MATHFUN Lecture FP1
Introduction to Functional Programming

Matthew Poole

moodle.port.ac.uk

School of Computing
University of Portsmouth

2014/15

Aim of this part of the unit

- The aim of this part of the unit is to learn a new programming **paradigm**, based on the mathematical concept of a **function**.

Aim of this part of the unit

- The aim of this part of the unit is to learn a new programming **paradigm**, based on the mathematical concept of a **function**.
- Most of the programming you've done so far during your course has been:
  - (primarily-) **procedural programming** in Python; and
  - **object-oriented** programming with Java.
- Use of these and similar languages can lead to a fairly narrow view of programming and problem solving.

Aim of this part of the unit

- The aim of this part of the unit is to learn a new programming **paradigm**, based on the mathematical concept of a **function**.
- Most of the programming you've done so far during your course has been:
  - (primarily-) **procedural programming** in Python; and
  - **object-oriented** programming with Java.
- Use of these and similar languages can lead to a fairly narrow view of programming and problem solving.
- By learning **functional programming** we aim to:
  - see how problems can be solved in different ways;
  - deepen your algorithm development & problem-solving skills.

Assessment

- The practical assessment for this part is worth 30% of the unit.

Assessment

- The practical assessment for this part is worth 30% of the unit.
- There will be a short (30 minute) in-class test worth 10% in the first practical of Teaching Block 2.
- A larger assignment worth 20% will be handed out in mid-TB2.
- The deadline is scheduled for Wednesday 18th March, with demos in the practicals on 19th/20th March.
- There will also be a compulsory exam question on functional programming which will be worth 25% of the exam.

Provisional lecture schedule

Wk	Date	Lecture topic
2	30 Sep	FP1 - Intro. to functional programming
4	14 Oct	FP2 - Intro. to functional programming 2
6	28 Oct	FP3 - Patterns & recursion
8	11 Nov	FP4 - Strings, tuples and lists
10	25 Nov	FP5 - List patterns and recursion
12	9 Dec	FP6 - Functions as values
Christmas break		
14	13 Jan	FP7 - Algebraic types
16	27 Jan	FP8 - Input/output
18	11 Feb	FP9 - Input/output (continued)
20	24 Feb	FP10 - Type classes
22	10 Mar	FP11 - Functional programming in Python
24	24 Mar	no lecture planned

Software (it's all free)

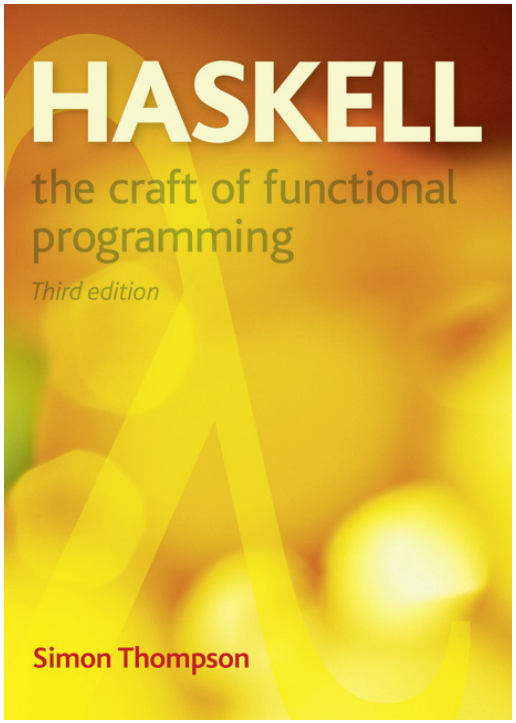
- We will use the functional programming language Haskell.
- We're using the Haskell Platform (which includes the Glasgow Haskell Compiler) for the practical work.
- The Haskell Platform can be downloaded free from `www.haskell.org/platform` for Windows, Mac or Linux.

Software (it's all free)

- We will use the functional programming language Haskell.
- We're using the Haskell Platform (which includes the Glasgow Haskell Compiler) for the practical work.
- The Haskell Platform can be downloaded free from `www.haskell.org/platform` for Windows, Mac or Linux.
- Haskell requires the use of a text editor for editing programs.
- At the University, we'll use Notepad++. This can be downloaded for Windows from `notepad-plus-plus.org`.

Recommended Book

- The following book gives the best match of topics we will cover.

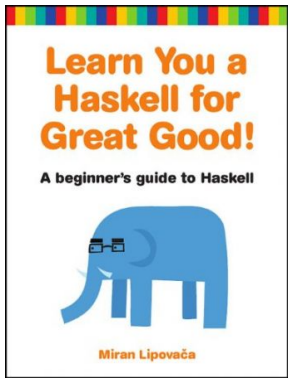


  - Simon Thompson
  - *Haskell: The Craft of Functional Programming*, 2nd - 3rd edition
  - Addison Wesley, 2000 - 2011.

- Many copies are available in the library, especially of the 2nd edition.

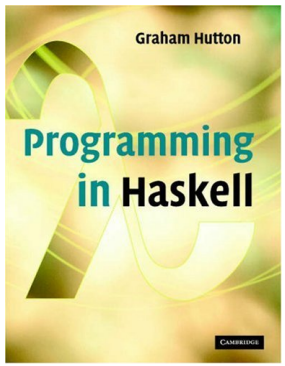
Alternative Books

- Another good textbook is:



  - Miran Lipovača
  - *Learn You a Haskell for Great Good!*
  - No Starch Press, 2011
  - Free online version at `learnyouahaskell.com`.

- A (shorter) alternative is:



  - Graham Hutton
  - *Programming in Haskell*
  - Cambridge University Press, 2007.

Introduction to functional programming

- Functional programming is an example **programming paradigm** (a style/view of programming & program execution).
- We'll begin by introducing some major programming paradigms, and show how functional programming compares with the others.
- There are many ways to classify languages; for example:
  - general-purpose versus special purpose languages; and
  - sequential versus parallel languages.

Introduction to functional programming

- Functional programming is an example **programming paradigm** (a style/view of programming & program execution).
- We'll begin by introducing some major programming paradigms, and show how functional programming compares with the others.
- There are many ways to classify languages; for example:
  - general-purpose versus special purpose languages; and
  - sequential versus parallel languages.
- However, perhaps a more fundamental classification is: **imperative** versus **declarative** languages.
- To understand this distinction, we'll first define what we mean by imperative programming and imperative programming languages.

Imperative programming

- Imperative programming is a style of programming where:
  - computation consists of the execution of **statements**; where
  - the statements operate on a program's **state**.

Imperative programming

- Imperative programming is a style of programming where:
  - computation consists of the execution of **statements**; where
  - the statements operate on a program's **state**.
- For example, the following Java code fragment:

```
x = y + 1;
System.out.println(x);
if (x > 5)
    y = 0;
```

consists of three commands (statements), which operate on the program's state (the values of variables `x` and `y`).

Imperative programming

- Imperative programming is a style of programming where:
  - computation consists of the execution of **statements**; where
  - the statements operate on a program's **state**.
- For example, the following Java code fragment:

```
x = y + 1;
System.out.println(x);
if (x > 5)
    y = 0;
```

  consists of three commands (statements), which operate on the program's state (the values of variables `x` and `y`).
- Imperative programming languages (i.e. languages which are programmed in this way) include Java, C, C++ and Python.

Object-oriented versus procedural programming

- We can further classify imperative languages depending on how code is **structured** (i.e. made modular):
  - in **object-oriented** programming, code is structured into **classes** (which include **methods**);

Object-oriented versus procedural programming

- We can further classify imperative languages depending on how code is **structured** (i.e. made modular):
  - in **object-oriented** programming, code is structured into **classes** (which include **methods**);
  - in **procedural programming**, code is structured into **procedures** (or routines, functions).

Object-oriented versus procedural programming

- We can further classify imperative languages depending on how code is **structured** (i.e. made modular):
  - in **object-oriented** programming, code is structured into **classes** (which include **methods**);
  - in **procedural programming**, code is structured into **procedures** (or routines, functions).
- Languages are classified by the style of programming supported:
  - Java and Smalltalk are object-oriented languages; and
  - C and Pascal are procedural languages.

Object-oriented versus procedural programming

- We can further classify imperative languages depending on how code is **structured** (i.e. made modular):
  - in **object-oriented** programming, code is structured into **classes** (which include **methods**);
  - in **procedural programming**, code is structured into **procedures** (or routines, functions).
- Languages are classified by the style of programming supported:
  - Java and Smalltalk are object-oriented languages; and
  - C and Pascal are procedural languages.
- There also exist **hybrid** languages that support both paradigms; examples include C++ and Python.

Imperative versus declarative programming

- In contrast to imperative prog., in **declarative** programming:
  - there are no statements; and
  - there is no program state to operate upon.

Imperative versus declarative programming

- In contrast to imperative prog., in **declarative** programming:
  - there are no statements; and
  - there is no program state to operate upon.
- Declarative programming languages can be further classified:
  - functional programming, where programs are written as collections of (stateless) function definitions; and
  - logic programming, where programs are made of logic clauses.

Imperative versus declarative programming

- In contrast to imperative prog., in **declarative** programming:
  - there are no statements; and
  - there is no program state to operate upon.
- Declarative programming languages can be further classified:
  - functional programming, where programs are written as collections of (stateless) function definitions; and
  - logic programming, where programs are made of logic clauses.
- Since logic programming is mainly restricted to applications in AI, we concentrate in this unit on functional programming.

The main programming paradigms

- A summary of the "main" programming paradigms is thus:
  - imperative programming:
    - procedural programming
    - object-oriented programming
  - declarative programming:
    - functional programming
    - logic programming

The main programming paradigms

- A summary of the "main" programming paradigms is thus:
  - imperative programming:
    - procedural programming
    - object-oriented programming
  - declarative programming:
    - functional programming
    - logic programming
- Note that not all languages fit nicely into individual classes.
- Also, imperative programming is sometimes (e.g. in some textbooks) used as a synonym for procedural programming.

Functional Programming Languages

- Functional programming has been around for almost as long as imperative programming.
- The first well-known functional language was Lisp (1958).

Functional Programming Languages

- Functional programming has been around for almost as long as imperative programming.
- The first well-known functional language was Lisp (1958).
- Functional languages used today include:
  - Common Lisp and Scheme (dialects of Lisp)
  - Erlang

Functional Programming Languages

- Functional programming has been around for almost as long as imperative programming.
- The first well-known functional language was Lisp (1958).
- Functional languages used today include:
  - Common Lisp and Scheme (dialects of Lisp)
  - Erlang
- Most functional languages can be considered impure, as they include some imperative features.
- We'll use Haskell since it is the most well-known pure functional language, making it easier to learn functional concepts.

Functional Programming Languages

- Many recently-designed languages are built around functional programming concepts; in particular:
  - Scala: a Java-like functional/OO language for the Java Virtual Machine (JVM);
  - F♯: a language from Microsoft for the .NET platform (integrated into Visual Studio 2012).

Functional Programming Languages

- Many recently-designed languages are built around functional programming concepts; in particular:
  - Scala: a Java-like functional/OO language for the Java Virtual Machine (JVM);
  - F♯: a language from Microsoft for the .NET platform (integrated into Visual Studio 2012).
- Finally, many popular imperative languages include functional programming features; e.g. Python, JavaScript, Java and C♯.
- Core functional techniques, concepts and problem-solving skills learnt using Haskell can be transferred to these languages.

Functional Programming in Haskell

- In the rest of this lecture we introduce the basics of Haskell.

Functional Programming in Haskell

- In the rest of this lecture we introduce the basics of Haskell.
- To begin, we need to make sure we understand the terms **expression**, **value** and **evaluation**:

Functional Programming in Haskell

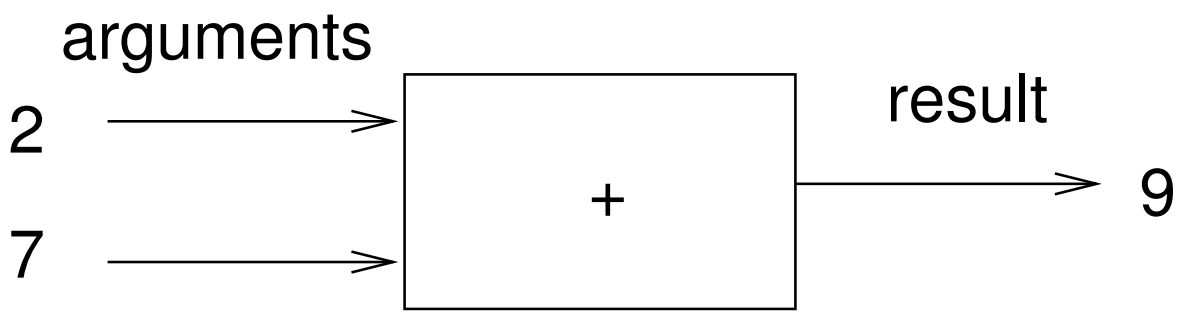
- In the rest of this lecture we introduce the basics of Haskell.
- To begin, we need to make sure we understand the terms **expression**, **value** and **evaluation**:

- We also need to know what a (mathematical) **function** is.

Functional Programming in Haskell

- In the rest of this lecture we introduce the basics of Haskell.
- To begin, we need to make sure we understand the terms **expression**, **value** and **evaluation**:

- We also need to know what a (mathematical) **function** is.
- We can represent a function as a box which maps **argument** or **parameter** values to a **result** value; for example

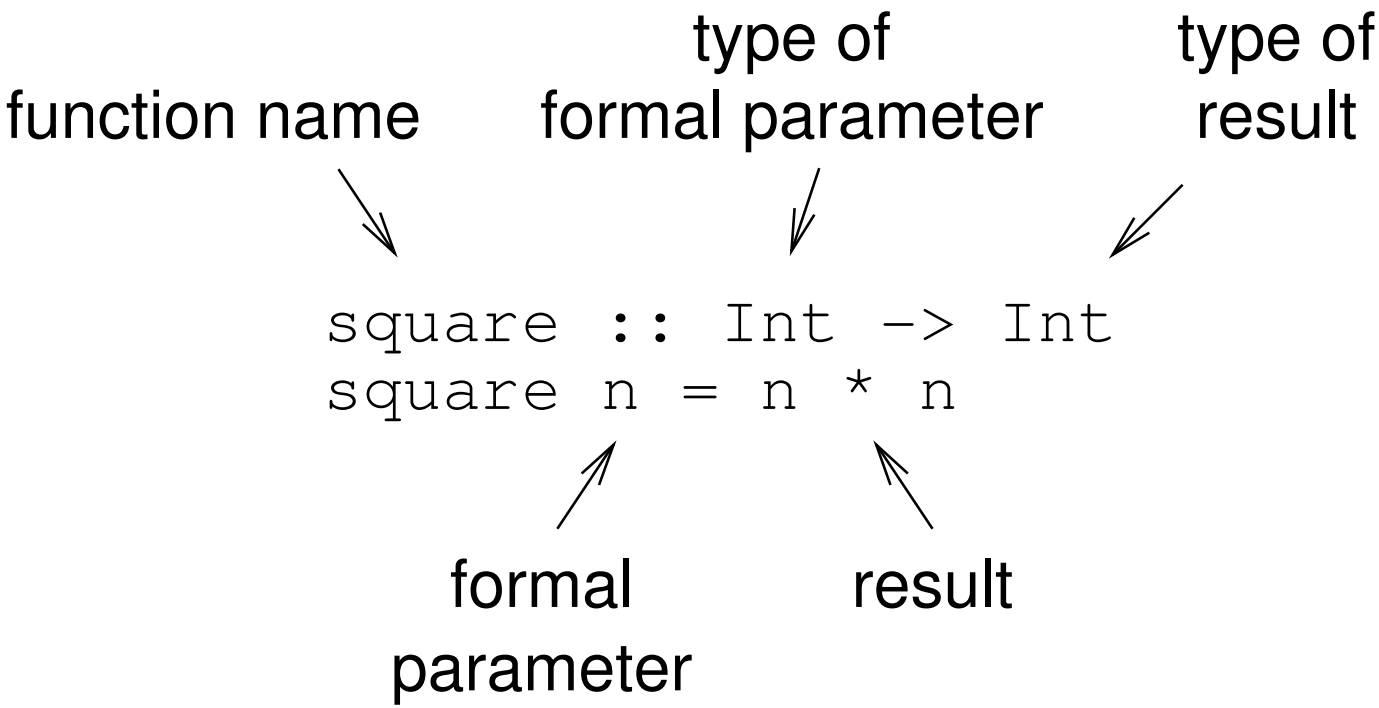


Definitions

- A Haskell program is made up of a number of function **definitions**.

Definitions

- A Haskell program is made up of a number of function **definitions**.
- An example definition, accompanied by a **declaration of its type**, is:

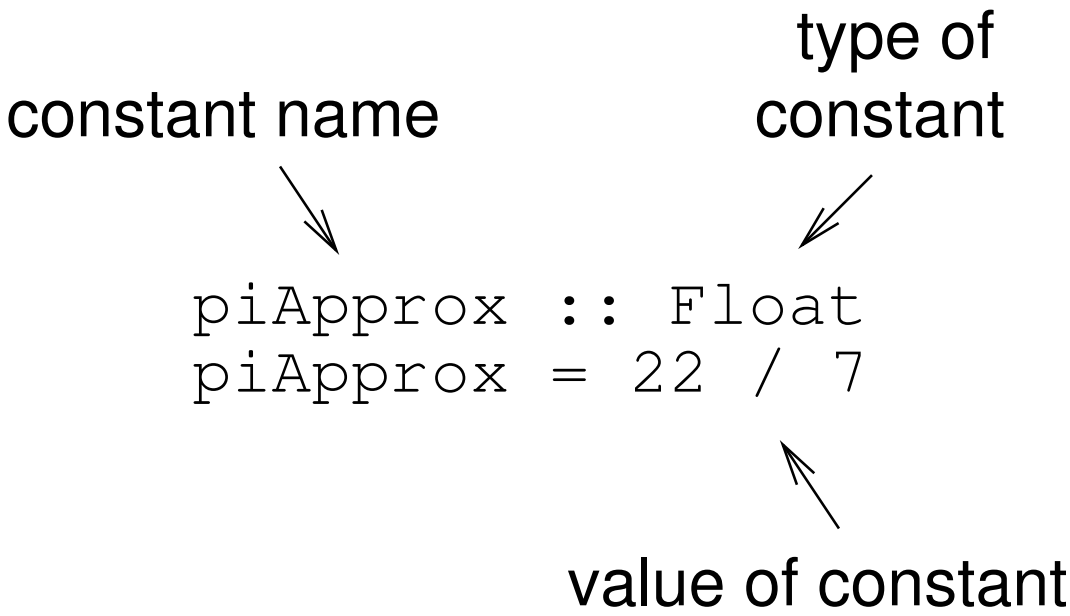


Definitions

- We read the $::$ in the above definition as "is of type"; most of the rest of the definition is standard mathematics.

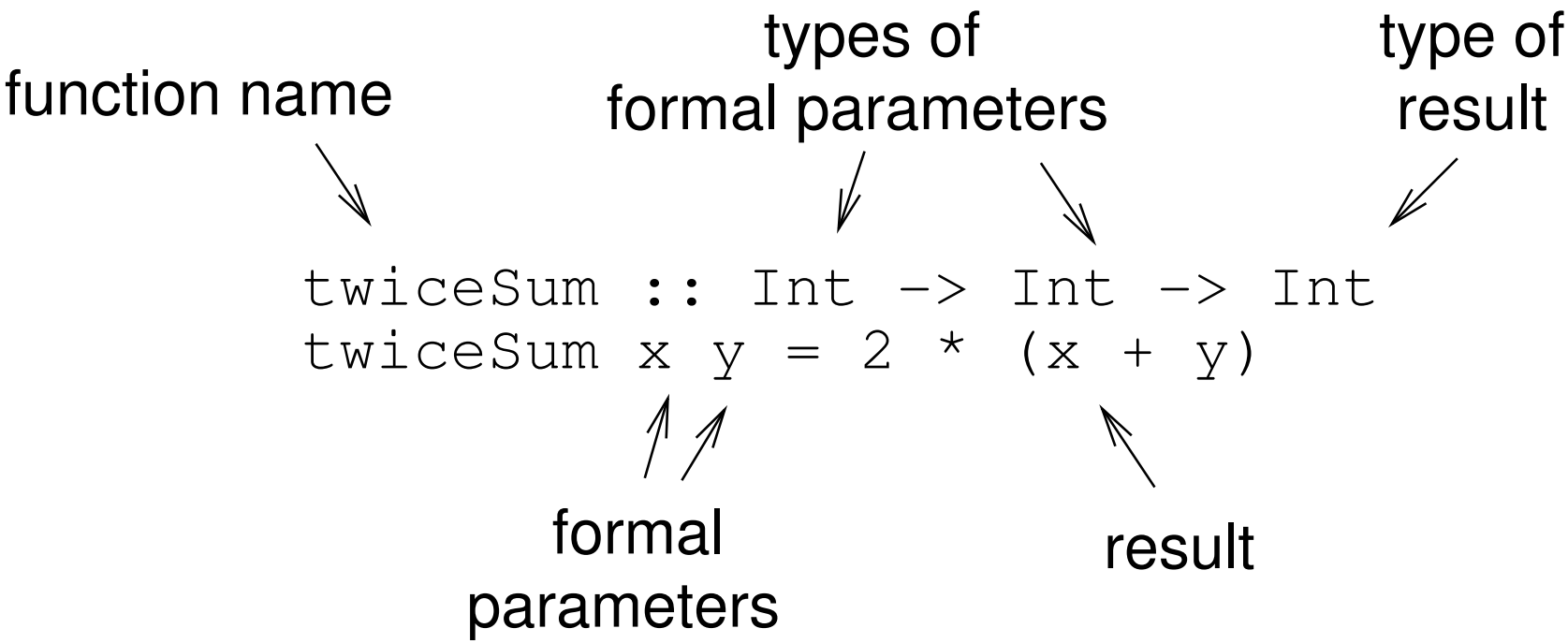
Definitions

- We read the `::` in the above definition as "is of type"; most of the rest of the definition is standard mathematics.
- The following definition doesn't have any formal parameters; it is a definition of a **constant**:



Definitions

- The following is a function definition with two parameters:



- The reason why we use -> to separate the types of the formal parameters, rather than $\times$, will become clear later.

Executing these functions using GHCi

- Let's try some expressions using GHCi assuming the above definitions have been loaded in.

```
ghci>
```

Executing these functions using GHCi

- Let's try some expressions using GHCi assuming the above definitions have been loaded in.

```
ghci> square 4
```

Executing these functions using GHCi

- Let's try some expressions using GHCi assuming the above definitions have been loaded in.

```
ghci> square 4
16
ghci>
```

Executing these functions using GHCi

- Let's try some expressions using GHCi assuming the above definitions have been loaded in.

```
ghci> square 4
16
ghci> piApprox + 1
```

Executing these functions using GHCi

- Let's try some expressions using GHCi assuming the above definitions have been loaded in.

```
ghci> square 4
16
ghci> piApprox + 1
4.142857
ghci>
```

Executing these functions using GHCi

- Let's try some expressions using GHCi assuming the above definitions have been loaded in.

```
ghci> square 4
16
ghci> piApprox + 1
4.142857
ghci> twiceSum 2 4
```

Executing these functions using GHCi

- Let's try some expressions using GHCi assuming the above definitions have been loaded in.

```
ghci> square 4
16
ghci> piApprox + 1
4.142857
ghci> twiceSum 2 4
12
```

Executing these functions using GHCi

- Let's try some expressions using GHCi assuming the above definitions have been loaded in.

```
ghci> square 4
16
ghci> piApprox + 1
4.142857
ghci> twiceSum 2 4
12
```

- (Because function application is so fundamental in Haskell, we don't use parentheses & commas, as in $twiceSum(2, 4)$.)
- Note: In the notes we will assume the GHCi prompt is `ghci>`.

Executing these functions using GHCi

- Let's continue:

```
ghci> square 2 + 1
5
ghci> square (2 + 1)
9
ghci> twiceSum (square 2) 3
14
```

- We note here that function application has higher precedence than operator application.

Basic Types: Bool

- Haskell's built-in types include `Bool`, `Int`, `Float` and `Char`.
- We'll look at some of these here, leaving `Char` until later.

Basic Types: Bool

- Haskell's built-in types include `Bool`, `Int`, `Float` and `Char`.
- We'll look at some of these here, leaving `Char` until later.
- The `Bool` data type has the values `True` and `False` and the Boolean operators `&&` (and), `||` (or) and `not`.

Basic Types: Bool

- Haskell's built-in types include `Bool`, `Int`, `Float` and `Char`.
- We'll look at some of these here, leaving `Char` until later.
- The `Bool` data type has the values `True` and `False` and the Boolean operators `&&` (and), `||` (or) and `not`.
- For example, the function definition:

```
exOr :: Bool -> Bool -> Bool
exOr x y = (x || y) && not (x && y)
```

  implements **exclusive or** (i.e. it gives True when exactly one of its arguments is True).

Basic Types: Int & Float

- We will use two of Haskell's numeric data types:
  - `Int` is a fixed-space integer data type; and
  - `Float` is a floating-point data type.

Basic Types: Int & Float

- We will use two of Haskell's numeric data types:
  - `Int` is a fixed-space integer data type; and
  - `Float` is a floating-point data type.
- Operators for these data types include:
  - the arithmetic operators `+`, `-` and `*`;
  - the `Float` data type includes floating point division `/`;
  - `Int` has integer division and remainder functions `div` and `mod`;
  - relational operators `==`, `/=` (not equals), `<`, `>`, `<=` and `>=`.
- These operators use the standard precedence rules (as for Java).

Basic Types: Int & Float

- We will use two of Haskell's numeric data types:
  - `Int` is a fixed-space integer data type; and
  - `Float` is a floating-point data type.
- Operators for these data types include:
  - the arithmetic operators `+`, `-` and `*`;
  - the `Float` data type includes floating point division `/`;
  - `Int` has integer division and remainder functions `div` and `mod`;
  - relational operators `==`, `/=` (not equals), `<`, `>`, `<=` and `>=`.
- These operators use the standard precedence rules (as for Java).

Conditional expressions

- Haskell includes a conditional **expression**, which takes the form:

  ```
  if condition then m else n
  ```

  where `condition` is a Boolean expression, and `m` & `n` are expressions of the same type.
- Here:
  - if the `condition` is True, the whole expression evaluates to `m`;
  - if the `condition` is False, it evaluates to `n`.

Conditional expressions

- Haskell includes a conditional **expression**, which takes the form:

  ```
  if condition then m else n
  ```

  where `condition` is a Boolean expression, and `m` & `n` are expressions of the same type.
- Here:
  - if the `condition` is True, the whole expression evaluates to `m`;
  - if the `condition` is False, it evaluates to `n`.
- For example, the expression

  ```
  if 3 > 4 then 7 else 9
  ```

  evaluates to 9.

Conditional expressions

- Haskell includes a conditional **expression**, which takes the form:

  ```
  if condition then m else n
  ```

  where `condition` is a Boolean expression, and `m` & `n` are expressions of the same type.
- Here:
  - if the `condition` is True, the whole expression evaluates to `m`;
  - if the `condition` is False, it evaluates to `n`.
- For example, the expression

  ```
  if 3 > 4 then 7 else 9
  ```

  evaluates to 9.
- In the next lecture, we'll see a more general alternative to conditional expressions.