Functional programming
Lecture 01 — First steps

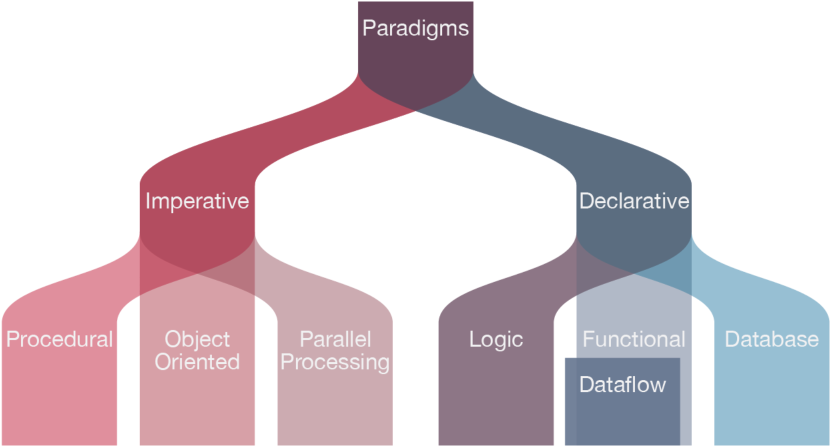
(version: 2026-02-26–11:52:20)

Stéphane Vialette
stephane.vialette@univ-eiffel.fr

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

Indroduction

Genealogy of programming languages



Main functional languages

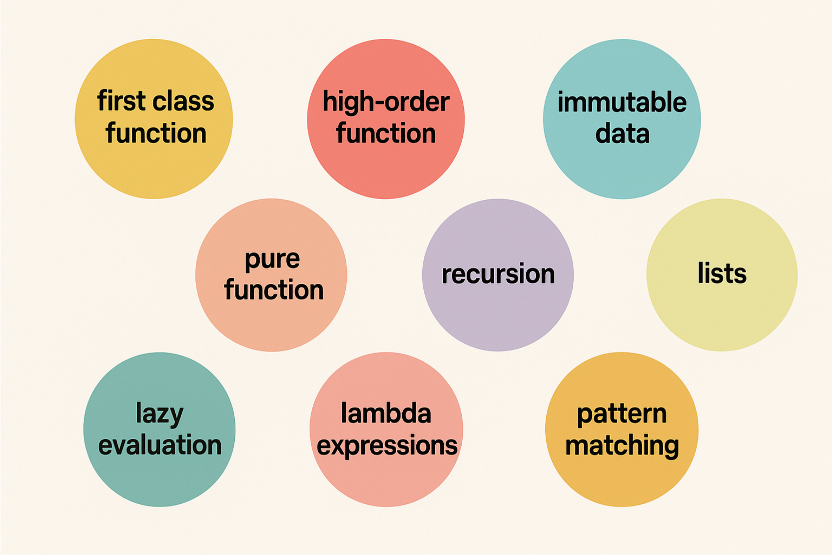
- Lisp, Common Lisp, Scheme, Racket, . . .
- Erlang, Elixir, . . .
- ML, Standard ML, Ocaml, F#, . . .
- Clojure, Scala, . . .
- Haskell, Elm, Miranda, Idris, Agda, . . .

Haskell

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.

Characteristics of functional programming (haskell)



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.

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.

Haskell landscape

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.

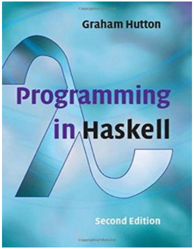
Dynamic Analysis Tools

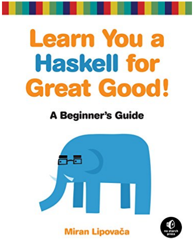
- criterion: powerful benchmarking framework.
- hpc: check evaluation coverage of a haskell program, useful for determining test coverage.

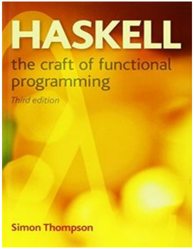
IDEs

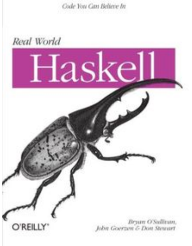
- VSCodium
- IntelliJ
- Vim
- GNU Emacs
- Haskell for Mac (commercial)
- Sublime Text (free/commercial)

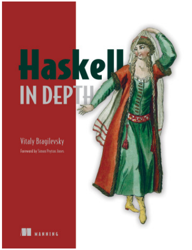
Haskell books

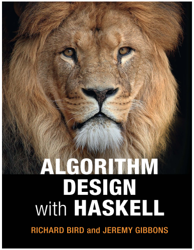


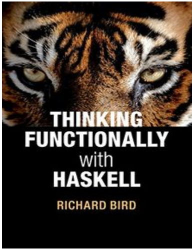


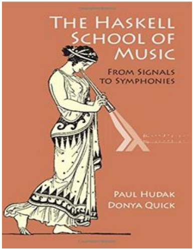




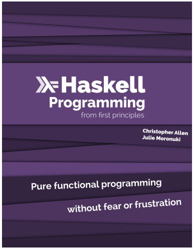


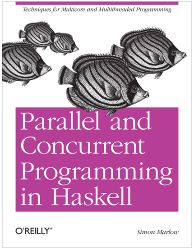


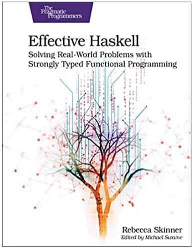


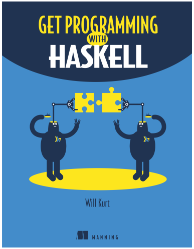


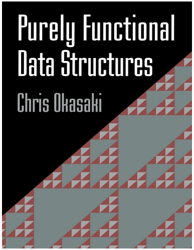
Haskell books

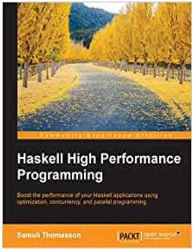


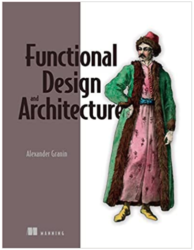


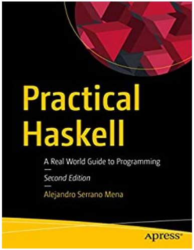




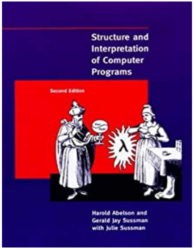


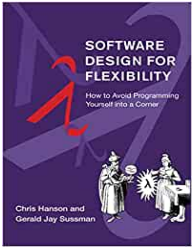


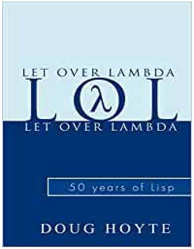


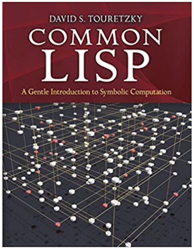


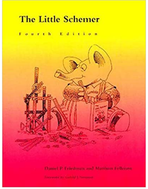
Functional programming books

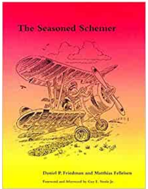


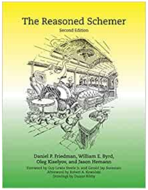


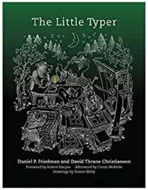


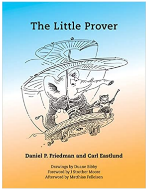




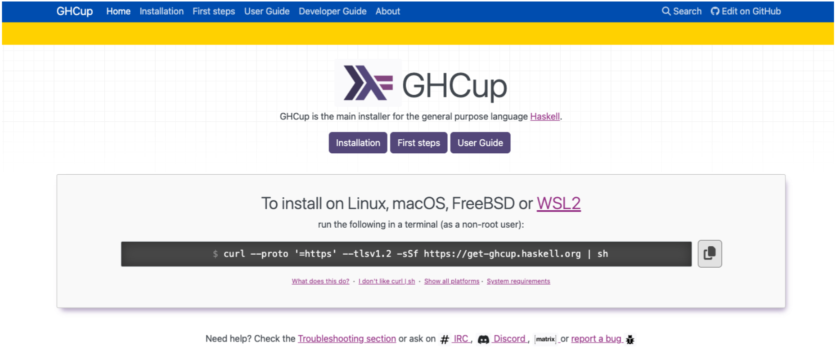








Install and manage the Haskell toolchain with GHCup



Haskell

Haskell can be used both as a compiled language and through an interpreter.

Programs can be compiled into efficient executables using the Glasgow Haskell Compiler (GHC), which ensures strong type checking and high performance.

At the same time, Haskell offers an interactive environment called GHCi (the Glasgow Haskell Compiler interactive), which acts as an interpreter.

With GHCi, developers can quickly test code snippets, experiment with functions, and explore ideas without compiling an entire program. This dual approach makes Haskell both practical for rapid prototyping and powerful for building production-ready applications.

A taste of haskell : Multiplying elements

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

A taste of haskell : Multiplying elements

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

```
λ> fact 0
1
λ> fact 1
1
λ> fact 3
6
λ> fact 5
120
λ> fact 40
815915283247897734345611269596115894272000000000
```

A taste of haskell : Multiplying elements

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

```
  fact 3
=   { applying function fact }
  3 * fact 2
=   { applying function fact }
  3 * 2 * fact 1
=   { applying function fact }
  3 * 2 * 1 * fact 0
=   { applying function fact }
  3 * 2 * 1 * 1
=   { applying function (+)  }
  6 * 1 * 1
=   { applying function (+)  }
  6 * 1
=   { applying function (+)  }
  6
```

A taste of haskell : Summing elements

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

A taste of haskell : Summing elements

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

```
λ > sum []
0
λ > sum [1]
1
λ > sum [1,2,3,4,5]
15
λ > sum [sum [1,2],sum [3,4], 5]
15
λ > sum [1,2] + sum [sum [3,4],5]
15
```

A taste of haskell : Summing elements

```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 (+) }
  3 + 3 + 0
=   { applying function (+) }
  6 + 0
=   { applying function (+) }
  6
```

A taste of haskell : Sorting lists

```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 : Sorting lists

```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]
```

```
λ> qSort []
[]
λ> qSort [1]
[1]
λ> qSort [1,2,3,4,5]
[1,2,3,4,5]
λ> qSort [4,1,3,5,2]
[1,2,3,4,5]
λ> qSort [4,-1,3,5,-2]
[-2,-1,3,4,5]
```

A taste of haskell : Sorting lists

```
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 []
=   { applying function qSort }
  [] ++ [x] ++ []
=   { applying function ++ (twice) }
  [x]
```

A taste of haskell : Sorting lists

```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 (twice) }
  (qSort [] ++ [1] ++ qSort [2]) ++ [3] ++ (qSort [4] ++ [5] ++ qSort [])
=   { applying function qSort (four times) }
  ([] ++ [1] ++ [2]) ++ [3] ++ ([4] ++ [5] ++ [])
=   { applying function ++ (four times) }
  [1,2] ++ [3] ++ [4,5]
=   { applying function ++ (twice) }
  [1,2,3,4,5]
```

First steps

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)
-5
λ> 2 * 3 / 4
1.5
```

GHCi

```
λ> 2 * pi
6.283185307179586
λ> (1 + sqrt 5) / 2
1.618033988749895
λ> log 2
0.6931471805599453
λ> abs (-3)
3
```

GHCi

```
λ> 2^3^4 -- == 2^(3^4)
2417851639229258349412352
λ> (2^3)^4
4096
λ> ceiling 2.6 -- the least integer not less than 2.6
3
λ> floor 2.6   -- the greatest integer not greater 2.6
2
λ> round 2.6   -- round to nearest integer
3
λ> (sin pi)^2 + (cos pi)^2
1.0
```

GHCi

```
λ > x = 42
λ > x+1
43
λ > x
42
λ > let x = 42 in x+1
43
λ > let x = 1 in let x = 2 in x
2
λ > x = 1
λ > x = x+1
λ > x
^CInterrupted.
λ > y = y+1
λ > y
^CInterrupted.
```

GHCi

```
λ> "Haskell!"
"Haskell!"
λ> :type "Haskell!"
"Haskell!" :: String
λ> "Haskell" ++ " " ++ "programming"
"Haskell programming"
λ> ['H','a','s','k','e','l','l','!']
"Haskell!"
λ> 'H' : ['a','s','k','e','l','l','!']
"Haskell!"
λ> 'H' : "askell!"
"Haskell!"
λ> 'H' : 'a' : 's' : 'k' : 'e' : 'l' : 'l' : '!' : []
"Haskell!"
```

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
λ> :type pi
pi :: Floating 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 (defining our first function)

```
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 f
f :: (Eq a, Num a) => a -> a

λ> f 5
120
λ> f (f 3)
720
```

Exercice

The binomial coefficient $\binom{n}{k}$ can be computed by the multiplicative formula

$$\binom{n}{k} = \frac{n \times (n-1) \times \cdots \times (n-k+1)}{k \times (k-1) \times \cdots \times 1}$$

which using factorial notation can be compactly expressed as

$$\binom{n}{k} = \frac{n!}{k!\,(n-k)!}$$

Write implementations for computing $\binom{n}{k}$.

GHCi

```
λ> 'a'
'a'
λ> :type 'a'
'a' :: Char
λ> 'abc'
error: Syntax error on 'abc'
λ> 'a' : "bc"
"abc"
λ> :type (:)
(:) :: a -> [a] -> [a]
```

GHCi

```
λ> "abc"
"abc"
λ> :type "abc"
"abc" :: String
λ> "abc" ++ "def"
"abcdef"
λ> :type (++)
(++) :: [a] -> [a] -> [a]
```

Types

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.
- `Integer` - 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]
λ> :type "ab" == :type "cd"
error: parse error on input ':'
```

List types

```
λ>  :type [cos, sin]
[cos, sin] :: Floating a => [a -> a]
λ>  :type [1, 'a']
error: 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]]]
```

List types – List constructor

`[]` is a type constructor taking one type argument `a` and returning the type `[] a`, which is also permitted to be written as `[a]`.

List types – List constructor

[] is a type constructor taking one type argument a and returning the type [] a, which is also permitted to be written as [a].

```
λ> :info []
type [] :: * -> *
data [] a = [] | a : [a]
λ> :kind []
[] :: * -> *
λ> :type []
[] :: [a]
```

```
λ> :type [[]]
[[]] :: [[a]]
λ> :type [[[]]]
[[[]]] :: [[[a]]]
```

List types – Cons operator

The `:` operator is known as the cons operator and is used to prepend a head element to a list.

```
(:) :: a -> [a] -> [a]
```

List types – Cons operator

The `:` operator is known as the cons operator and is used to prepend a head element to a list.

```
(:) :: a -> [a] -> [a]
```

```
λ> [1,2,3]
[1,2,3]
λ> 1:[2,3]
[1,2,3]
λ> 1:2:[3]
[1,2,3]
λ> 1:2:3:[]
[1,2,3]
```

List types

Exercise

Which of these are valid Haskell, and why?

```
[1,2,3,[]]
```

```
[1,[2,3],4]
```

```
[[1,2,3],[]]
```

List types

Exercise

Which of these are valid Haskell, and which are not? Rewrite in comma and bracket notation.

```
[]:[[1,2,3],[4,5,6]]
```

```
[]:[]
```

```
[]:[]:[]
```

```
[1]:[]:[]
```

```
["hi"]:[1]:[]
```

List types

Exercice

Can Haskell have lists of lists of lists? Why or why not?

Exercise

Why is the following list invalid in Haskell?

```
[[1,2],3,[4,5]]
```

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)
error: Couldn't match expected type 'b' with actual type '(a, b)'
```

Tuple types

Exercise

Which of these are valid Haskell, and why?

```
1 : (2,3)
```

```
(2,4) : (2,3)
```

```
(2,4) : []
```

```
[(2,4),(5,5),('a','b')]
```

```
([2,4],[2,4,5])
```

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 -- parity predicate (see also odd)
even :: Integral a => a -> Bool
λ> :type mod -- modulo
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

`T1 -> T2 -> T3`

means

`T1 -> (T2 -> T3)`

Curried functions

The type

```
T1 -> T2 -> T3 -> T4
```

means

```
T1 -> (T2 -> (T3 -> T4))
```

Curried functions

The type

`T1 -> T2 -> T3 -> T4 -> T5`

means

`T1 -> (T2 -> (T3 -> (T4 -> T5)))`

Curried functions

Multiplying three integers

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

Curried functions

Multiplying three integers

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

```
λ> mult 2 3 4 -- mult 2 3 4 == ((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> mult' = mult  2
λ> mult'' = mult' 3
λ> mult'' 4
24
λ> :type mult'
mult' :: Int -> Int -> Int
λ> :type mult''
mult'' :: Int -> Int
```

Curried functions

Exercice

`uncurry` is a function that undoes currying; that is, it converts a function of two arguments into a function that takes a pair as its only argument.

```
uncurry :: (a -> b -> c) -> (a, b) -> c
```

Write implementations for `uncurry`.

Exercise

`curry` is is the opposite of `uncurry`.

```
curry :: ((a, b) -> c) -> a -> b -> c
```

Write implementations for `curry`.

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'
```

Classes

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
```

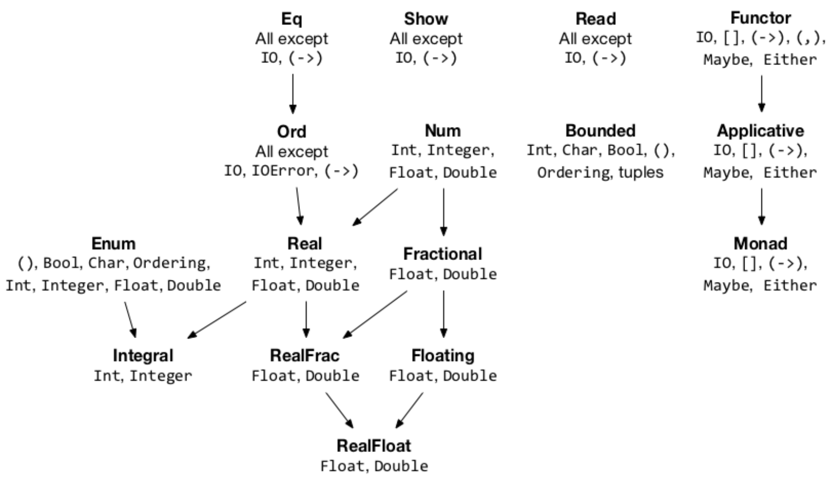
Overloaded types

```
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.

Haskell classes



`Eq` – **Equality types**

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

```
class Eq a where
  (==) :: a -> a -> Bool
  (/=) :: a -> a -> Bool
```

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

Eq – Equality types

```
λ> True == True
True
λ> 'a' == 'b'
False
λ> "abc" == "abc"
True
λ> 2.5 == 5.2
False
```

Eq – Equality types

```
λ> ('a', 1) == ('b', 1)
False
λ> (1, 2, 3) == (1, 2)
error: 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
error: No instance for (Eq (Double -> Double)) arising from a
       use of '=='
```

Basic classes

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:

```
class Eq a => Ord a where
  (<)  :: 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.

`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
```

`Ord` – Ordered types

```
λ> (1,2) < (1,3)
True

λ> (1,2,3) < (1,1)
error: 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
error: No instance for (Ord (Double -> Double)) arising from a
       use of '<'
λ> (1, sin) > (2, cos)
error: 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:

```
class Show a where
  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:

```
class Read a where
  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])
```

Basic classes

Num – Numeric types

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

```
class Num a where
  (+) :: 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:

```
class (Real a, Enum a) => Integral a where
  div :: a -> a -> a
  mod :: a -> a -> a
```

`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
```

Basic classes

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:

```
class Num a => Fractional a where
  (/) :: a -> a -> a
  recip :: a -> a -> a
```

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

Basic classes

`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
```

Converting numbers

The workhorse for converting from integral types is `fromInegral`, which will convert from any Integral type into any Numeric type (which includes `Int`, `Integer`, `Rational`, and `Double`):

```
fromIntegral :: (Num b) => a -> b
```

For example, given an `Int` value `n`, one does not simply take its square root by typing `sqrt n`, since `sqrt` can only be applied to `Floating`-point numbers.

Instead, one must write `sqrt (fromIntegral n)` to explicitly convert `n` to a floating-point number.