# Basic types in Haskell

## Contents

# Functional programming

**Types** play the role of **sets**. They can consist of numbers, pictures, or even functions!

- Function type `f :: Int -> Int`
- Sum type (next lecture)
- Product type (next lecture)

**Programs** inhabit types, so they are like the elements of sets

- programs usually have function type
- but they can also be a function without input (without `->`)
- these functions are partial, so they can crash or fail to return an output

# Functional programming

A **function** accepts inputs from particular types and gives a result of a particular type

### Example

```
(+) :: Int -> Int -> Int
```

first input type

second input type

output type

# Some types

There are lots of **built-in types**

- `Bool` `True, False`
- `Char` `'h', 'c'`
- `String` `"hello"`
- `Int` `42, -123`
- `Double` `3.14`

We can also define **custom types**

- `Triangle` 
- `Picture` 

# Why types?

(Thompson: §1.4)

## Static typing

- helps clarify the program structure
- serves as a form of documentation
- turns run-time errors into compile-time errors

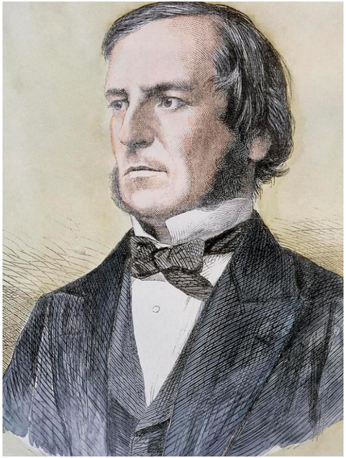
## In Haskell

- every expression has a type
- types are checked at compile-time
- programs with type errors will not compile

# Boolean values

Named after **George Boole**

- Lived in the 1800s
- English Mathematician/philosopher/logician
- Founder of the **information age**



In Haskell `Bool` is defined by

```
data Bool = False | True
```

name of type possible values

# Boolean operations

(Thompson: §3.1)

Some **operators** on Boolean values are `&&` (and), `||` (or), `not` (negation)

| `a` | `b` | `a && b` | `a \|\| b` | `not a` |
|---|---|---|---|---|
| `True` | `True` | `True` | `True` | `False` |
| `True` | `False` | `False` | `True` | `False` |
| `False` | `True` | `False` | `True` | `True` |
| `False` | `False` | `False` | `False` | `True` |

In Haskell

- `&& :: Bool -> Bool -> Bool`
- if `a :: Bool` and `b :: Bool` then `a && b :: Bool`
- Similar for `||` and `not`

# Exclusive or

(THOMPSON: §3.1)

Suppose we want to define

```haskell
exOr :: Bool -> Bool -> Bool
```

| a | b | exOr a b |
|---|---|---|
| True | True | False |
| True | False | True |
| False | True | True |
| False | False | False |

We can define this as follows:

```haskell
exOr :: Bool -> Bool -> Bool            -- type
exOr a b = (a || b) && not (a && b)     -- definition
```

# Some functions that return a Boolean value

(Thompson: §3.2)

- `==` equal to
- `/=` not equal to
- `>` strictly greater than
- `>=` greater than or equal to
- `<` strictly less than
- `<=` less than or equal to

**Example** Let `2 :: Int` and `4 :: Int`, then

- `(2 == 4) = False`
- `(2 /= 4) = True`
- `(2 > 4) = False`

# Char



The type `Char` consists of **literal characters**

- `'a', ..., 'z', 'A', ..., 'Z'`

as well as **escape characters**

- `'\t'` tab
- `'\n'` newline
- `'\\'` backslash (\)
- `'\''` single quote (')
- `'\"'` double quote (")

# String



The type `String` consists of **lists of characters**, written as `"string"`

```
ghci> "This is a string!"
"This is a string!"
ghci> "blue" ++ "tongue"
"bluetongue"
ghci> head "blue"
'b'
ghci> tail "blue"
"lue"
```

# Integer and Int



`Integer` represents whole numbers of any size, limited by the machine's memory

`Int` represents integers in a fixed amount of space

- At least $[-2^{29}, \ldots, 2^{29} - 1]$, but sometimes more
- Find minimum and maximum via `minBound :: Int` and `maxBound :: Int`

**Operations** on `Integer` (for `Int` they work between `minBound` and `maxBound`)

| | | |
|---|---|---|
| `+`, `*`, `-` | add, multiply, subtract | `2 + 2` |
| `^` | exponentiation (raise integer to a power) | `2^3` |
| `div` | integer division (rounded down) | `div 11 5` |
| `mod` | remainder of integer division | `mod 11 5` |
| `abs` | absolute value of an integer | `abs (-5)` |
| `negate` | change the sign of an integer | `negate (-5)` |

# Double

(Thompson: §3.6)

`Double` is used to represent numbers with fractional parts

- double-precision floating point numbers
- fixed amount of space to represent number, may cause **imprecise arithmetics** (follow this link for more info)

**Example**

```
ghci> 10/3 :: Double
3.3333333333333335
```

We can use all the **operations** mentioned, and more:

| | | |
|---|---|---|
| `/` | fractional division | `432.1 / 1357.9` |
| `**` | floating-point exponentiation | `3.2 ** 4.5` |
| `sqrt` | square root | `sqrt 2.6` |
| `sin`, `cos`, `tan` | sine, cosine and tangent | `cos 43` |

# Some functions

(Thompson: §3.6)

Some functions between numeric types

- `fromIntegral :: Int -> Double` converts integer to floating point

works from `Int` and `Integral` to any other numeric type

- `round :: Double -> Int` rounds to closest integer
- `floor :: Double -> Int` rounds down
- `ceiling :: Double -> Int` rounds up

The last three also work as `Double -> Integer`

# Integer versus double



**Example** We may have non-numeric results

```
ghci> 1/0
Infinity
```

**Example** `Integral` and `Double` do not automatically interact

```
ghci> (floor 5.6) + 6.7
<interactive>:8:1: error: ...

ghci> fromIntegral (floor 5.6) + 6.7
11.7
```

# More numerical types



There are many more numerical types

- `Float` is like `Double` but uses less space
- `Rational` for rational numbers, is precise (unlike `Double`) but slow
- `Word` is for natural numbers, bounded like `Int`

There are many more in basic libraries

- `Complex`
- `Natural`
- ...

# Type synonyms

(Thompson: §5.3)

**Type synonyms** can give existing types new names via

```
type newName = Int
```

**Example** Convert fahrenheit to celcius

```
fahrToCel :: Double -> Double
fahrToCel x = (x - 32)/1.8
```

Now using **type synonyms**

```
type Fahrenheit = Double
type Celcius = Double

fahrToCel :: Fahrenheit -> Celcius
fahrToCel x = (x - 32)/1.8
```

# Summary

## Haskell ...

- has many built-in types
- has many built-in functions
- lets us define type synonyms
- allows us to define our own types

## Next lectures

- Thursday: Algebraic datatypes
- Next week: Cases, lists and recursion