6.4 *linkage-specification* ambiguity

The generalised name parsing supports interpretation of

```
    extern "C";
```

as a conventional declaration since a *string-literal* satisfies the syntax of a generalised name.

Resolution

This is a false ambiguity resolved by unconditional treatment of the keyword `extern` followed by a *string-literal* as the pre-amble for a *linkage-specification*.

F.2.7 Extra FOG ambiguities

The presentation of the FOG extensions in Chapter 2 identified many of the ambiguities associated with the extensions and parsing approaches to avoid them. All ambiguities that cannot easily be avoided are identified in this section.

F.2.7.1 *built-in-type-id* maximised

The lack of distinction between meta-type-names and meta-names leads to two ambiguities when a meta-name follows a meta-type as in a *meta-variable-declaration*.

```
    auto short long :: int :: a = 0;
```

This could be

- a meta-declaration of `::a` with meta-type `short long::int`
- a meta-declaration of `::int::a` with meta-type `short long`
- a meta-declaration of `long::int::a` with meta-type `short`
- a meta-expression assigning to `short long::int::a`

Resolution

These ambiguities are resolved by two of the C++ disambiguation principles: maximise to the left, and prefer declarations to expressions.

The elemental meta-type-name is therefore maximised syntactically as `short long` even though this is a semantic error. The entire name is maximised as `short long::int::a` to select the expression interpretation.

F.2.7.2 Label preferred to anonymous bit-field

There is an ambiguity between a label and bit-field:

```
    label: int(value);       // label then simple-declaration
    type : int(value);       // anonymous bit field
```

which does not arise in C++ since labels and bit-fields are syntactically exclusive. Syntax generalisation in FOG removes this exclusivity introducing a parsing conflict.

Resolution

The conflict is resolved to preserve C++ semantics, by preferring the label interpretation as part of a statement. The same `%prec` that forces an identifier < to be shifted for a template test also forces a shift to prefer a *labeled-statement* whenever a *labeled-statement* is a syntactic option.

```
id:                  identifier                %prec SHIFT_THERE
                     ...
labeled_statement:   identifier ':' looping_statement
                     ...
```

F.2.7.3 *handler-seq* maximised

The generic syntax parsing for a tree statement supports a *handler-seq* following a *try-block*.

```
    void f() try {} catch (a) {} catch (b) {} catch (c) {}
```

leading to an ambiguity as to where the function ends and a subsequent *handler-seq* begins.

Resolution

The ambiguity is resolved by definition to maximise the length of the left hand syntax element.

```
handler_seq:         handler                   %prec SHIFT_THERE
    |                handler handler_seq
```

F.2.7.4 *access-specifier*

The false ambiguity resulting from the generalised name interpretation of

```
    public : x(y);
```

as an implicit `int` anonymous bit-field was discussed in Section 2.1.3.2.

Resolution

The ambiguity is resolved by a `%prec` to favour the *accessibility-specifier*.

```
decl_specifier_affix:            ...
    |                    access_specifier      %prec SHIFT_THERE
accessibility_specifier: access_specifier ':'
```

F.2.7.5 `inline/` and `virtual/` cannot be expressions

The false ambiguities resulting from the generalised name parsing of

```
    name1 inline / interface ( name2 ) ;
```

were discussed in Sections 2.1.3.5 and 2.1.3.6.

Resolution

The ambiguities are resolved by %precs to favour the switch.

```
function_specifier:   "explicit"
    |                 "inline"                 %prec SHIFT_THERE
    |                 "virtual"                %prec SHIFT_THERE
    |                 '!' "inline"
    |                 "inline" '/' "implementation"
    |                 "inline" '/' "interface"
    |                 '!' "virtual"
    |                 "virtual" '/' "pure"
```

F.2.7.6 using *string-literal* is not an expression

A false ambiguity arises from the generalised name interpretation of

```
    using "string";
```

as a *re-using-declaration* since a *string-literal* satisfies the generalised syntax of a name and a *re-using-declaration* is parsed as a generalised declaration or expression.

Resolution

The ambiguity is resolved by treating the keyword using followed by a *string-literal* as an *include-declaration* unconditionally.

```
decl_specifier_affix:    ...
    |                    "using"               %prec SHIFT_THERE
include_declaration:     "using" string
                         ...
```

F.3 Semantic checks

The syntactic analysis tolerates a very generalised syntax that merges declarations and expressions and requires no knowledge of template or type names. The analysis builds an Abstract Syntax Tree that must then be processed to incorporate semantic information.

The semantic processing comprises four parts; resolution of syntax ambiguities, correction of misparses, validation of semantics and implementation. Each part is naturally performed in a distinct pass over the AST adding little to the substantial amount of processing needed for a complex language such as C++.

The origin of most of the deferred ambiguities has been described in Appendices F.2.5, F.2.6 and F.2.7. Their resolution during semantic analysis is considerably eased by operation on the AST where the whole of the construct to be analysed is available, whereas the more conventional approach must struggle with constraints of parser lookahead and shift-reduce conflicts. Both approaches ultimately require the same decision code.

F.3.1 Resolution

A conventional C++ parse uses semantic information to resolve most if not all ambiguities during the syntax parse. The superset parse uses a generalised syntax to avoid semantic leakage. As a result the parse is incomplete and requires ambiguities deferred from the syntactic analysis to be resolved during semantic analysis. The most significant ambiguity is the declaration / expression ambiguity, but there are a number of other minor type related problems to resolve.

F.3.1.1 Declaration / Expression (§6.8)

The usage of the same AST nodes for declarations and expressions must be resolved. The superset grammar parses declarations as a slightly generalised expression, and so when appropriate, the declaration must be determined from an expression tree. The context in which a tree is used sometimes determines whether a declaration or expression is required. For instance, an initializer for a default argument can only be an expression. In most situations, such as a statement within a function, an ambiguity must be resolved.

Some aspects of the ambiguity can and must be resolved by an accurate syntax check. For instance `* const` can only occur in a declaration, whereas && can only occur in an expression. Further aspects can be resolved when type information is available as it is for actual declarations. The syntax resolution for an actual declaration can therefore perform a strong check, whereas only a weak check can be performed for a potential declaration.

Resolution occurs by a dataflow propagation of a set of boolean flags that indicate whether the AST satisfies a number of syntactic hypotheses. These hypotheses are propagated from the leaves to the root, applying the constraints applicable at each node so that the syntaxes satisfied by the tree can be determined.

The current partially functional implementation propagates a bit vector of 76 hypotheses ranging from EPSILON, IDENTIFIER, CLASS_NAME, ELABORATED_TYPE_SPECIFIER, via PARAMETER_DECLARATOR and PARAMETER_DECLARATION_CLAUSE, to BIT_FIELD_DECLARATION or META_FUNCTION_DECLARATION.

Maintenance of these hypotheses at the 20 or so interesting tree nodes is relatively straightforward since few hypotheses propagate through each node type. Most operator nodes propagate only the BASIC_EXPRESSION hypothesis. Unary operators such as & * and ~ that have meaning in declarators also propagate the various DECLARATOR hypotheses. Tree nodes for the binary operators * and & resolve their alternate parse for a left-hand type and right-hand declarator. Tree nodes involving parentheses are where the conventional ambiguities are resolved and where there is some implementation complexity, but only in the number of ifs that map incoming to outgoing hypotheses.

The overall operation of the tree nodes of course restores the declaration grammar which was folded into the expression grammar; there are separate hypotheses for DECLARATOR and DIRECT_DECLARATOR. However, because of the very different semantic constraints associated with different names, it is necessary to maintain independent hypotheses for ABSTRACT, CONVERSION, FUNCTION, PURE_FUNCTION, INIT, META, PARAMETER or BIT_FIELD DECLARATIONs and DECLARATORs and sometimes DIRECT_DECLARATORs, DECLARATION_IDs and DECLARATOR_IDs. It is not appropriate to merge all DECLARATOR hypotheses as a single parameterised hypothesis, since each hypothesis is potentially independent and propagated up the tree in parallel. Sharing state between hypotheses could cause cross-talk and therefore fail to achieve sufficient precision to resolve very finely balanced declaration / expression ambiguities correctly.

The same propagation algorithm is used for weak and strong hypotheses. The difference lies at the leaves. For a strong determination, an identifier node is assessed to determine whether the identifier satisfies each of a class, enum namespace, typedef and template name hypothesis. For a weak determination, all name hypotheses are satisfied at an identifier node, since there is no context to contradict the hypothesis.

All hypotheses are propagated in parallel, and so a single pass over the tree identifies all satisfied syntaxes. Ambiguity resolution amongst those of interest determines whether to

- correct misparses in the expression meta-object
- create a potential declaration specifier meta-object

The corrections that may be required are outlined in Appendix F.3.2. Creation of a potential declaration specifier involves a further tree traversal towards the naming node where the appropriate meta-object can be created, and then decorated as it is returned through intervening nodes.

F.3.1.2 *new-placement* / *new-initializer* ambiguity

See Section F.2.5.5.

F.3.1.3 `sizeof` and `typeid` ambiguity

The reuse of the same AST nodes for types and values probably simplifies the implementation.

F.3.1.4 *pure-specifier*

The

pure-specifier:
        `= 0`

syntax is covered by

constant-initializer:
        = *constant-expression*

so that if even the lexer distinguishes the lexeme '0' from the number 0, there is little possibility of identifying the distinction in a generalised parse.

Resolution

The superset parse does not resolve *pure-specifier*s or even *constant-initializer*. The semantic pass must identify each from the more general *assignment-expression*, taking care to distinguish original character streams such as `00` or `0x0` from `0`.

F.3.1.5 *explicit-instantiation*

The generalised name binds `template` close to the name, rather than as a prefix to a *declaration*. Semantic processing must therefore locate the keyword deeper in the AST than might be expected. *explicit-instantiation* does not appear as a distinct production.

F.3.1.6 Implicit `int`

[Roskind91] questions the enthusiasm with which compilers have stopped supporting the deprecated implicit `int` for C functions with no declared return type.

Resolution

The purely syntactic superset parse does not distinguish implicit `int` from a constructor. Implicit `int` is therefore parsed successfully. The semantic pass may easily distinguish implicit `int` from a constructor, since a constructor has a name that matches its scope. Any other name is implicit `int`.

F.3.2 Correction

Misparses resulting from incorrect assumptions during the syntactic analysis must be corrected, or accounted for during subsequent processing.

F.3.2.1 Template / arithmetic ambiguity

The binary tree search to identify a consistent syntax for the template or arithmetic interpretation of identifier followed by < identifies the wrong syntax for approximately 0.01% of statements (Section 4.8.2). These errors must be corrected. Detection of the errors just requires the semantic test that should have been performed during lexical analysis to be performed upon the AST. If the test result is inconsistent with the tree structure, then the tree must be rearranged. This is not particularly easy, since the two interpretations have distinctly different tree structures, and rearrangements of nodes associated with arithmetic operators must account for arithmetic precedences. The complexity is reduced a little by the use of generic syntax elements, and so there is no difference between the sub-tree for an *expression* and that for a *template-argument*. There is no `template_argument` meta-type.

F.3.2.2 Parenthesised-call / cast-parenthesis, parenthesised-binary / cast-unary ambiguity

Semantic correction of the AST is required where a binary operator that is also a unary operator has a type as its first child. This indicates that the binary operator should be replaced by a C-style cast, and that the equivalent unary operator be applied to the first term in the tree headed by the second child.

The reverse correction occurs where the parenthesised function name in a function call was misparsed as a cast. An apparent cast to a non-type should be corrected to a parenthesised function call.

F.3.2.3 Call / functional-cast ambiguity

A function call invoking a non-class type is recognised as a functional-cast.

A function call invoking a class type is recognised as a constructor.

F.3.2.4 Destructor name / one's complement ambiguity (§5.3.1-9)

A function call to a complemented class-name in a declaration is detected as a destructor declaration.

F.3.2.5 Array of operator ambiguity

Arrays of `operator new` or `operator delete` are recognised as the array forms `operator new[]` and `operator delete[]`.

F.3.2.6 `delete[]` ambiguity

An `[]` prefix following a `delete` operator is recognised as denoting `delete[]`.

F.3.3 Validation

The syntax generalisations to establish a simple superset grammar accept many nonsensical constructs.

```
double root2 = static sqrt(2) const throw();// declaration clutter
void b extern = int a char;                 // unlimited generality
A::A() : this(0), 5+7, ~3, int(...) {}  // expression/name clutter
```

The form of each generalised expression and declaration must be carefully validated to diagnose the numerous anomalies that should normally be trapped by syntax analysis.

This has advantages in terms of error diagnosis. Because the syntax accepts so much, there is a good chance that many common semantic errors will survive the syntactic analysis, build a plausible tree, at which point the difference from a closely matching possibility can be reported. An accurate syntax parse is liable to encounter the ubiquitous unclassified “parse error” more often, or require extra elaboration in the grammar to cover probable user errors.

F.3.4 Implementation

The semantic analysis of the AST for each parsed statement must update the symbol table to keep track of the enlarged program and make any additional declarations accessible to subsequent statements.

For some unpleasant statements such as typedefs containing more than one declarator, or function parameter lists, the semantic analysis update must perform the symbol table updates so that names introduced early in the statement have appropriate visibility later in the same statement. Achieving this during syntactic analysis, and before expression / declaration ambiguities have been resolved, requires very careful coding of the symbol table update to reflect the tentative semantic analysis of an incomplete syntactic analysis.

Again these are exactly the same decisions that need to be performed by any accurate C++ compiler, however their implementation is eased by operation in the context of the AST rather than within the straitjacket imposed by avoiding conflicts while looking infinitely far ahead in the parser grammar.

INDEX