Functional programming
Lecture 02 — First steps

Stéphane Vialette
stephane.vialette@univ-eiffel.fr

February 13, 2023

Laboratoire d'Informatique Gaspard-Monge, UMR CNRS 8049,
Université Gustave Eiffel

Schedule

1. Types & functions
2. Function on lists
3. High-order functions & list comprehensions
4. Test
5. Algebraic data type – Binary search trees
6. Algebraic data type – General trees
7. Problem solving – Countdown
8. *Functional images*
9. *File system*
10. *Radar* or *Cloud simulator*
11. Test

Functional programming concepts

Functional programming concepts

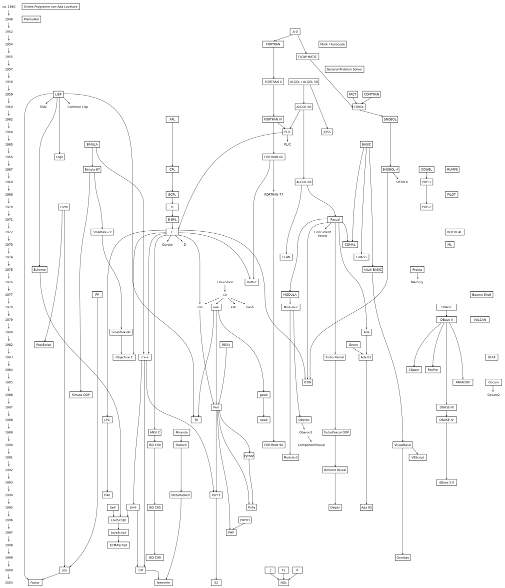
First steps

Types and classes

Defining functions

Some functions

Genealogy of programming languages



Functional languages

Scala

elixir

ERLANG

cljs

PURESCRIPT

BS

F#

OCaml

Main functional programming languages

Lisp

Lisp (historically, LISP) is a family of computer programming languages with a long history and a distinctive, fully parenthesized prefix notation. Originally specified in 1958, Lisp is the second-oldest high-level programming language in widespread use today. (Only Fortran is older, by one year.)

Main functional programming languages

ERLANG

Erlang

Erlang is a general-purpose, concurrent, functional programming language, as well as a garbage-collected runtime system.

Main functional programming languages

Elixir
Elixir is a functional, concurrent, general-purpose programming language that runs on the Erlang virtual machine (BEAM).

Main functional programming languages

F#

F# is a strongly typed, multi-paradigm programming language that encompasses functional, imperative, and object-oriented programming methods. It is being developed at Microsoft Developer Division and is being distributed as a fully supported language in the .NET framework.

Main functional programming languages

Ocaml

Ocaml, originally named Objective Caml, is the main implementation of the programming language Caml. OCaml's toolset includes an interactive top-level interpreter, a bytecode compiler, a reversible debugger, a package manager (OPAM), and an optimizing native code compiler.

Main functional programming languages

Clojure

Clojure is a dialect of the Lisp programming language. Clojure is a general-purpose programming language with an emphasis on functional programming. It runs on the Java virtual machine and the Common Language Runtime.

Main functional programming languages

Racket

Racket, formerly PLT Scheme, is a general purpose, multi-paradigm programming language in the Lisp-Scheme family. One of its design goals is to serve as a platform for language creation, design, and implementation

Main functional programming languages

Elm

Elm is a domain-specific programming language for declaratively creating web browser-based graphical user interfaces. Elm is purely functional, and is developed with emphasis on usability, performance, and robustness.

Main functional programming languages

Scala

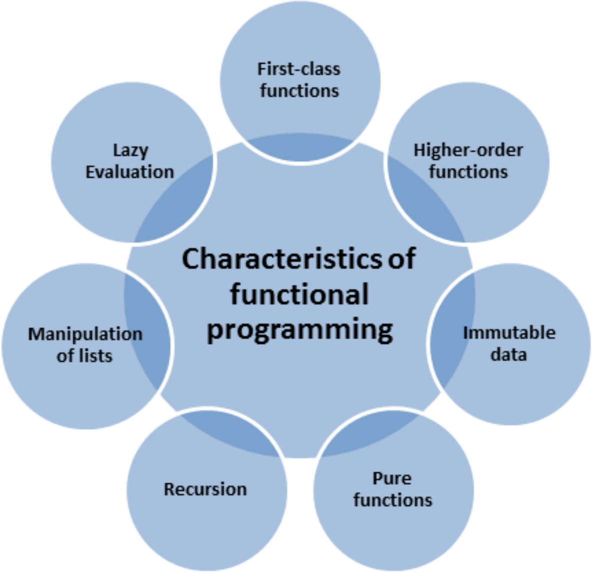
Scala is a general-purpose programming language providing support for functional programming and a strong static type system. Designed to be concise, many of Scala's design decisions aimed to address criticisms of Java.

Main functional programming languages

Haskell

Haskell is a general-purpose, statically-typed, purely functional programming language with type inference and lazy evaluation. Designed for teaching, research and industrial applications, Haskell has pioneered a number of programming language features such as type classes, which enable type-safe operator overloading, and monadic IO. Haskell's main implementation is the Glasgow Haskell Compiler (GHC). It is named after logician Haskell Curry.

Characteristics of functional programming



Haskell

- Haskell is a compiled, statically typed, functional programming language.
- It was created in the early 1990s as one of the first open-source purely functional programming languages.
- It is named after the American logician Haskell Brooks Curry.

Glasgow Haskell Compiler

- Concise programs
- Powerful type system
- List comprehensions
- Recursive functions
- High-order functions
- Effectful functions
- Generic functions
- Lazy evaluation
- Equational reasoning

Haskell landscape

The imperatives

- GHC: state-of-the-art, open source, compiler and interactive environment for the functional language Haskell.
- GHCi: GHC's interactive environment.
- Hackage: Haskell community's central package archive of open source software.

Haskell landscape

Testing Frameworks

- QuickCheck: powerful testing framework where test cases are generated according to specific properties.
- HUnit: unit testing framework similar to JUnit.
- Hspec: a testing framework similar to RSpec with support for QuickCheck and HUnit.
- The Haskell Test Framework, HTF: integrates both Hunit and QuickCheck.

Ancillary Tools

- darcs: revision control system.
- haddock: documentation system.
- cabal: build system.
- stack: build system.
- hoogle: type-aware API search engine.

Haskell landscape

Static Analysis Tools

- hlint: detect common style mistakes and redundant parts of syntax, improving code quality.
- Sourcegraph: Haskell visualizer.

Dynamic Analysis Tools

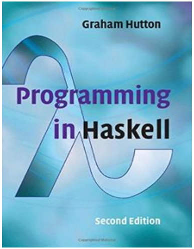
- criterion: powerful benchmarking framework.
- hpc: check evaluation coverage of a haskell program, useful for determining test coverage.

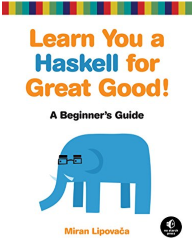
Haskell landscape

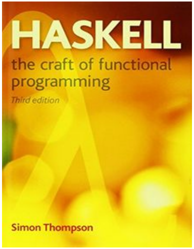
IDEs

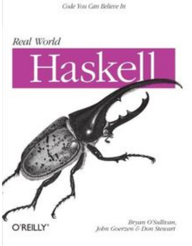
- VSCodium.
- IntelliJ.
- Vim.
- GNU Emacs.
- Haskell for Mac (commercial).
- Sublime Text (commercial)

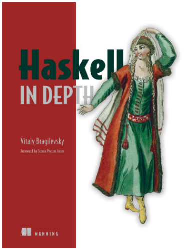
Haskell books

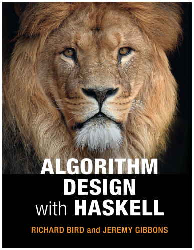


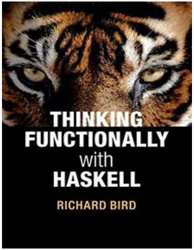


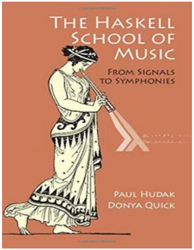




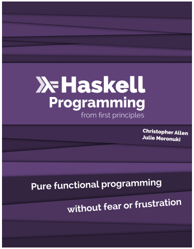


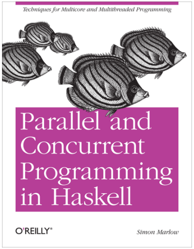


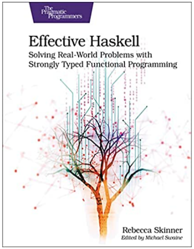


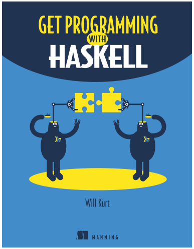


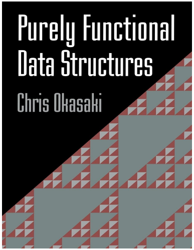
Haskell books

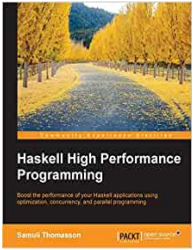


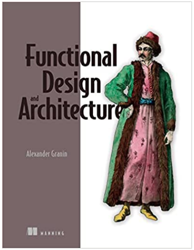


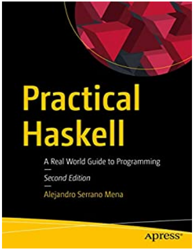




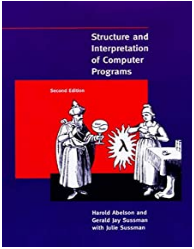


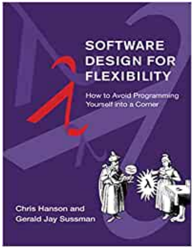


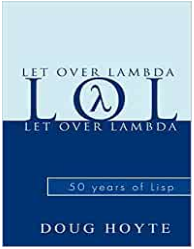


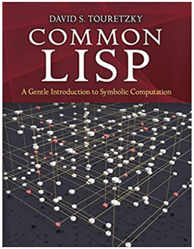


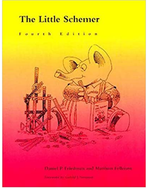
Functional programming books

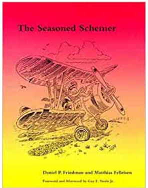


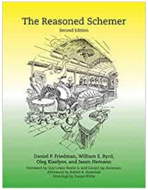


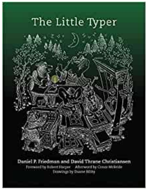


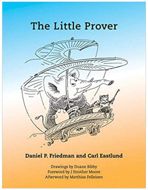












A taste of haskell

```haskell
sum :: Num a => [a] -> a
sum []          = 0
sum (x : xs) = x + sum xs
```

A taste of haskell

```haskell
sum :: Num a => [a] -> a
sum []        = 0
sum (x : xs) = x + sum xs
```

```
  sum [1,2,3]
=   { applying function sum }
  1 + sum [2,3]
=   { applying function sum }
  1 + (2 + sum [3])
=   { applying function sum }
  1 + 2 + (3 + sum [])
=   { applying function sum }
  1 + (2 + (3 + 0))
=   { applying function +   }
  1 + (2 + 3)
=   { applying function +   }
  1 + 5
=   { applying function +   }
  6
```

A taste of haskell

```haskell
qsort :: Ord a => [a] -> [a]
qsort [] = []
qsort (x : xs) = qsort smaller ++ [x] ++ qsort larger
  where
    smaller = [x' | x' <- xs, x' <= x]
    larger  = [x' | x' <- xs, x' >  x]
```

A taste of haskell

```haskell
qsort :: Ord a => [a] -> [a]
qsort [] = []
qsort (x : xs) = qsort smaller ++ [x] ++ qsort larger
  where
    smaller = [x' | x' <- xs, x' <= x]
    larger  = [x' | x' <- xs, x' >  x]
```

```
  qsort [x]
=   { applying function qsort }
  qsort [] ++ [x] ++ qsort [x]
=   { applying function qsort }
  [] ++ [x] ++ []
=   { applying function ++ }
  [x]
```

A taste of haskell

```haskell
qsort :: Ord a => [a] -> [a]
qsort [] = []
qsort (x : xs) = qsort smaller ++ [x] ++ qsort larger
  where
    smaller = [x' | x' <- xs, x' <= x]
    larger  = [x' | x' <- xs, x' >  x]
```

```
  qsort [3,5,1,4,2]
=   { applying function qsort }
  qsort [1,2] ++ [3] ++ qsort [5,4]
=   { applying function qsort }
  (qsort [] ++ [1] ++ qsort [2]) ++ [3] ++ (qsort [4] ++ [5] ++ qsort [])
=   { applying function qsort }
  ([] ++ [1] ++ [2]) ++ [3] ++ ([4] ++ [5] ++ [])
=   { applying function ++ }
  [1,2] ++ [3] ++ [4,5]
=   { applying function ++ }
  [1,2,3,4,5]
```

First steps

Glasgow Haskell Compiler

- The *Glasgow Haskell Compiler* (GHC) is the state-of-the-art open source implementation of Haskell
- The GHC if freely available for a range of operating systems from the Haskell home page http://www.haskell.org
- We recommand downloading the *Haskell Platform*
- Once installed, the interface GHCi system can be started from the terminal command prompt by simply typing `ghci`.

GHCi

```
λ> 1+2+3
6

λ> 1+2*3
7

λ> (1+2)*3
9

λ> 2-3+4
3

λ> 2*3/4
1.5
```

GHCi

```
λ> 2*pi
6.283185307179586

λ> (1 + sqrt 5) / 2
1.618033988749895

λ> log 2
0.6931471805599453
```

GHCi

```
λ> 2^3^4
2417851639229258349412352

λ> (2^3)^4
4096

λ> ceiling 2.6
3

λ> floor 2.6
2

λ> round 2.6
3

λ> (sin pi)^2 + (cos pi)^2
1.0
```

GHCi

```
λ> x = 42
λ> x+1
43

λ> let x = 42 in x+1
43
```

GHCi

```
λ > "Haskell rocks!"
"Haskell rocks!"

λ > "Haskell " ++ "rocks!"
"Haskell rocks!"

λ > "Haskell " <> "rocks!"
"Haskell rocks!"

λ > ['H','a','s','k','e','l','l',' ','r','o','c','k','s','!']
"Haskell rocks!"
```

GHCi

Command	Meaning
`:load` *name*	load script *name*
`:reload`	reload current script
`:set editor` *name*	set editor to *name*
`:edit` *name*	edit script *name*
`:edit`	edit current script
`:type` *expr*	show type of *expr*
`:?`	show all commands
`:quit`	quit GHCi
...	

GHCi

```
λ> :type 1
1 :: Num a => a

λ> :type 2.5
2.5 :: Fractional a => a

λ> :type 5/2
5/2 :: Fractional a => a

λ> :type 5 `div` 2
5 `div` 2 :: Integral a => a
```

GHCi

```
λ >  :type 1+2
1+2 :: Num a => a

λ >  :type (+)
(+) :: Num a => a -> a -> a

λ >  :type (1 +)
(1 +) :: Num a => a -> a

λ > :type (+ 1)
(+ 1) :: Num a => a -> a
```

GHCi

```
λ> :type 2.5
2.5 :: Fractional a => a

λ> :type 5/2
5/2 :: Fractional a => a

λ> :type (/)
(/) :: Fractional a => a -> a -> a

λ> :type (/ 2)
(/ 2) :: Fractional a => a -> a
```

GHCi

```
λ> :type pi
pi :: Floating a => a

λ> :type sqrt 2
sqrt 2 :: Floating a => a

λ> :type cos
cos :: Floating a => a -> a
```

GHCi

```
λ> fact n = if n == 0 then 1 else n * fact (n-1)

λ> :type fact
fact :: (Eq a, Num a) => a -> a

λ> fact 5
120

λ> fact 0
1

λ> fact 5.0
120.0

λ> fact 2.5
^CInterrupted.
```

GHCi

```
λ> f = fact

λ> :type fact
fact :: (Eq a, Num a) => a -> a

λ> f 5
120

λ> f (f 3)
720
```

GHCi

```
λ> 'a'
'a'

λ> :type 'a'
'a' :: Char

λ> 'abc'
<interactive>: error:
    o Syntax error on 'abc'

λ> 'a':"bc"
"abc"
```

GHCi

```
λ> "abc"
"abc"

λ> :type "abc"
"abc" :: String

λ> "abc" ++ "def"
"abcdef"

λ> :type (++)
(++) :: [a] -> [a] -> [a]
```

Types and classes

Basic concepts

- In Haskell every expression must have a type.
- A type is a collection of related values.
- We use the notation `v :: T` to mean that `v` is a value in the type `T`.

Example

```
True  :: Bool
False :: Bool
not   :: Bool -> Bool
(&&)  :: Bool -> Bool -> Bool
(||)  :: Bool -> Bool -> Bool
```

Basic types

- `Bool` - Logical values.
- `Char` - Single characters.
- `String` - Strings of characters.
- `Int` - Fixed-precision integers.
- `Integers` - Arbitrary-precision integers.
- `Float` - Since-precision floating-point numbers.
- `Double` - Double-precision floating-point numbers.

List types

- A list is a sequence of elements of the same type, with the elements being enclosed in square parentheses and separated by commas.
- We write `[T]` for the type of all lists whose elements have type `T`.
- The number of elements in a list is called its length.
- The list `[]` of length zero is called the empty list.
- `[]` and `[[]]` (and `[[[]]]`, `[[[[]]]]`, ...) are different lists.

List types

```
λ> :type []
[] :: [a]

λ> :type [1,2,3,4,5]
[1,2,3,4,5] :: Num a => [a]

λ> :type ['a', 'b', 'c', 'd']
['a', 'b', 'c', 'd'] :: [Char]

λ> :type ["ab", "cd", "ef", "gh"]
["ab", "cd", "ef", "gh"] :: [String]
```

List types

```
λ> :type [cos, sin]
[cos, sin] :: Floating a => [a -> a]

λ> :type [1, 'a']
<interactive>: error:
    o No instance for (Num Char) arising from the literal '1'

λ> :type [[1],[2,3],[4,5,6]]
[[1],[2,3],[4,5,6]] :: Num a => [[a]]

λ> :type [[[1]],[[2,3],[4,5,6]]]
[[[1]],[[2,3],[4,5,6]]] :: Num a => [[[a]]]
```

Tuple types

- A tuple is a sequence of components of possibly different types, with the components being enclosed in round parentheses and separated by commas.
- We write `(T1, T2, ..., Tn)` for the type of all tuples whose i-th component have type `Ti` for any $1 \leqslant i \leqslant n$.
- The number of elements in a tuple is called its arity.
- The tuple `()` of arity zero is called the empty tuple.
- Tuple of arity one are not permitted.

Tuple types

```
λ> :type ()
() :: ()

λ> :type (1,'a')
(1,'a') :: Num a => (a, Char)

λ> :type (1,2,'a',"abc")
(1,2,'a',"abc") :: (Num a, Num b) => (a, b, Char, String)

λ> :type (sqrt, 'a')
(sqrt, 'a') :: Floating a => (a -> a, Char)

λ> :type (1, ('a', "cd"))
(1, ('a', "cd")) :: Num a => (a, (Char, String))
```

Tuple types

```
λ> :type (1, ('a', "cd"))
(1, ('a', "cd")) :: Num a => (a, (Char, String))

λ> :type (1, [cos, sin])
(1, [cos, sin]) :: (Floating a1, Num a2) => (a2, [a1 -> a1])

λ> :type (1)
(1) :: Num a => a

λ> let t = (1,2) in (t, 3)
((1,2),3)

λ> let t = (1,t)
<interactive>: error:
  o Couldn't match expected type 'b' with actual type '(a, b)'
```

Function types

- A function is a mapping of one type to results of another type.
- We write `T1 -> T2` for the type of all functions that map arguments of type `T1` to results of type `T2`.
- There is no restriction that function must be total on their argument type.

Function types

```
λ> :type not
not :: Bool -> Bool

λ> :type even -- :type odd
even :: Integral a => a -> Bool

λ> :type mod
mod :: Integral a => a -> a -> a

λ> add x y = x+y
λ> :type add
add :: Num a => a -> a -> a

λ> add' (x,y) = x+y
λ> :type add'
add' :: Num a => (a, a) -> a
```

Curried functions

- Currying is the process of transforming a function that takes multiple arguments in a tuple as its argument, into a function that takes just a single argument and returns another function which accepts further arguments, one by one, that the original function would receive in the rest of that tuple.
- The function arrow `->` in type is assumed to associate to the right.

  The type

  `T1 -> T2 -> T3 -> ... -> Tn`

  means

  `T1 -> (T2 -> (T3 -> ( ... -> Tn)...))`

Curried functions

The type

`a1 -> a2 -> a3`

means

`a1 -> (a2 -> a3)`

Curried functions

The type

```
a1 -> a2 -> a3 -> a4
```

means

```
a1 -> (a2 -> (a3 -> a4))
```

Curried functions

The type

```
a1 -> a2 -> a3 -> a4 -> a5
```

means

```
a1 -> (a2 -> (a3 -> (a4 -> a5)))
```

Curried functions

Multiplying three integers

```
--  mult :: Int -> (Int -> (Int -> Int))
mult :: Int -> Int -> Int -> Int
mult x y z = x*y*z
```

Curried functions

Multiplying three integers

```
--  mult :: Int -> (Int -> (Int -> Int))
mult :: Int -> Int -> Int -> Int
mult x y z = x*y*z
```

```
λ> mult 2 3 4
24

λ> :type mult 2
mult 2 :: Int -> Int -> Int

λ> :type mult 2 3
mult 2 3 :: Int -> Int

λ> :type mult 2 3 4
mult 2 3 4 :: Int
```

Curried functions

Multiplying three integers

```haskell
--  mult :: Int -> (Int -> (Int -> Int))
mult :: Int -> Int -> Int -> Int
mult x y z = x*y*z
```

```
λ > mult2 = mult  2
λ > mult3 = mult2 3
λ > mult3 4
24
```

```
λ > :type mult2
mult2 :: Int -> Int -> Int
```

```
λ > :type mult3
mult3 :: Int -> Int
```

Polymorphic types

- Parametric polymorphism refers to when the type of a value contains one or more (unconstrained) type variables, so that the value may adopt any type that results from substituting those variables with concrete types.
- For example, the function `id :: a -> a` contains an unconstrained type variable `a` in its type, and so can be used in a context requiring `Char -> Char` or `Integer -> Integer` or `(Bool -> Bool) -> (Bool -> Bool)` or any of a literally infinite list of other possibilities.
- The empty list `[] :: [a]` belongs to every list type.

Polymorphic types

```
λ> length []
0

λ> length [1,3,5,7,2,4,6,8]
8

λ> length ["Huey","Dewey","Louie"]
3

λ> length [sin, cos, tan]
3
```

Polymorphic types

```
λ> :type length
length :: Foldable t => t a -> Int

λ> :info length
type Foldable :: (* -> *) -> Constraint
class Foldable t where
  length :: t a -> Int
  ...
  -- Defined in 'Data.Foldable'
```

Overloaded types

- A type that contains one or more class constraints is called overloaded.
- Class constraints are written in the form `C a`, where `C` is the name of the class and `a` is a type variable.

Overloaded types

```
λ> 1 + 2
3

λ> :type 1
1 :: Num a => a

λ> :type 1 + 2
1 + 2 :: Num a => a
```

```
λ> 1.0 + 2.0
3.0

λ> :type 1.0
1.0 :: Fractional a => a

λ> :type 1.0 + 2.0
1.0 + 2.0 :: Fractional a => a
```

```
λ> sqrt 2 + sqrt 3
3.1462643699419726

λ> :type sqrt 2
sqrt 2 :: Floating a => a

λ> :type sqrt 2 + sqrt 3
sqrt 2 + sqrt 3 :: Floating a => a
```

Overloaded types

```
λ> :type (+)
(+) :: Num a => a -> a -> a

λ> :type (-)
(-) :: Num a => a -> a -> a

λ> :type (*)
(*) :: Num a => a -> a -> a

λ> :type (/)
(/) :: Fractional a => a -> a -> a

λ> :type sqrt
sqrt :: Floating a => a -> a
```

Basic classes

- A class is collection of types that support certain overloaded operations called methods.
- Haskell provides a number of basic classes that are built-in to the language.

Eq – Equality types

This class contains types whose values can be compared for equality and inequality using the following two methods:

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

All the basic types `Bool`, `Char`, `String`, `Int`, `Integers`, `Float` and `Double` are instances of the `Eq` class.

Basic classes

`Eq` – Equality types

```
λ> True == True
True

λ> 'a' == 'b'
False

λ> "abc" == "abc"
True

λ> 2.5 == 5.2
False
```

Basic classes

Eq – Equality types

```
λ> ('a', 1) == ('b', 1)
False

λ> (1, 2, 3) == (1, 2)
<interactive>:120:14: error:
  o Couldn't match expected type: (a0, b0, c0)
    with actual type: (a1, b1)

λ> [1,2,3] == [1,2,3,4]
False

λ> cos == cos
<interactive>: error:
  o No instance for (Eq (Double -> Double))
    arising from a use of '=='
```

`Ord` – Ordered types

This class contains types that are instances of the equality class `Eq`, but in addition these values are totally ordered, and as such can be compared using the following six methods:

```
(<)  :: a -> a -> Bool
(<=) :: a -> a -> Bool
(>)  :: a -> a -> Bool
(>=) :: a -> a -> Bool
min  :: a -> a -> a
max  :: a -> a -> a
```

All the basic types `Bool`, `Char`, `String`, `Int`, `Integers`, `Float` and `Double` are instances of the `Ord` class.

Basic classes

Ord – Ordered types

```
λ> False < True
True

λ> "elegant" < "elephant"
True

λ> "a" < "ab"
True

λ> 'b' > 'a'
True

λ> [1,2,3] <= [1,2]
False

λ> [] < [1]
True
```

Basic classes

`Ord` – Ordered types

```
λ> (1,2) < (1,3)
True

λ> (1,2,3) < (1,1)
<interactive>: error:
  o Couldn't match expected type: (a0, b0, c0)
    with actual type: (a1, b1)

λ> [True] < [False,False]
False

λ> (False,False) <= (False,True)
True
```

Ord – Ordered types

```
λ>

λ> min ('a',2) ('a',1)
('a',1)

λ> max ('a',2) ('a',1)
('a',2)

λ> sin < cos
<interactive>: error:
  o No instance for (Ord (Double -> Double))
    arising from a use of '<'

λ> (1, sin) > (2, cos)
<interactive>: error:
  o No instance for (Ord (Double -> Double))
    arising from a use of '>'
```

Basic classes

`Show` – **Showable types**

This class contains types that can be converted into strings of characters using the following method:

```
show :: a -> String
```

All the basic types `Bool`, `Char`, `String`, `Int`, `Integers`, `Float` and `Double` are instances of the `Show` class.

Basic classes

Show – Showable types

```
λ> show True
"True"

λ> show 'a'
"'a'"

λ> show "abc"
"\"abc\""

λ> show [1,2,3]
"[1,2,3]"

λ> show (1, True, [1,2,3])
"(1,True,[1,2,3])"
```

Basic classes

`Read` – **Readable types**

This class is dual to `Read` and contains types whose values can be converted from string of characters using the following method:

```
read :: String -> a
```

All the basic types `Bool`, `Char`, `String`, `Int`, `Integers`, `Float` and `Double` are instances of the `Read` class.

Basic classes

Read – Readable types

```
λ> read "False" :: Bool
False

λ> read "'a'" :: Char
'a'

λ> read "\"abc\"" :: String
"abc"

λ> read "[1,2,3]" :: [Int]
[1,2,3]

λ> read "(1, True, [1,2,3])" :: (Int, Bool, [Int])
(1,True,[1,2,3])
```

`Num` – **Numeric types**

This class contains types whose values are numeric, and as such can be processed using the following six methods:

```
(+) :: a -> a -> a
(-) :: a -> a -> a
(*) :: a -> a -> a
negate :: a -> a
abs    :: a -> a
signum :: a -> a
```

Note that the `Num` class does not provide a division method.

Basic classes

`Num` – Numeric types

```
λ> 1+2
3

λ> 1-2
-1

λ> 1.0+2.0
3.0

λ> 2*3
6

λ> 2.0*3.0
6.0
```

Basic classes

Num – Numeric types

```
λ> negate 3.0
-3.0

λ> negate (-2)
2

λ> abs(-1.5)
1.5

λ> signum 3
1

λ> signum (-3)
-1
```

`Integral` – **Integral types**

This class contains types that are instances of the numeric class `Num`, but in addition whose values are integers, and as such support the method of integer division and integer remainder:

```
div :: a -> a -> a
mod :: a -> a -> a
```

Basic classes

`Integral` – Integral types

```
λ> div 7 2
3

λ> 7 `div` 2
3

λ> 8 `div` 2
4

λ> 7 `mod` 2
1

λ> 8 `mod` 2
0
```

`Integral` – **Integral types**

```
λ> (-7) `div` 2
-4

λ> (-7) `div` (-2)
3

λ> (-7) `mod` 2
1

λ> (-7) `mod` (-2)
-1
```

`Fractional` – **Fractional types**

This class contains types that are instances of the numeric class `Num`, but in addition whose values are non-integral, and as such support the method of integer fractional division and fractional reciprocation:

```
(/) :: a -> a -> a
recip :: a -> a -> a
```

The basic types `Float` and `Double` are instances of the `Fractional` class.

`Fractional` – **Fractional types**

```
λ> 7.0 / 2.0
3.5

λ> 2.0 / 7.0
0.2857142857142857

λ> recip 2.0
0.5

λ> recip 1.0
1.0
```

Defining functions

New from old

```haskell
even :: Integral a => a -> Bool
even n = n `mod` 2 == 0

odd  :: Integral a => a -> Bool
odd n = n `mod` 2 /= 0

recip :: Fractional a => a -> a
recip n = 1 / n
```

Conditional expressions

For processing conditions, the `if-then-else` syntax was defined in Haskell98.

```
if <condition> then <true-value> else <false-value>
```

`if` is an expression (which is converted to a value) and not a statement (which is executed) as in many imperative languages. As a consequence, the `else` is mandatory in Haskell. Since `if` is an expression, it must evaluate to a result whether the condition is true or false, and the `else` ensures this.

Conditional expressions

```haskell
abs :: Int -> Int
abs n = if n >= 0 then n else -n

signum :: Int -> Int
signum n = if n < 0 then -1 else
              if n == 0 then 0 else 1

describeLetter :: Char -> String
describeLetter c = if c >= 'a' && c <= 'z'
                   then "Lower case"
                   else if c >= 'A' && c <= 'Z'
                        then "Upper case"
                        else "Not an ASCII letter"
```

Conditional expressions

```
addOneIfEven1 :: Integral a => a -> a
addOneIfEven1 n = if even n then n+1 else n

addOneIfEven2 :: Integral a => a -> a
addOneIfEven2 n = n + if even n then 1 else 0

addOneIfEven3 :: Integral a => a -> a
addOneIfEven3 n = (if even n then (+ 1) else (+ 0)) n

addOneIfEven4 :: Integral a => a -> a
addOneIfEven4 n = (if even n then (+ 1) else id) n
```

Conditional expressions

Remember that

```
isNullLength :: Foldable t => t a -> Bool
isNullLength xs = if length xs == 0 then True else False
```

is nothing but

```
isNullLength :: Foldable t => t a -> Bool
isNullLength xs = length xs == 0
```

or (as we we shall see soon ... but not really better here!)

```
isNullLength :: Foldable t => t a -> Bool
isNullLength = (== 0) . length
```

Guarded expressions

As an alternative to using conditional expressions, functions can also be defined using guarded expressions, in which a sequence of logical expressions called guards is used to choose between a sequence of results of the same type.

- If the first guard is `True`, then the first result is chosen.
- Otherwise, if the second guard is `True`, then the second result is chosen.
- And so on.

Guarded expressions

```haskell
abs1 :: Int -> Int
abs1 n = if n >= 0 then n else -n

abs2 :: Int -> Int
abs2 n
  | n >= 0    = n
  | otherwise = -n
```

Guarded expressions

```haskell
signum1 :: Int -> Int
signum1 n = if n < 0 then -1 else
              if n == 0 then 0 else 1

signum2 :: Int -> Int
signum2 n
  | n < 0     = -1
  | n == 0    = 0
  | otherwise = 1
```

Guarded expressions

```haskell
describeLetter1 :: Char -> String
describeLetter1 c = if c >= 'a' && c <= 'z'
                  then "Lower case"
                  else if c >= 'A' && c <= 'Z'
                       then "Upper case"
                       else "Not an ASCII letter"

describeLetter2 :: Char -> String
describeLetter2 c
  | c >= 'a' && c <= 'z' = "Lower case"
  | c >= 'A' && c <= 'Z' = "Upper case"
  | otherwise            = "Not an ASCII letter"
```

Guarded expressions

```
fact :: (Eq t, Num t) => t -> t
fact n
  | n == 0    = 1
  | otherwise = n * fact (n-1)

mult :: (Eq t, Num t, Num a) => a -> t -> a
mult n m
  | m == 0    = 0
  | otherwise = n + mult n (m - 1)
```

Pattern matching

Many functions have a simple and intuitive definition using pattern matching, in which a sequence of syntactic expressions called patterns is used to choose between a sequence of results of the same type.

The wildcard pattern `_` matches any value.

- If the first pattern is matched, then the first result is chosen.
- Otherwise, if the second pattern is matched, then the second result is chosen.
- And so on...

Pattern matching

```haskell
-- conditional expression
not :: Bool -> Bool
not b = if b == True then False else True

-- guarded function
not :: Bool -> Bool
not b
  | b == True = False
  | otherwise = True

-- pattern matching
not :: Bool -> Bool
not False = True
not True = False
```

Pattern matching

```haskell
(&&) :: Bool -> Bool -> Bool
True  && True  = True
True  && False = False
False && True  = False
False && False = False

(&&) :: Bool -> Bool -> Bool
True && True  = True
_    && _     = False

(&&) :: Bool -> Bool -> Bool
True  && b = b
False && _ = False
```

Pattern matching

```haskell
guess :: Int -> String
guess 0 = "I am zero"
guess 1 = "I am one"
guess 2 = "I am two"
guess _ = "I am at least three"

-- be careful with the wildcard pattern
guess :: Int -> String
guess _ = "I am at least three"
guess 0 = "I am zero"
guess 1 = "I am one"
guess 2 = "I am two"
```

Pattern matching – Tuple patterns

A tuple of patterns is itself a pattern, which matches any tuple of the same arity whose components all match the corresponding patterns in order.

```
first :: (a, b, c) -> a
first (x, _, _) = x

second :: (a, b, c) -> b
second (_, y, _) = y

third :: (a, b, c) -> c
third (_, _, z) = z
```

Pattern matching – Tuple patterns

A tuple of patterns is itself a pattern, which matches any tuple of the same arity whose components all match the corresponding patterns in order.

Functions `fst` and `snd` are defined in the module `Data.Tuple`:

```
λ> :type fst
fst :: (a, b) -> a
λ> fst (1,2)
1

λ> :type snd
snd :: (a, b) -> b
λ> snd (1,2)
2
```

Pattern matching – List patterns

A list of patterns is itself a pattern, which matches any list of the same length whose components all match the corresponding patterns in order.

```
-- three characters beginning with the letter 'a'
test :: [Char] -> Bool
test ['a',_,_] = True
test _         = False

-- four characters ending with the letter 'z'
test :: [Char] -> Bool
test [_,_,_,'z'] = True
test _         = False
```

Pattern matching – Lambda expression

- An anonymous function is a function without a name.
- It is a Lambda abstraction and might look like this:
  `\x -> x + 1`.

  (That backslash is Haskell's way of expressing a λ and is supposed to look like a Lambda.)

```
λ>  :type (\x -> x+1)
(\x -> x+1) :: Num a => a -> a

λ>  (\x -> x+1) 2
3
```

Pattern matching – Lambda expression

The definition

```
add :: Int -> Int -> Int -> Int
add x y z = x+y+z
```

can be understood as meaning

```
add :: Int -> Int -> Int -> Int
add = \x -> (\y -> (\z -> x+y+z))
```

which makes precise that `add` is a function that takes an integer `x` and returns a function which in turn takes another integer `y` and returns a function which in turn takes another integer `z` and returns the result `x+y+z`.

Pattern matching – Lambda expression

λ-expressions are useful when defining functions that returns function as results by their very nature, rather than a consequence of currying.

```
const :: a -> b -> a
const x _ = x

-- emphasis const :: a -> (b -> a)
const :: a -> b -> a
const x = \_ -> x
```

Pattern matching – Lambda expression

A closure (the opposite of a combinator) is a function that makes use of free variables in its definition. It closes around some portion of its environment.

```
f :: Num a => a -> a -> a
f x = \y -> x + y
```

`f` returns a closure, because the variable `x`, which is bounded outside of the lambda abstraction is used inside its definition.

```
λ> g = f 1
λ> g 2
3

λ> g 3
4
```

Pattern matching – Operator sections

- Functions such as + that are written between their two arguments are called section
- Any operator can be converted into a curried function by enclosing the name of the operator in parentheses, such as `(+) 1 2`.
- More generally, if o is an operator, then expression of the form `(o)`, `(x o)` and `(o y)` are called sections whose meaning as functions can be formalised using λ-expressions as follows:

```
(o)   = \x -> (\y -> x o y))
(x o) = \y -> x o y
(o y) = \x -> x o y
```

Pattern matching – Operator sections

- `(+)` is the addition function `\x -> (\y -> x+y)`.
- `(1 +)` is the successor function `\y -> 1+y`.
- `(1 /)` is the reciprocation function `\y -> 1/y`.
- `(* 2)` is the doubling function `\x -> x*2`.
- `(/ 2)` is the halving function `\x -> x/2`.

Pattern matching – Bindings

- A `where` clause is used to divide the more complex logic or calculation into smaller parts, which makes the logic or calculation easy to understand and handle
- A `where` clause is bound to a surrounding syntactic construct, like the pattern matching line of a function definition.
- A `where` clause is a syntactic construct

Pattern matching – Bindings

```haskell
bmiTell :: (RealFloat a) => a -> a -> String
bmiTell weight height
  | weight / height ^ 2 <= 18.5 = "Underweight"
  | weight / height ^ 2 <  25.0 = "Healthy weight"
  | weight / height ^ 2 <  30.0 = "Overweight"
  | otherwise                   = "Obese"

bmiTell :: (RealFloat a) => a -> a -> String
bmiTell weight height
  | bmi <= 18.5 = "Underweight"
  | bmi < 25.0  = "Healthy weight"
  | bmi < 30.0  = "Overweight"
  | otherwise   = "Obese"
  where
    bmi = weight / height ^ 2
```

Pattern matching – Bindings

```haskell
bmiTell :: (RealFloat a) => a -> a -> String
bmiTell weight height
  | bmi <= underweight = "Underweight"
  | bmi <  healthy     = "Healthy weight"
  | bmi <  overweight  = "Overweight"
  | otherwise          = "Obese"
  where
    bmi = weight / height ^ 2
    underweight = 18.5
    healthy     = 25
    overweight  = 30
```

Pattern matching – Bindings

- A let binding binds variables anywhere and is an expression itself, but its scope is tied to where the let expression appears.
- if a let binding is defined within a guard, its scope is local and it will not be available for another guard.
- A let binding can take global scope overall pattern-matching clauses of a function definition if it is defined at that level.

Pattern matching – Bindings

```haskell
cylinder :: (RealFloat a) => a -> a -> a
cylinder r h =
  let sideArea = 2 * pi * r * h
      topArea = pi * r ^2
  in  sideArea + 2*topArea
```

Pattern matching – Bindings

```
λ> let zoot x y z = x*y + z
λ> :type zoot
zoot :: Num a => a -> a -> a -> a

λ> zoot 3 9 2
29

λ> let boot x y z = x*y + z in boot 3 9 2
29

λ> :type boot
<interactive>: error:
  o Variable not in scope: boot
```

Pattern matching – Bindings

```
λ> let a = 1; b = 2 in a + b
3

λ> let a = 1; b = a + 2 in a + b
4

λ> let a = 1; a = 2 in a
<interactive>:: error:
    Conflicting definitions for 'a'

λ> let a = 1; b = 2+a; c = 3+a+b in (a, b, c)
(1,3,7)
```

Pattern matching – Bindings

```
λ> let a = 1 in let a = 2; b = 3+a in b
5

λ> let a = 1 in let a = a+2 in let b = 3+a in b
^CInterrupted.

λ> let f x y = x+y+1 in f 3 5
9

λ> let f x y = x+y; g x = f x (x+1) in g 5
11
```

Pattern matching – Bindings

```
dist :: Floating a => (a, a) -> (a, a) -> a
dist (x1,y1) (x2,y2) =
      let xdist = x2 - x1
          ydist = y2 - y1
          sqr z = z*z
      in sqrt ((sqr xdist) + (sqr ydist))

dist :: Floating a => (a, a) -> (a, a) -> a
dist (x1,y1) (x2,y2) = sqrt((sqr xdist) + (sqr ydist))
  where
    xdist = x2 - x1
    ydist = y2 - y1
    sqr z = z*z
```

Pattern matching – Bindings

We can pattern match with `let` bindings. E.g., we can dismantle a tuple into components and bind the components to names.

```
λ> f x y z = let (sx,sy,sz) = (x*x,y*y,z*z) in (sx,sy,sz)
λ> f 1 2 3
(1,4,9)

λ> g x y = let (sx,_) = (x*x,y*y) in sx
λ> g 2 3
4

λ> h x = let ((sx,cx),qx) = ((x*x,x*x*x),x*x*x*x) in (sx,cx,qx)
λ> h 2
(4,8,16)
```

Pattern matching – Bindings

`let` bindings are expressions.

```
λ> 1 + let x = 2 in x*x
5

λ> (let x = 2 in x*x) + 1
5

λ> (let (x,y,z) = (1,2,3) in x+y+z) * 100
600

λ> (let x = 2 in (+ x)) 3
5

λ> let x=3 in x*x + let x=4 in x*x
25
```

Some functions

Double factorial

The double factorial (or semifactorial of a number n, denoted by $n!!$, is the product of all the integers from 1 up to n that have the same parity (odd or even) as n

Double factorial

The double factorial (or semifactorial of a number n, denoted by $n!!$, is the product of all the integers from 1 up to n that have the same parity (odd or even) as n

```
dblFact1 :: Integral a => a -> a
dblFact1 n = go n
  where
    p m = (even n && even m) || (odd n && odd m)
    go 0 = 1
    go m
      | p m       = m * go (m-1)
      | otherwise = go (m-1)
```

Double factorial

The double factorial (or semifactorial of a number n, denoted by $n!!$, is the product of all the integers from 1 up to n that have the same parity (odd or even) as n

```
dblFact2 :: Integral a => a -> a
dblFact2 n = go n
  where
    nParity2 = n `mod` 2
    p m = m `mod` 2 == nParity2
    go 0 = 1
    go m
      | p m       = m * go (m-1)
      | otherwise = go (m-1)
```

Double factorial

The double factorial (or semifactorial of a number n, denoted by $n!!$, is the product of all the integers from 1 up to n that have the same parity (odd or even) as n

```
dblFact3 :: (Eq a, Num a) => a -> a
dblFact3 0 = 1
dblFact3 1 = 1
dblFact3 n = n * dblFact3 (n-2)
```

Double factorial

The double factorial (or semifactorial of a number n, denoted by $n!!$, is the product of all the integers from 1 up to n that have the same parity (odd or even) as n

```
dblFact4 :: (Num a, Enum a) => a -> a
dblFact4 n = product [n,n-2..1]
```

Collatz conjecture

The Collatz conjecture is one of the most famous unsolved problems in mathematics. It concerns sequences of integers in which each term is obtained from the previous term as follows:

$$u_n = \begin{cases} u_{n-1}/2 & \text{if } u_{n-1} \text{ is even} \\ 3u_{n-1}+1 & \text{if } u_{n-1} \text{ is odd} \end{cases}$$

For instance, starting with $n = 19$, one gets the sequence $19, 58, 29, 88, 44, 22, 11, 34, 17, 52, 26, 13, 40, 20, 10, 5, 16, 8, 4, 2, 1.$

Collatz conjecture

The Collatz conjecture is one of the most famous unsolved problems in mathematics. It concerns sequences of integers in which each term is obtained from the previous term as follows:

$$u_n = \begin{cases} u_{n-1}/2 & \text{if } u_{n-1} \text{ is even} \\ 3u_{n-1}+1 & \text{if } u_{n-1} \text{ is odd} \end{cases}$$

For instance, starting with $n = 19$, one gets the sequence $19, 58, 29, 88, 44, 22, 11, 34, 17, 52, 26, 13, 40, 20, 10, 5, 16, 8, 4, 2, 1$.

```
collatz1 1 = "win"
collatz1 n = collatz1 (if even n
                       then n `div` 2
                       b
                       else 3*n + 1)
```

Collatz conjecture

The Collatz conjecture is one of the most famous unsolved problems in mathematics. It concerns sequences of integers in which each term is obtained from the previous term as follows:

$$u_n = \begin{cases} u_{n-1}/2 & \text{if } u_{n-1} \text{ is even} \\ 3u_{n-1}+1 & \text{if } u_{n-1} \text{ is odd} \end{cases}$$

For instance, starting with $n = 19$, one gets the sequence $19, 58, 29, 88, 44, 22, 11, 34, 17, 52, 26, 13, 40, 20, 10, 5, 16, 8, 4, 2, 1.$

```
collatz2 :: Integral a => a -> String
collatz2 1 = "win"
collatz2 n
  | even n    = collatz2 (n `div` 2)
  | otherwise = collatz2 (3*n + 1)
```

Ackermann–Péter function

$$
\begin{aligned}
A(0, n) &= n + 1 \\
A(m + 1, 0) &= A(m, 1) \\
A(m + 1, n + 1) &= A(m, A(m + 1, n))
\end{aligned}
$$

```
aP :: (Num a, Eq a, Num b, Eq b) => a -> b -> b
aP 0 n = n+1
aP m 0 = aP (m-1) 1
aP m n = aP (m-1) (aP m (n-1))
```

Prime numbers

A prime number (or a prime) is a natural number greater than 1 that is not a product of two smaller natural numbers.

```haskell
-- very naive
isPrime :: Integral a => a -> Bool
isPrime 0 = False
isPrime 1 = False
isPrime n = go 2
  where
    go k
      | k >= n    = True
      | otherwise = n `mod` k /= 0 && go (k+1)
```

Ping-pong programming

```haskell
-- odd number predicate
isOdd :: (Eq a, Num a) => a -> Bool
isOdd 0 = False
isOdd 1 = True
isOdd n = isEven (n-1)

-- even number predicate
isEven :: (Eq a, Num a) => a -> Bool
isEven 0 = True
isEven 1 = False
isEven n = isOdd (n-1)
```

Factorial

```haskell
fact1 :: (Eq a, Num a) => a -> a
fact1 n = if n == 0 then 1 else n * fact1 (n-1)

fact2 :: (Eq a, Num a) => a -> a
fact2 n
  | n == 0 = 1
  | otherwise = n * fact2 (n-1)
```

Factorial

```haskell
fact3 :: (Ord a, Num a) => a -> a
fact3 = go 1
  where
    go m n
      | m > n     = 1
      | otherwise = m * go (m+1) n

fact4 :: (Eq t, Num t) => t -> t
fact4 n = go 1 n
  where
    go acc 0 = acc
    go acc m = go (acc*m) (m-1)
```

Factorial

```haskell
fact5 :: (Enum a, Num a) => a -> a
fact5 n = product [1..n]
```

Pascal triangle

							1							
						1		1						
					1		2		1					
				1		3		3		1				
			1		4		6		4		1			
		1		5		10		10		5		1		
	1		6		15		20		15		6		1	
1		7		21		35		35		21		7		1

```haskell
pT :: (Num a, Ord a, Num b) => a -> a -> b
pT r c
  | r == 1 && c == 1 = 1
  | c < 1  || c > r  = 0
  | otherwise        = pT (r-1) (c-1) + pT (r-1) c
```