

Lab 1 – Recap

- Starting up Hugs (with Linux!)
 - Hugs>: default session (functions predefined in `Prelude.hs`)
 - Main>: session after loading your own script (functions predefined in `Prelude.hs` AND functions defined in `itoea.hs`, `exercise1.hs`, etc.)
- Evaluating expressions using predefined functions and operators and user-defined functions
 - `3`, `3.0`, `'3'`, `"3"`, etc.
 - `sqrt 2.0`, `sin (pi/2.0)`, `6 < 4`, `3 > 4 && 4 > 3`, etc.
 - `cube 2`, `volume`, `surfaceArea 3`
- Creating, loading and using a Haskell script with basic content
 - comments (`-- Languages and Computability`)
 - type declaration (`cube :: Int -> Int`)
 - function definition (`cube x = x * x * x`)
- You have a rough idea of what a type is, how it is declared, how a function is defined
- Let us go into more details...

Defining Functions

- In Mathematics, a function f associates each member of a set A (the domain of f) with a single member of a set B (the codomain of f): $f : A \rightarrow B$
- If $a \in A$ then $f(a)$ is the associated member of B
- Example: for $f : \mathbb{R} \rightarrow \mathbb{R}$ define $f(x) = x^2$
- Haskell uses this same concept and similar notation, e.g.
square' :: Float -> Float (type declaration)
square' x = x^2 (function definition)
- Less formally, a function gives an output value which depends upon the input value(s) (output meaning result and input(s) meaning argument(s) or parameter(s))

Defining Functions

- One notational difference to be aware of is that in Haskell we can write `f x` (even if it can be confusing at first) instead of `f(x)` as the brackets serve no real purpose
- Suppose you write `square x+1`; the spacing may suggest `square (x+1)`, but in fact it means `(square x)+1`
- The best way to avoid confusion is just to remember that *function application* has *higher priority* than all other operators (e.g. `+` or `*`) and it is *left associative*
- *Function application* is the process of giving particular inputs to a function
- For example, for addition over numbers (function/operator `+`), given input values 12 and 10 the corresponding output is 22

Defining Functions

- Always chose *meaningful names* for the functions you define
- You can use any sequence of letters, digits, underscores (_) and primes (') except that:
 - the name must begin with a lowercase letter (names beginning with upper case letters are used for types, modules and constructors)
 - you cannot use a name that has already been used (that includes all the functions in prelude)
 - some names are reserved words, and have a special meaning in Haskell so you cannot use them for any other purpose

Defining Functions

- Haskell functions can take more than one argument, for example
`add :: Int -> Int -> Int`
`add x y = x+y`
- Corresponding expressions are written `add 3 4` rather than `add(3,4)` or `add (3,4)`
- The expression `add (3,4)` is illegal
- To be able to write such an expression, the function must have a different type and be defined differently
- Which type and How?

Defining Functions

- Haskell functions can take more than one argument, for example
`add :: Int -> Int -> Int`
`add x y = x+y`
- We write `add 3 4` rather than `add(3,4)` or `add (3,4)` for function application
- The expression `add (3,4)` is illegal
- To be able to write such an expression, the function must have a different type and be defined differently
- Which type and How?
`add' :: (Int, Int) -> Int`
`add' (x,y) = x+y`
- Now `add'` takes a single argument which is a pair of integers

Built-In Functions

<code>abs</code>	absolute value
<code>ceiling, floor, round</code>	rounding up, down or to the closest integer
<code>sqrt</code>	square root
<code>cos, sin, tan</code>	trigonometric functions
<code>acos, asin, atan</code>	inverse trigonometric functions
<code>exp</code>	powers of e
<code>log</code>	logarithm to base e
<code>negate</code>	change their sign of a number

Central Ideas in Functional Programming

- A type is a collection of values (a kind of value), like the whole set of integers
- A function is an operation which takes one or more arguments (inputs, parameters) to produce a result (output)
- Functions operate over particular types

Function Layout

- In any programming language, the layout of programmes is important for their readability
- In Haskell, *layout rules* help to get rid of the annoying punctuation used in many other languages (semicolons, braces, begin/end, etc.)
- Haskell uses *indentation* to decide the ends of definitions, expressions, and so on
- **Basic rule: A definition ends when a (non-space) symbol appears in the same column as the first symbol of the definition**

Function Layout

```
■ cube x = x * x * x
  answer = 6 * 7
```

This is correct. The definition of `cube` ends when the 'a' of `answer` appears in the same column as the 'c' of `cube`

```
■ cube x
    = x *
      x * x
  answer = 6 * 7
```

This is correct too but harder to read

```
■ cube x
= x *
  x * x
```

This is wrong. The definition of `cube` is ended by (before) the equal sign. Hugs will detect this as a syntax error:

```
ERROR ‘‘cube.hs’’:2 - Syntax error in declaration (unexpected ‘;’)
```

- The error message may seem strange: what semicolon?
- Explanation: definitions may also be ended by ';', so this message is just telling you that the definition ended too soon.

Function Layout

- Some more examples:

```
myMin x y | x < y = x | otherwise = y
```

This is correct but hard to read

```
myMin x y
  | x < y = x
  | otherwise = y
```

This is better

Function Layout

```
■ cube x
    = x * x * x
  answer = 6 * 7
```

```
■ cube x
    = x * x * x
  answer = 6 * 7
```

Both of these are wrong since `answer` does not line up under `cube`

Note that this must hold even if sections of code are separated by comments

Expression Evaluation

- Haskell evaluates an expression by repeatedly performing *simplifications* (or *reductions*) on the expression until no more reductions can be performed
- When an expression is fully reduced in this way it is said to be in *normal form*
- Examples of the different kinds of reduction are:
 - replacing a built-in function application with its value (e.g. `sqrt 2`)
 - replacing a user-defined function name with its body and substitute the actual parameters into it to replace the corresponding identifiers (e.g. `cube 2`)
- For example:

`cube (2*(3+4))`

$\Rightarrow (2*(3+4)) * (2*(3+4)) * (2*(3+4))$

$\Rightarrow (2*7) * (2*7) + (2*7)$

$\Rightarrow 14*14*14$

$\Rightarrow 2744$