Pure Functional Programming
Functional Programming and Reasoning

Dr Hans Georg Schaathun

University of Surrey

Spring 2010
Outline

1. Testing and Proof
2. Pure Functional Programming
3. Definedness and Termination
4. Conclusion
This week we aim to answer two questions:
- How do we know that our program is correct?
- What’s so special about functional languages?

Students should be able to
- use testing to verify that functions and programs are correct
- use (simple) reasoning to verify that functions are correct
- understand when testing and reasoning (respectively) is appropriate
- understand the concept of side effects, including when they are useful and when they should be avoided
This week we aim to answer two questions:

- How do we know that our program is correct?
- What’s so special about functional languages?

Students should be able to:

- use testing to verify that functions and programs are correct
- use (simple) reasoning to verify that functions are correct
- understand when testing and reasoning (respectively) is appropriate
- understand the concept of side effects, including when they are useful and when they should be avoided
Outline

1. Testing and Proof
   - Verifying programs
   - A little logic
   - Testing and induction
   - Reason and deduction

2. Pure Functional Programming

3. Definedness and Termination

4. Conclusion
Outline

1. Testing and Proof
   - Verifying programs
     - A little logic
     - Testing and induction
     - Reason and deduction

2. Pure Functional Programming

3. Definedness and Termination

4. Conclusion
How can we know what a function (program) does?

1. Try it in an implementation (hugs)
2. Evaluate it by hand (pen and scrap paper)
3. Analyse it mathematically (proof and argument)
Understanding Programs

*How can we know what a function (program) does?*

1. **Try it** in an implementation (hugs)
2. **Evaluate it** by hand (pen and scrap paper)
3. **Analyse it** mathematically (proof and argument)
Testing
Selected test cases

- Has everyone used testing?
  - Select a number of test cases.
    - Specify input
    - Determine expected output
    - Run, and check if output is as expected
- Finite number of test cases
  - A test is not complete
Testing

Selected test cases

- Has everyone used testing?
- Select a number of test cases.
  - Specify input
  - Determine expected output
  - Run, and check if output is as expected
- Finite number of test cases
  - A test is not complete
Testing

Selected test cases

- Has everyone used testing?
- Select a number of test cases.
  - Specify input
  - Determine expected output
  - Run, and check if output is as expected
- Finite number of test cases
  - A test is not complete
Analysis and generalisation

- Can you logically prove that the program is correct?
- Has anyone ever tried it?
- Generalise. Aim at a proof valid for all cases.
- Not limited by a finite number of cases.
Can you logically prove that the program is correct?

Has anyone ever tried it?

Generalise. Aim at a proof valid for *all* cases.

Not limited by a finite number of cases.
Analysis and generalisation

- Can you logically prove that the program is correct?
- Has anyone ever tried it?
- Generalise. Aim at a proof valid for all cases.
- Not limited by a finite number of cases.
Hand Evaluation

- **When is hand evaluation useful?**
  - Development: trial and error
    - bug tracking
  - Small cases; easy to do by hand
  - Understand the algorithm
    - recognise patterns
    - help to generalise a definition (often recursive)
  - Like testing, limited to a small number of test cases
    - advantage: track the calculation to find errors
Hand Evaluation

- When is hand evaluation useful?
- Development: trial and error
  - bug tracking
- Small cases; easy to do by hand
- Understand the algorithm
  - recognise patterns
  - help to generalise a definition (often recursive)
- Like testing, limited to a small number of test cases
  - advantage: track the calculation to find errors
Hand Evaluation

- When is hand evaluation useful?
- Development: trial and error
  - bug tracking
- Small cases; easy to do by hand
  - Understand the algorithm
    - recognise patterns
    - help to generalise a definition (often recursive)
  - Like testing, limited to a small number of test cases
    - advantage: track the calculation to find errors
Hand Evaluation

- When is hand evaluation useful?
  - Development: trial and error
    - bug tracking
  - Small cases; easy to do by hand
  - Understand the algorithm
    - recognise patterns
    - help to generalise a definition (often recursive)

- Like testing, limited to a small number of test cases
  - advantage: track the calculation to find errors
Hand Evaluation

- When is hand evaluation useful?
- Development: trial and error
  - bug tracking
- Small cases; easy to do by hand
- Understand the algorithm
  - recognise patterns
  - help to generalise a definition (often recursive)
- Like testing, limited to a small number of test cases
  - advantage: track the calculation to find errors
Evaluate it
Why it is a good idea

- Testing tells you if there is a mistake
- Manual evaluation can tell you where there is an error

1. Do you apply the correct guard?
2. Is there a typo at an early stage?

- Of course, only appropriate for small examples
Evaluate it
Why it is a good idea

- Testing tells you if there is a mistake
- Manual evaluation can tell you where there is an error

1. Do you apply the correct guard?
2. Is there a typo at an early stage?

- Of course, only appropriate for small examples
Evaluate it
Why it is a good idea

- Testing tells you if there is a mistake
- Manual evaluation can tell you where there is an error

1. Do you apply the correct guard?
2. Is there a typo at an early stage?

- Of course, only appropriate for small examples
Evaluate it
For example

1. \texttt{factorial 0 = 1}
2. \texttt{factorial n = n*factorial n}

- \texttt{factorial 3}
  - = 3*factorial 2 (line 2)
  - = 3*2*factorial 1 (line 2)
  - = 6*1*factorial 0 (line 2)
  - = 6*1 (line 1)
  - = 6 (multiplication)
Evaluate it
For example

1. \textit{factorial} 0 = 1
2. \textit{factorial} n = n*\textit{factorial} n

\textit{factorial} 3
\begin{align*}
\text{\quad} &= 3*\textit{factorial} \ 2 \quad \text{(line 2)} \\
\text{\quad} &= 3*2*\textit{factorial} \ 1 \quad \text{(line 2)} \\
\text{\quad} &= 6*1*\textit{factorial} \ 0 \quad \text{(line 2)} \\
\text{\quad} &= 6*1 \quad \text{(line 1)} \\
\text{\quad} &= 6 \quad \text{(multiplication)}
\end{align*}
Break the problem into smaller pieces
  - separate functions for each subproblem

Verify each function separately
  - starting with the most fundamental ones
  - using testing or analysis as appropriate

If the test fails
  - you have little code to search for the bug
1. Testing and Proof
   - Verifying programs
   - A little logic
   - Testing and induction
   - Reason and deduction

2. Pure Functional Programming

3. Definedness and Termination

4. Conclusion
Consider a statement such as

1 Sheep are white
2 Therefore, Dolly the Sheep is white

This is an implication
- There is an assumption (1)
- and a consequence (2)

The validity of the assumption is uncertain

But if the assumption is true, there is no option for the consequence

Also note that if (2) is false then (1) must be false
Consider a statement such as

1. Sheep are white
2. Therefore, Dolly the Sheep is white

This is an *implication*

- There is an *assumption* (1)
- and a *consequence* (2)

The validity of the assumption is uncertain

But if the assumption is true, there is no option for the consequence

Also note that if (2) is false then (1) must be false
Consider a statement such as

1. Sheep are white
2. Therefore, Dolly the Sheep is white

This is an **implication**

- There is an *assumption* (1)
- and a *consequence* (2)

The validity of the assumption is uncertain

- But if the assumption is true, there is no option for the consequence
- Also note that if (2) is false then (1) must be false
Consider a statement such as

1. Sheep are white
2. Therefore, Dolly the Sheep is white

This is an implication

- There is an assumption (1)
- and a consequence (2)

The validity of the assumption is uncertain

But if the assumption is true, there is no option for the consequence

Also note that if (2) is false then (1) must be false
Consider a statement such as

1. Sheep are white
2. Therefore, Dolly the Sheep is white

This is an implication

- There is an assumption (1)
- and a consequence (2)

The validity of the assumption is uncertain

But if the assumption is true, there is no option for the consequence

Also note that if (2) is false then (1) must be false
Logical Notation

- Sheep are white $\Rightarrow$ Dolly the Sheep is white
- Dolly the Sheep is white $\Leftarrow$ Sheep are white
Logical Notation

- Sheep are white $\Rightarrow$ Dolly the Sheep is white
- Dolly the Sheep is white $\Leftarrow$ Sheep are white
Checking the Validity

- Two separate questions
  1. Is the assumption (1) true?
  2. Is the inference \((\Rightarrow)\) valid?

- The consequence (2) follows if both answers are yes
Deduction

- Sheep are white $\Rightarrow$ Dolly the Sheep is white
- Example of Deduction
  - i.e. a valid inference
- $W$ be the set of all white things
- $S$ be the set of all sheep
- $d \in S$ is Dolly the Sheep
- We have assumed $S \subset W$
- We have $d \in S \subset W$
  - Hence $d \in W$
Deduction

- Sheep are white $\Rightarrow$ Dolly the Sheep is white
- Example of Deduction
  - i.e. a valid inference
- $W$ be the set of all white things
- $S$ be the set of all sheep
- $d \in S$ is Dolly the Sheep
- We have assumed $S \subset W$
- We have $d \in S \subset W$
  - Hence $d \in W$
Deduction

- Sheep are white $\Rightarrow$ Dolly the Sheep is white
- Example of Deduction
  - i.e. a valid inference
- $W$ be the set of all white things
- $S$ be the set of all sheep
- $d \in S$ is Dolly the Sheep
- We have assumed $S \subset W$
- We have $d \in S \subset W$
  - Hence $d \in W$
Deduction

- Sheep are white $\Rightarrow$ Dolly the Sheep is white
- Example of Deduction
  - i.e. a valid inference
- $W$ be the set of all white things
- $S$ be the set of all sheep
- $d \in S$ is Dolly the Sheep
- We have assumed $S \subset W$
- We have $d \in S \subset W$
  - Hence $d \in W$
Deduction

- Sheep are white $\Rightarrow$ Dolly the Sheep is white
- Example of Deduction
  - i.e. a valid inference
- $W$ be the set of all white things
- $S$ be the set of all sheep
- $d \in S$ is Dolly the Sheep
- We have assumed $S \subset W$
  - We have $d \in S \subset W$
    - Hence $d \in W$
Deduction

- Sheep are white \( \Rightarrow \) Dolly the Sheep is white
- Example of Deduction
  - i.e. a valid inference
- \( W \) be the set of all white things
- \( S \) be the set of all sheep
- \( d \in S \) is Dolly the Sheep
- We have assumed \( S \subset W \)
- We have \( d \in S \subset W \)
  - Hence \( d \in W \)
Deduction

- Sheep are white $\Rightarrow$ Dolly the Sheep is white

Example of Deduction

- i.e. a valid inference

- $W$ be the set of all white things
- $S$ be the set of all sheep
- $d \in S$ is Dolly the Sheep
- We have assumed $S \subset W$
- We have $d \in S \subset W$
  - Hence $d \in W$
Induction
Generalisation for a special case

Dolly the Sheep is white ⇒ Sheep are white
- This would be an invalid inference
- (generalising from a special case)

Known as induction
Attempts to generalise from specific information
Induction

Generalisation for a special case

- Dolly the Sheep is white $\Rightarrow$ Sheep are white
  - This would be an invalid inference
  - (generalising from a special case)

Known as induction

- Attempts to generalise from specific information
Induction
Generalisation for a special case

- Dolly the Sheep is white $\Rightarrow$ Sheep are white
  - This would be an invalid inference
  - (generalising from a special case)
- Known as induction
- Attempts to generalise from specific information
Learning by Observation

Dolly is white
Learning by Observation

All (?) sheep are white

Not quite. **Many** sheep are white.
Learning by Observation

All (?) sheep are white

Not quite. Many sheep are white.
Learning by Observation
Not all sheep are white
 observable. A white sheep exists.

Not observable. All sheep are white.

Observable. A non-white sheep exists.

Observable. Not all sheep are white.
Learning by Observation

Conclusion

- **Observable.** A white sheep exists.
- **Not observable.** All sheep are white.
- **Observable.** A non-white sheep exists.
- **Observable.** Not all sheep are white.
Learning by Observation

Conclusion

- Observable. A white sheep exists.
- Not observable. All sheep are white.
- Observable. A non-white sheep exists.
- Observable. Not all sheep are white.
Learning by Observation

Conclusion

- Observable. A white sheep exists.
- Not observable. All sheep are white.
- Observable. A non-white sheep exists.
- Observable. Not all sheep are white.
Outline

1. Testing and Proof
   - Verifying programs
   - A little logic
   - Testing and induction
   - Reason and deduction

2. Pure Functional Programming

3. Definedness and Termination

4. Conclusion
Testing does special cases only

1. Program works for input dolly
2. Program works for any input (of type Sheep)

- The former is observable – by testing
- The latter cannot be tested
- And the induction \(1 \rightarrow 2\) is not valid
Choosing test cases

Black Box Testing

- Testing cannot prove that the program works
  - but it can prove that it does not
  - find the black sheep in the flock

- Choosing test cases is a key skill
- appropriate broad selection
  - maximises the chances of finding a black sheep

- Black Box Testing
  - choose test cases based on the specification
  - with no knowledge of the implementation
Choosing test cases

Black Box Testing

- Testing cannot prove that the program works
  - but it can prove that it does not
  - find the black sheep in the flock

- Choosing test cases is a key skill
- appropriate broad selection
  - maximises the chances of finding a black sheep
- Black Box Testing
  - choose test cases based on the specification
  - with no knowledge of the implementation
Choosing test cases

Black Box Testing

- Testing cannot prove that the program works
  - but it can prove that it does not
  - find the black sheep in the flock

- Choosing test cases is a key skill

- appropriate broad selection
  - maximises the chances of finding a black sheep

- Black Box Testing
  - choose test cases based on the specification
  - with no knowledge of the implementation
Choosing test cases

Black Box Testing

- Testing cannot prove that the program works
  - but it can prove that it does not
  - find the black sheep in the flock
- Choosing test cases is a key skill
- appropriate broad selection
  - maximises the chances of finding a black sheep
- Black Box Testing
  - choose test cases based on the specification
  - with no knowledge of the implementation
Example: \texttt{minThree}

- \texttt{minThree :: Int -> Int -> Int -> Int}
- Returns the smallest of three integers
- Which inputs do we have to test?
  - Three different numbers
    - (small,middle,large); (small,large,middle); (large,middle,small); etc.
    - Two equal + one smaller
    - Two equal + one larger
    - Three equal
  - At least one test case from each category
    - not so important to test both \((1,2,3)\) and \((1,2,4)\)
Example: `minThree`

- `minThree :: Int -> Int -> Int -> Int`
- Returns the smallest of three integers
- Which inputs do we have to test?
  - Three different numbers
    - (small, middle, large); (small, large, middle); (large, middle, small); etc.
    - Two equal + one smaller
    - Two equal + one larger
    - Three equal
  - At least one test case from each category
    - not so important to test both (1, 2, 3) and (1, 2, 4)
Example: \texttt{minThree}

\texttt{minThree} :: \texttt{Int} \to \texttt{Int} \to \texttt{Int} \to \texttt{Int}

Returns the smallest of three integers

Which inputs do we have to test?

- Three different numbers
  - (small, middle, large); (small, large, middle); (large, middle, small); etc.
  - Two equal + one smaller
  - Two equal + one larger
  - Three equal

- At least one test case from each category
  - not so important to test both (1, 2, 3) and (1, 2, 4)
Outline

1. Testing and Proof
   - Verifying programs
   - A little logic
   - Testing and induction
   - Reason and deduction

2. Pure Functional Programming

3. Definedness and Termination

4. Conclusion
Analyse it

Go general

- Testing and evaluation consider **special cases**
  - Analytical argument aim to generalise

- Good argument is about
  - being structured and systematic
  - covering all cases

- This is about practice and talent
  - so let’s go through an example
Analyse it
Go general

- Testing and evaluation consider **special cases**
  - Analytical argument aim to generalise
- Good argument is about
  - being structured and systematic
  - covering all cases
- This is about practice and talent
  - so let’s go through an example
Analyse it
Go general

- Testing and evaluation consider **special cases**
  - Analytical argument aim to generalise

- Good argument is about
  - being structured and systematic
  - covering all cases

- This is about practice and talent
  - so let’s go through an example
Is the following a correct calculation of maximum?

```
minThree x y z
  | x < y && x < z   = x
  | y < x && y < z   = y
  | otherwise       = z
```
Analyse it
Continued

1  |  $x < y \land x < z = x$
   |  Clearly, $x$ is strictly smaller than $y$ and $z$. Correct.

2  |  $y < x \land y < z = y$
   |  Clearly, $x$ is strictly smaller than $x$ and $z$. Correct.

Which cases remain for otherwise?
Analyse it
Continued

1. \( x < y \land x < z = x \)
   Clearly, \( x \) is strictly smaller than \( y \) and \( z \). Correct.

2. \( y < x \land y < z = y \)
   Clearly, \( x \) is strictly smaller than \( x \) and \( z \). Correct.

Which cases remain for otherwise?
Analyse it
Continued

1. $x < y \land x < z = x$
   - Clearly, $x$ is strictly smaller than $y$ and $z$. Correct.

2. $y < x \land y < z = y$
   - Clearly, $x$ is strictly smaller than $x$ and $z$. Correct.

- Which cases remain for otherwise?
Analyse it

The otherwise cases

1. We may have $z < x$ and $z < y$, when $z$ is the correct output.

2. We may have $x = y = z$, when $z$ is correct (as would $x$ or $y$ be)

3. Cases with two equal inputs
   - ... next slide ...
Analyse it
The otherwise cases

1. We may have $z < x$ and $z < y$,
   - when $z$ is the correct output.
2. We may have $x = y = z$,
   - when $z$ is correct (as would $x$ or $y$ be)
3. Cases with two equal inputs
   - ... next slide ...
Analyse it

The otherwise cases

1. We may have \( z < x \) and \( z < y \),
   - when \( z \) is the correct output.

2. We may have \( x = y = z \),
   - when \( z \) is correct (as would \( x \) or \( y \) be).

3. Cases with two equal inputs
   - ... next slide ...
Analyse it

Two equal inputs

- $z = x$
  - $z \leq y$: otherwise applies – correct
  - $z > y$: Case 2 applies – correct

- $z = y$
  - $z \leq x$: otherwise applies – correct
  - $z > x$: Case 1 applies – correct

- $x = y$
  - $z \leq x$: otherwise applies – correct
  - $z > x$: otherwise applies – minimum is $x (=y)$
Analyse it

Two equal inputs

- \( z = x \)
  - \( z \leq y \): otherwise applies – correct
  - \( z > y \): Case 2 applies – correct

- \( z = y \)
  - \( z \leq x \): otherwise applies – correct
  - \( z > x \): Case 1 applies – correct

- \( x = y \)
  - \( z \leq x \): otherwise applies – correct
  - \( z > x \): otherwise applies – minimum is \( x (=y) \)
Analyse it
Two equal inputs

- $z = x$
  - $z \leq y$: otherwise applies – correct
  - $z > y$: Case 2 applies – correct

- $z = y$
  - $z \leq x$: otherwise applies – correct
  - $z > x$: Case 1 applies – correct

- $x = y$
  - $z \leq x$: otherwise applies – correct
  - $z > x$: otherwise applies – minimum is $x (=y)$
Analysis

Summary

- We found the bug
  - by carefully checking every case, class by class
  - the classes cover all possible inputs

- Suppose there were no bug
  - would this method allow us to prove correctness?

- Yes – we have not only found the bug
  - we know exactly which inputs for which the output is wrong

- For each class of inputs, it is either correct or wrong
  - and any possible inputs falls in at least one class
We found the bug
- by carefully checking every case, class by class
- the classes cover all possible inputs

Suppose there were no bug
- would this method allow us to prove correctness?

Yes – we have not only found the bug
- we know exactly which inputs for which the output is wrong

For each class of inputs, it is either correct or wrong
- and any possible inputs falls in at least one class
Analysis

Summary

- We found the bug
  - by carefully checking every case, class by class
  - the classes cover all possible inputs

- Suppose there were no bug
  - would this method allow us to prove correctness?

- Yes – we have not only found the bug
  - we know exactly which inputs for which the output is wrong

- For each class of inputs, it is either correct or wrong
  - and any possible inputs falls in at least one class
Analysis

Summary

- **We found the bug**
  - by carefully checking every case, class by class
  - the classes cover all possible inputs

- **Suppose there were no bug**
  - would this method allow us to **prove** correctness?

- **Yes – we have not only found the bug**
  - we know exactly which inputs for which the output is wrong

- **For each class of inputs, it is either correct or wrong**
  - and any possible inputs falls in at least one class
The bug fix

How do you correct the bug in the code?

(one way) replace < by <= in the guards

Exercise:
  - run through the analysis above for the corrected code
  - are you convinced that the implementation is correct for any input?
The bug fix

How do you correct the bug in the code?

(one way) replace $<$ by $\leq$ in the guards

Exercise:
- run through the analysis above for the corrected code
- are you convinced that the implementation is correct for any input?
The bug fix

How do you correct the bug in the code?

(one way) replace < by <= in the guards

Exercise:

- run through the analysis above for the corrected code
- are you convinced that the implementation is correct for any input?
White Box Testing

- Black Box Testing
  - consider the implementation as a black box
  - no knowledge of the implementation

- White Box Testing
  - analyse the implementation to choose test cases
  - In particular, look at the *guards*
    - when evaluation branches, use test sets for each branch
  - This often helps adding important test cases otherwise forgotten
White Box Testing

- **Black Box Testing**
  - consider the implementation as a black box
  - no knowledge of the implementation

- **White Box Testing**
  - analyse the implementation to choose test cases
    - In particular, look at the *guards*
    - when evaluation branches, use test sets for each branch
  - This often helps adding important test cases otherwise forgotten
White Box Testing

- **Black Box Testing**
  - consider the implementation as a black box
  - no knowledge of the implementation

- **White Box Testing**
  - analyse the implementation to choose test cases

- In particular, look at the *guards*
  - when evaluation branches, use test sets for each branch

- This often helps adding important test cases otherwise forgotten
White Box Testing

- Black Box Testing
  - consider the implementation as a black box
  - no knowledge of the implementation
- White Box Testing
  - analyse the implementation to choose test cases
  - In particular, look at the *guards*
    - when evaluation branches, use test sets for each branch
  - This often helps adding important test cases otherwise forgotten
Outline

1 Testing and Proof

2 Pure Functional Programming

3 Definedness and Termination

4 Conclusion
Haskell is a pure functional language
  therefore, lends itself well to reasoning
It adheres to a strict set of principles
  contrary to languages providing imperative and functional features together
What is a program

- Functional: list of definitions
  - one expression being evaluated at the end
- Imperative: list of instructions
  - executed one by one
  - modifying variables (system state)
- Time is important in imperative programming
  - instructions are executed in order
  - and instructions can redefine objects
- Definitions are constant in functional programming
  - nothing changes over time
Referential transparency

- All calls to the same function
  - same input $\Rightarrow$ same output

- Two major benefits
  - the compiler can optimise (avoid recomputation)
  - the developer can make mathematical proofs

- This is not true for imperative programming
  - as functions may depend on the system state (variables)
  - such dependencies may be hard to spot, and makes reasoning more difficult
Referential transparency

- All calls to the same function
  - same input $\Rightarrow$ same output

- Two major benefits
  - the compiler can optimise (avoid recomputation)
  - the developer can make mathematical proofs

- This is not true for imperative programming
  - as functions may depend on the system state (variables)
  - such dependencies may be hard to spot, and makes reasoning more difficult
Referential transparency

- All calls to the same function
  - same input $\Rightarrow$ same output

- Two major benefits
  - the compiler can optimise (avoid recomputation)
  - the developer can make mathematical proofs

- This is not true for imperative programming
  - as functions may depend on the system state (variables)
  - such dependencies may be hard to spot, and makes reasoning more difficult
Side-effects

- Functions have no side-effect
  - all communication is via the output
  - cannot modify behaviour of other functions
  - cannot do Input/Output (IO) directly
  - cannot modify variables

- Pure functional languages forbid side-effects completely
- Avoiding side-effects is good practice
  - easier to reuse code
  - other applications often need
    - same return value
    - different side-effects (output)

- e.g. computational back-end without IO can be combined with different user interfaces
Side-effects

- Functions have no side-effect
  - all communication is via the output
  - cannot modify behaviour of other functions
  - cannot do Input/Output (IO) directly
  - cannot modify variables

- Pure functional languages forbid side-effects completely

- Avoiding side-effects is good practice
  - easier to reuse code
  - other applications often need
    - same return value
    - different side-effects (output)

- e.g. computational back-end without IO can be combined with different user interfaces
Side-effects

- Functions have no side-effect
  - all communication is via the output
  - cannot modify behaviour of other functions
  - cannot do Input/Output (IO) directly
  - cannot modify variables

- Pure functional languages forbid side-effects completely

- Avoiding side-effects is good practice
  - easier to reuse code
  - other applications often need
    - same return value
    - different side-effects (output)

- e.g. computational back-end without IO can be combined with different user interfaces
Side-effects

- Functions have no side-effect
  - all communication is via the output
  - cannot modify behaviour of other functions
  - cannot do Input/Output (IO) directly
  - cannot modify variables

- Pure functional languages forbid side-effects completely

- Avoiding side-effects is good practice
  - easier to reuse code
  - other applications often need
    - same return value
    - different side-effects (output)

- e.g. computational back-end without IO can be combined with different user interfaces
Side-effects

An example

```python
function PseudoRandom()
    state := state * 17822228932 + 178324742
    return state mod 256
```

- You want a new pseudo-random number every time
- Side-effects is useful in this case
Side-effects

An example

function PseudoRandom()
    state := state * 17822228932 + 178324742
    return state mod 256

- You want a new pseudo-random number every time
- Side-effects is useful in this case
Other side-effects

- The most common side-effect is output
- This is often a bad side-effect
- You don’t always want the same output
- Some users will need the calculations without output
- Separation is useful
Imperative programming
including Java

- Global state (memory)
  - methods (functions) can change the state
  - method behaviour can depend on state

- Different calls to \( f \) can give different output
  - typical example: `getchar` (read keyboard input)
  - one call removes a character from input buffer
    - changes the value of the next call

- Functional *principles* can be used in imperative languages
  - avoid side-effects when they are not necessary
  - separate computations and IO
Imperative versus Functional

- Imperative programming:
  - list of actions to be performed by the computer

- Functional programming:
  - list of mathematical definitions and declarations
Termination

Definition

Termination means that a function completes and returns a value.

- Termination is a desired property
- It does happen that a function never terminates
**Example**

**Termination**

fac \(\cdot\) \(\cdot\)

- \textbf{fac 2} terminates (and returns 2)
- \textbf{fac \((-2)\)} does not terminate
  - \textbf{fac \((-2)\) = \((-2)\) \cdot \((-3)\) \cdot \((-4)\) \ldots}
  - The evaluation continues to expand the expression infinitely
Undefined expression

- Function calls which do not terminate (like `fac (-2)`) lead to **undefined** expressions.
- Undefined expressions lead to other undefined expressions.
  - We know that $0 \cdot a = 0$ for any number $a$.
  - ...what about $0 \cdot fac (-2)$?
- When `fac (-2)` is undefined, then $0 \cdot fac (-2)$ cannot be defined.
Definedness and Termination

Undefined expression

- Function calls which do not terminate (like `fac (-2)`)
  - lead to undefined expressions
- Undefined expressions lead to other undefined expressions
  - We know that \( 0 \cdot a = 0 \) for any number \( a \)
  - ...what about \( 0 \times \text{fac} (-2) \)?
- When `fac (-2)` is undefined,
  - then \( 0 \times \text{fac} (-2) \) cannot be defined.
Free variables and quantifiers

- $\text{square } x = x \times x$
- $x$ here is called a free variable.
- For which values of $x$ is the definition valid?
- We normally mean for every $x$.
- Explicitely: $\forall x, \text{square } x = x \times x$
- Or restrict it: $\forall x \in \{0, 1, 2, 3, \ldots\}$, $\text{square } x = x \times x$
Free variables and quantifiers

- \texttt{square \ x = x*x}
- \(x\) here is called a \textit{free variable}
- For which values of \(x\) is the definition valid?
- We normally mean \textit{for every} \(x\)
- Explicitely: \(\forall x, \text{square } x = x*x\)
- Or restrict it: \(\forall x \in \{0, 1, 2, 3, \ldots\}, \text{square } x = x*x\)
Definedness and Termination

Free variables and quantifiers

- \( \text{square } x = x \times x \)
- \( x \) here is called a \textit{free variable}
- For which values of \( x \) is the definition valid?
- We normally mean \textit{for every} \( x \)
- Explicitely: \( \forall x, \text{square } x = x \times x \)
- Or restrict it: \( \forall x \in \{0, 1, 2, 3, \ldots\}, \text{square } x = x \times x \)
Free variables and quantifiers

- **square** \( x = x \times x \)
- \( x \) here is called a *free variable*
- For which values of \( x \) is the definition valid?
- We normally mean *for every* \( x \)
- Explicitely: \( \forall x, \text{square } x = x \times x \)
- Or restrict it: \( \forall x \in \{0, 1, 2, 3, \ldots\}, \text{square } x = x \times x \)
Error handling

- Use the `error` statement for error handling

  - `error` interrupts evaluation
    - very odd behaviour for a function

- For a more advanced and functional error handling
  - see Thompson Chapter 14.4
Outline

1. Testing and Proof
2. Pure Functional Programming
3. Definedness and Termination
4. Conclusion
You should be familiar with every method of validating code
- testing
- manual evaluation
- mathematical argument

Use all of them
- not always all in every case though