

# CS\_191 Functional Programming I

Ulrich Berger

Department of Computer Science  
Swansea University  
Spring 2010

`u.berger@swan.ac.uk` ,  
`http://www.cs.swan.ac.uk/~csulrich/`  
tel 513380, fax 295708, room 306, Faraday Tower

# What is a functional program?

# What is a functional program?

- ▶ A functional program is an *expression*.

# What is a functional program?

- ▶ A functional program is an *expression*.  
Example:  $(2 * 3) + 4$

# What is a functional program?

- ▶ A functional program is an *expression*.  
Example:  $(2 * 3) + 4$
- ▶ To *run* a functional program means to *evaluate* it.

# What is a functional program?

- ▶ A functional program is an *expression*.  
Example:  $(2 * 3) + 4$
- ▶ To *run* a functional program means to *evaluate* it.  
Example:  $(2 * 3) + 4$  evaluates to 10.

# What is a functional program?

- ▶ A functional program is an *expression*.  
Example:  $(2 * 3) + 4$
- ▶ To *run* a functional program means to *evaluate* it.  
Example:  $(2 * 3) + 4$  evaluates to 10.
- ▶ There are *no assignments* (such as  $x = x+1$ ).

## Example: Summing up the integers 1 to 10

## Example: Summing up the integers 1 to 10

In Java:

```
total = 0;
for (i = 1; i <= 10; ++i)
    total = total+i;
```

## Example: Summing up the integers 1 to 10

In Java:

```
total = 0;
for (i = 1; i <= 10; ++i)
    total = total+i;
```

The computation method is *assignment*.

## Example: Summing up the integers 1 to 10

In Java:

```
total = 0;
for (i = 1; i <= 10; ++i)
    total = total+i;
```

The computation method is *assignment*.

In the functional language Haskell:

```
sum [1..10]
```

## Example: Summing up the integers 1 to 10

In Java:

```
total = 0;
for (i = 1; i <= 10; ++i)
    total = total+i;
```

The computation method is *assignment*.

In the functional language Haskell:

```
sum [1..10]
```

The computation method is *function application*

# Strengths of functional programming

# Strengths of functional programming

- ▶ Simplicity and clarity (easy to learn, readable code)

# Strengths of functional programming

- ▶ Simplicity and clarity (easy to learn, readable code)
- ▶ Reliability (correctness of a program can be proven),

## Strengths of functional programming

- ▶ Simplicity and clarity (easy to learn, readable code)
- ▶ Reliability (correctness of a program can be proven),
- ▶ Productivity (algorithmic problems can be solved quickly).

# Strengths of functional programming

- ▶ Simplicity and clarity (easy to learn, readable code)
- ▶ Reliability (correctness of a program can be proven),
- ▶ Productivity (algorithmic problems can be solved quickly).
- ▶ Maintainability and adaptability (programs can easily be adapted to changing requirements without introducing errors).

## Strengths of functional programming

- ▶ Simplicity and clarity (easy to learn, readable code)
- ▶ Reliability (correctness of a program can be proven),
- ▶ Productivity (algorithmic problems can be solved quickly).
- ▶ Maintainability and adaptability (programs can easily be adapted to changing requirements without introducing errors).
- ▶ Ericsson measured an improvement factor in productivity and reliability of between 9 and 25 in experiments on telephony software.

## Haskell B Curry (1900 – 1982)



Source: <http://www-groups.dcs.st-and.ac.uk/~history>

# Short history of functional programming

- ▶ Foundations 1920–30  
Combinatory Logic and Lambda-calculus (Schönfinkel, Curry, Church)
- ▶ First functional languages 1960  
LISP (McCarthy), ISWIM (Landin)
- ▶ Further functional languages 1970–80  
FP (Backus); ML (Milner, Gordon), later SML and CAML;  
Hope (Burstall); Miranda (Turner)
- ▶ 1990: Haskell, Clean

# The main current functional programming languages

Language	Types	Evaluation	I/O
<i>Lisp, Scheme</i>	type free	eager	via side effects
<i>ML, CAML</i>	polymorphic	eager	via side effects
<i>Haskell, Clean</i>	polymorphic	lazy	via monads

# Lab classes and exams

## ▶ Lab classes

**Attendance** is compulsory and will be monitored.

**Lab exercises** are compulsory and are assessed (10%).

**Place and time:** Linux lab, room 217, Mondays, 12-1, 1-2, start 25th of February (today!).

# Lab classes and exams

## ▶ Lab classes

**Attendance** is compulsory and will be monitored.

**Lab exercises** are compulsory and are assessed (10%).

**Place and time:** Linux lab, room 217, Mondays, 12-1, 1-2, start 25th of February (today!).

## ▶ Coursework

There will be **2 courseworks**, each worth 10%.

**Deadlines are strict!** Solutions will be handed out on the first Thursday after each submission date.

**Submission:** please put printout with signature(s) in the wooden box on the second floor. The box will be emptied in the morning after the submission date.

## ghci and hugs

We will work with interactive versions of Haskell. This makes it particularly easy to learn the language and test programs.

- ▶ ghci (Glasgow Haskell Compiler Interactive) comes with the GHC distribution. It is installed on the Linux computers in the labs.
- ▶ hugs (Haskell User Gofer System) is installed on the Linux and Windows computers in the labs.

Information on how to download, install and start ghci and hugs can be found on the course web page.

In the lectures we will work with ghci and it is recommended to use ghci in the Linux labs.

# Definitions in Haskell

A Haskell definition consists of

# Definitions in Haskell

A Haskell definition consists of

a *type declaration* (or signature)

```
x :: Int
```

# Definitions in Haskell

A Haskell definition consists of

a *type declaration* (or signature)

```
x :: Int
```

the *defining equation*

```
x = 2+3
```

# Definitions in Haskell

A Haskell definition consists of

a *type declaration* (or signature)

```
x :: Int
```

the *defining equation*

```
x = 2+3
```

Both, type declaration and defining equation must start at the beginning of a line!

## Defining a function

We can define a *function* that computes the area of a circle for a given radius  $r$ .

```
ar :: Float -> Float
ar r = r^2 * pi
```

The signature `ar :: Float -> Float` means that `ar` is a function that expects a floating point number as input and returns a floating point number as output.

The variable `r` is called *formal parameter*. It represents an arbitrary input.

Instead of "input of a function" one often says "argument of a function".

## Functions of more than one argument

```
average :: Float -> Float -> Float
average x y = (x+y)/2
```

The signature `average :: Float -> Float -> Float` means that `average` is a function that expects two floating point numbers as input (represented by the formal parameters `x` and `y`) and returns a floating point number as output.

## Mathematical notation versus Haskell syntax

Maths	Haskell
$\max(x, y)$	<code>max x y</code>
$3(x^2 + y^2) + 5$	<code>3*(x^2+y^2)+5</code>
$\text{ar} : \text{Float} \rightarrow \text{Float}$	<code>ar :: Float -&gt; Float</code>

## Case sensitivity, prefix vs. infix

- ▶ Haskell is case sensitive:
  - identifiers, function names and formal parameters must be lower cases (x and area, but not X and not Area)
  - types must be upper case (Float, but not float)

## Case sensitivity, prefix vs. infix

- ▶ Haskell is case sensitive:
  - identifiers, function names and formal parameters must be lower cases (`x` and `area`, but not `X` and not `Area`)
  - types must be upper case (`Float`, but not `float`)
- ▶ Prefix operators bind stronger than infix operators:
  - `max 3 4 + 5` means `(max 3 4) + 5`
  - but not `max 3 (4 + 5)`.

# What are types?

# What are types?

A type is a name for a collection of similar objects.

# What are types?

A type is a name for a collection of similar objects.

There are

- ▶ *predefined primitive types* such as Bool, Int, Integer, Float, Double, Char (these will be discussed next)

# What are types?

A type is a name for a collection of similar objects.

There are

- ▶ *predefined primitive types* such as Bool, Int, Integer, Float, Double, Char (these will be discussed next)
- ▶ *complex types* such as pairs, lists, arrays, function types

# What are types?

A type is a name for a collection of similar objects.

There are

- ▶ *predefined primitive types* such as Bool, Int, Integer, Float, Double, Char (these will be discussed next)
- ▶ *complex types* such as pairs, lists, arrays, function types
- ▶ *user defined types*

# Why types?

- ▶ Early detection of errors at compile time
- ▶ Compiler can use type information to improve efficiency
- ▶ Type signatures facilitate program development
- ▶ and make programs more readable
- ▶ Types increase productivity and security

# Booleans

- ▶ Values: True and False (uppercase!)

- ▶ Predefined functions:

<code>not</code>	<code>:: Bool -&gt; Bool</code>	<i>negation</i> (not)
<code>(&amp;&amp;)</code>	<code>:: Bool -&gt; Bool -&gt; Bool</code>	<i>conjunction</i> (and)
<code>(  )</code>	<code>:: Bool -&gt; Bool -&gt; Bool</code>	<i>disjunction</i> (or)

## Example: Exclusive Or

The predefined function `||` defines *Inclusive Or*:

`x || y` evaluates to `True` if `x`, or `y`, or both are `True`.

We can define *Exclusive Or* using the predefined functions `||`, `&&` and `not`:

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

## Infix vs. prefix

A function of two arguments whose name is composed from the special symbols

!, #, \$, %, &, \*, +, -, /, <, =, >, ?, @, ^, |

for example `||`, can be used

- ▶ infix without brackets: `x || y`
- ▶ prefix with brackets: `(||) x y`

A function of two arguments whose name is composed of letters, for example `max`, can be used

- ▶ infix with inverted commas: `x 'max' y`
- ▶ prefix without inverted commas: `max x y`

In a type signature the prefix version of a name must be used:

```
(||) :: Bool -> Bool -> Bool.
```

## Defining a logic gate by its truth table

We can define the function `xor` by listing its truth table:

```
xor :: Bool -> Bool -> Bool
xor True  True   = False
xor True  False  = True
xor False True    = True
xor False False  = False
```

One may also use wildcards:

```
xor :: Bool -> Bool -> Bool
xor True  False = True
xor False True  = True
xor _     _     = False
```

If wildcards are used, then the *order* of the defining equations matters!

# Basic numeric types

## *Computing with numbers*

Limited precision      arbitrary precision  
constant cost       $\longleftrightarrow$       increasing cost

# Basic numeric types

## *Computing with numbers*

Limited precision  
constant cost  $\longleftrightarrow$  arbitrary precision  
increasing cost

Haskell offers:

- ▶ `Int` - integers as machine words
- ▶ `Integer` - arbitrarily large integers
- ▶ `Rational` - arbitrarily precise rational numbers
- ▶ `Float` - floating point numbers
- ▶ `Double` - double precision floating point numbers

## Some predefined numeric functions

```
(+), (*), (^), (-), min, max :: Int -> Int -> Int
-      :: Int -> Int          -- unary minus
abs    :: Int -> Int         -- absolute value
div    :: Int -> Int -> Int  -- integer division
mod    :: Int -> Int -> Int  -- remainder of int. div.
gcd    :: Int -> Int -> Int  -- greatest common divisor
```

## Comparison operators

`(==), (/=), (<=), (<), (>=), (>) :: Int -> Int -> Bool`

But also

`(==), (/=), (<=), (<), (>=), (>) :: a -> a -> Bool`

where `a = Bool, Integer, Rational, Float, Double`

## Floating point numbers: Float, Double

- ▶ Single and double precision Floating point numbers
- ▶ The arithmetic operations (+), (-), (\*), (/) may also be used for Float and Double
- ▶ Float and Double support the same operations

```
(/) :: Float -> Float -> Float
```

```
pi  :: Float
```

```
exp,log,sqrt,logBase,sin,cos :: Float -> Float
```

## Conversion from and to integers

```
fromIntegral :: Int -> Float
fromIntegral :: Integer -> Float
```

```
round :: Float -> Int      -- round to nearest integer
round :: Float -> Integer
```

### *Example*

```
half :: Int -> Float
half x = x / 2
```

Does not work because division (/) expects two floating point numbers as arguments, but x has type Int.

Solution:

```
half :: Int -> Float
half x = (fromIntegral x) / 2
```

Notation for characters: 'a'

Notation for strings: "Hello"

```
(:)  :: Char -> String -> String  -- prefixing  
(++) :: String -> String -> String -- concatenation
```

(:) binds stronger than (++)

```
"Hello " ++ 'W' : "orld!"  ⇒  "Hello World!"
```

Is the same as

```
"Hello " ++ ('W' : "orld!")
```

## The ASCII code of a character

```
fromEnum :: Char -> Int
```

computes the ASCII code of a character:

```
fromEnum 'a' ⇒ 97
```

```
fromEnum 'b' ⇒ 98
```

```
fromEnum '9' ⇒ 57
```

```
toEnum :: Int -> Char
```

is its inverse. `toEnum 97 :: Char ⇒ 'a'`

The ASCII codes of characters denoting capital letters, small letters and digits are arranged in blocks. For example, the characters 'A', ..., 'Z' have the codes 65, ..., 90.

## If-then-else

```
max2 :: Int -> Int -> Int
max2 x y = if x < y then y else x
```

```
signum :: Int -> Int
signum x = if x > 0 then 1 else
           if x == 0 then 0 else (-1)
```

(exercises)

## Guarded equations

```
signum1 :: Int -> Int
signum1 x
  | x > 0      = 1
  | x == 0     = 0
  | otherwise  = 1
```

(exercise)

## Case analysis with pattern matching

```
empty :: String -> Bool
empty s = case s of
    ""    -> True
    _     -> False
```

```
tl :: String -> String
tl xs = case xs of
    ""      -> ""
    (_:xs) -> xs
```

(layout)

## Booleans or Strings as pattern

Case-expressions can be also used with Boolean constants or constant strings as pattern:

```
(&&&) :: Bool -> Bool -> Bool  
x &&& y = case x of {False -> False ; True -> y}
```

```
age :: String -> Int  
age x = case x of  
    {"John" -> 21 ; "Alice" -> 20 ; _ -> -1}
```

## Defining equations with pattern matching

```
(&&&) :: Bool -> Bool -> Bool
False &&& y = False
_      &&& y = y
```

```
age :: String -> Int
age "John"    = 21
age "Alice"   = 20
age _         = -1
```

(exercises, explain case)

## Exception handling

A better solution for the function `age` is to raise an exception when the person is not known:

```
age :: String -> Int
age "John"    = 21
age "Alice"  = 20
age x = error ("The age of " ++ x ++ " is not known")
```

(explain error, exercise)

## let and where

```
g :: Float -> Float -> Float
g x y = (x2 + y2) / (x2 + y2 + 1)
```

better

```
g x y = let s = x2 + y2
         in s / (s + 1)
```

or

```
g x y = s / (s + 1) where
    s = x2 + y2
```

(exercise)

## Local definition of a function

The sum of the areas of two circles with radii  $r, s$ .

```
totalArea :: Float -> Float -> Float
totalArea r s = pi * r^2 + pi * s^2
```

We make the program more modular and readable by using an auxiliary function to compute the area of one circle:

```
totalArea :: Float -> Float -> Float
totalArea r s = circleArea r + circleArea s

where circleArea x = pi * x^2
```

(exercises)

## Modules for large program systems

Local definitions are useful for structuring small program units.

# Modules for large program systems

Local definitions are useful for structuring small program units.

In order to structure large program systems one uses *modules*.

(demo)

## Exercise

Create a module that contains test functions for Haskell's implementation of the square root function (`sqrt`) and the natural logarithm (`log`). Note that the following equations are expected to hold.

$$(\sqrt{x})^2 = x \quad \text{for } x \geq 0$$

$$e^{\log(x)} = x \quad \text{for } x > 0$$

The Haskell function mapping  $x$  to  $e^x$  is called `exp`. Your test functions should have an extra parameter controlling the “hardness” of the test.

Import your module in another Haskell script and try it out.

# Type synonyms

Programs become more readable if types are given names that indicate what its members are used for.

## Type synonyms

Programs become more readable if types are given names that indicate what its members are used for.

For example, in the definitions of the areas of a rectangle and a circle the formal parameters of type `Float` sometimes play the role of the side of a rectangle and sometimes the role of the radius of a circle. The results represent areas.

## Type synonyms

```
type Side    = Float
type Radius  = Float
type Area    = Float
```

```
rectangleArea :: Side -> Side -> Area
rectangleArea a b = a*b
```

```
circleArea :: Radius -> Area
circleArea r = pi*r^2
```

## Exercise

Introduce type synonyms to make the program computing the roots of a quadratic more readable.