Basic Types

Booleans

Characters

Strings

Numeric types

Various Types

Built-in:

```
Bool      ☛ True, False
Char      ☛ 'h'
String    ☛ "hello"
Int       ☛ 42, -69
Double    ☛ 3.14
```

Custom-defined:

```
Picture   ☛  ♞
```

Types

A **type** is the programming equivalent of the mathematical notion of **set**

Its elements might be

- numbers
- pictures
- …
- even functions!

A function accepts inputs from a particular type and gives a result of a particular type too.

For example, function ’+’ accepts two inputs of some numerical type, and outputs a value of the same numerical type.

Static typing

- Helps clarify thinking and express program structure.
- Serves as a form of documentation.
- Turns run-time errors into compile-time errors.

Haskell’s expressions are Statically Typed

- Every Haskell expression has **a type**
- Types are all checked at compile-time.
- Programs with type errors will not compile!

Basic Types

Some basic types and constructions are available in the Prelude:

- https://hackage.haskell.org/package/base/docs/Prelude.html
- Look for the keywords ‘data’ and ‘type’
- Each comes with its own defined functions

Even more types and defined functions in the basic libraries:

- https://hackage.haskell.org/package/base
- These need to explicitly ‘imported’ if you want to use them

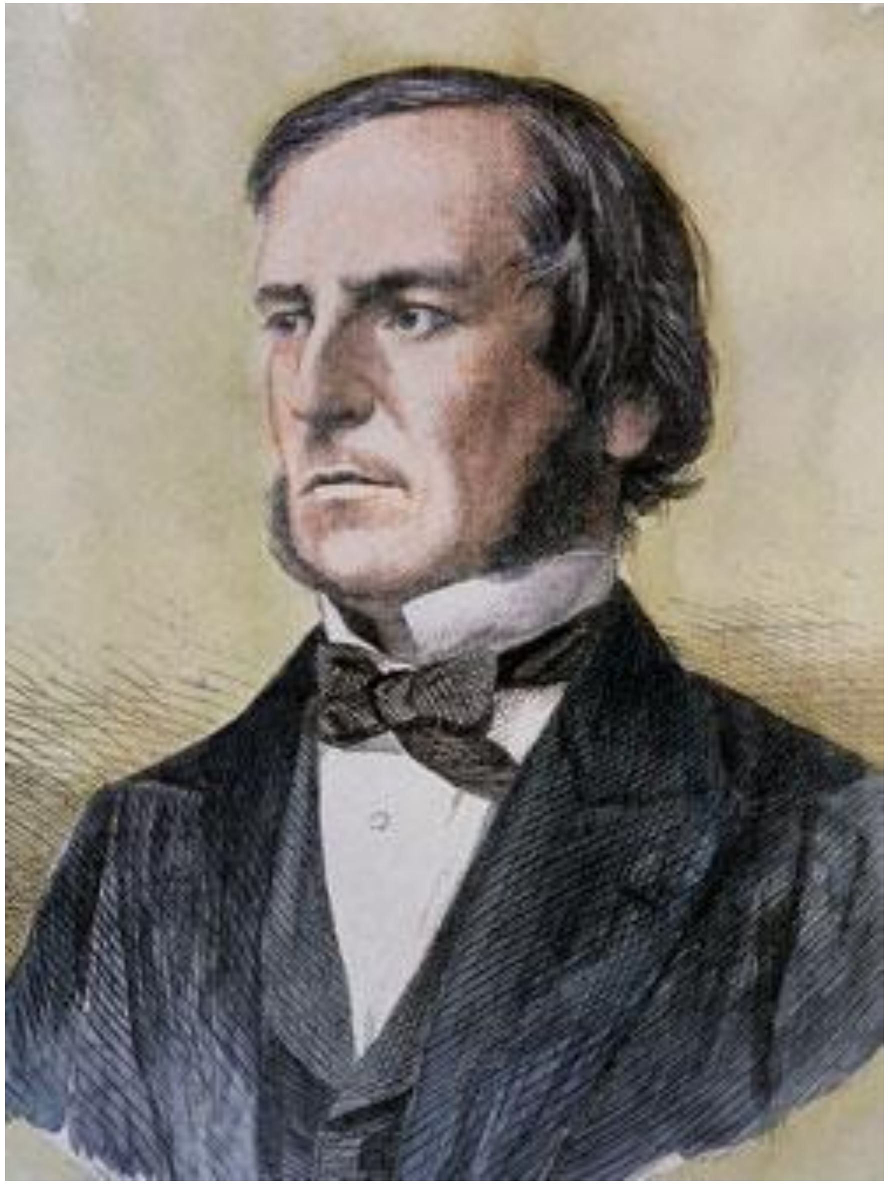
Booleans

Named after logician George Boole

The Haskell type is called `Bool`.

Boolean operators:

`&&`	logical “and”
`||`	logical “or” (inclusive)
`not`	logical “negation”



Bool

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

Boolean Operators (logical connectives)

Operator	Description
&&	and
||	or
not	not (negation)

Truth tables

t_1	t_2	t_1 && t_2	$t_1 \mid\mid t_2$	not t_1
True	True	True	True	False
False	True	False	True	True
True	False	False	True	
False	False	False	False	

Boolean function definition: “*exclusive* or”

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

t_1	t_2	exOr t_1 t_2
True	True	False
False	True	True
True	False	True
False	False	False

```
exOr x y = (x || y) && not (x && y)
```

Some special functions that return a Boolean

Operator	Description
==	equal to
/=	not equal to
>	greater than (and not equal to)
>=	greater than or equal to
<	less than (and not equal to)
<=	less than or equal to

`Char`: character

Literal characters are written inside single quotes:
`'a'`, …, `'z'`, `'A'`, ..., `'Z'`, etc.

Escape characters:

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

String

```
Prelude> "This is a string!"
"This is a string!"
```

```
Prelude> "blue" ++ "tongue"
"bluetongue"
Prelude> head "blue"
'b'
```

Integer

`Integer` represents whole numbers (positive, zero and negative) **of any size** (up to the limit of your machine's memory).

Operation	Description	Example
+, *, -	Add, subtract, multiply two integers	2 + 2
^	Raise an integer to the power	2^3
div	Whole number division (rounded down)	div 11 5
mod	The remainder from whole number division	mod 11 5
abs	The absolute value of an integer	abs (-5)
negate	Change the sign of an integer	negate (-5)

`Int`

The `Int` type represents integers in a fixed amount of space,

i.e. `Int` is **bounded**.

Thus `Int` only represents a **finite range of integers** and the range is guaranteed to be **at least**

$$[-2^{29} \; \dots 2^{29} - 1]$$

However, the range can actually be bigger, depending on the compiler and your machine. To find its lowest and greatest bounds on your machine, enter in your GHCi prompt:

```
minBound :: Int
```

```
maxBound :: Int
```

Arithmetic operations applicable to `Integer` are also applicable to `Int`, but will often be faster.

However, one should take care that the result stays within `minBound` and `maxBound` limits to prevent **arithmetic overflow**.

```
Prelude> (maxBound :: Int) + 1
-9223372036854775808
```

Double

- Type Double can be used to represent numbers with fractional parts (i.e., **double-precision floating-point numbers**).
- However, there is a **fixed amount of space** allocated to representing each value of type Double. Therefore, not all real numbers (or even rationals) can be represented by floating-point numbers. **This may result in imprecise arithmetic results:**

https://wiki.haskell.org/Performance/Floating_point

```
Prelude> (3.3)^2 :: Double
10.889999999999999
```

Operations applicable to Floating-point Numbers

Operation	Description	Example
+, *, -	Add, subtract, multiply two integers	2 + 2
/	Fractional division	453.3 / 1346.6
^	Exponentiation x^n for an integer n	3.2^4
**	Exponentiation x^n for a floating-point number n	3.2**4.5
sqrt	Square root	sqrt 2.6
abs	Absolute value	abs (-5.442)
negate	Change the sign of a number	negate (-5.882)
cos, sin, tan	Cosine, sine and tangent	cos 43

Floating point ⇔ Integral Conversion

`fromIntegral` converts from any integral type (`Int` or `Integer`) to any other numeric type.

`round`, `floor`, `ceiling` convert floating point numbers to `Int` or `Integer`.

Beware of the following

- non-numerical results

```
Prelude> 1 / 0
Infinity
```

- no automatic conversion from `Integral` to `Double`

```
Prelude> (floor 5.6) + 6.7
<interactive>:8:1: error:
...
Prelude> fromIntegral (floor 5.6) + 6.7
11.7
```

Other Numerical Types

Our course will usually use `Int` and `Double`, but other numerical types exist in the Prelude:

- `Float` – like `Double`, but uses less space
- `Rational` – Rational numbers; precise unlike `Double`, but significantly slower to compute with
- `Word` – Natural numbers; bounded in space like `Int`

And many more exist in the basic libraries - `Complex`, `Natural`, …

`type`

We can give existing types new names with the `type` keyword

```
type IdNumber = Int
```

This has **no** computational significance but can make programs more readable.