

# Values, Functions, Types

**COMP1100 — Introduction to Programming and Algorithms**

Clem Baker-Finch

Australian National University

Semester 1, 2006

- Lab classes commence **this week**.  
You should have registered by now.
- Lab class exercises are available on the comp1100 web site.
  - Hard copies will not be handed out in lectures.
  - You should *at least* read them and think about them before attending your lab session.
  - We expect you to come prepared.
- Workload: average 10 hours per week for 16 weeks.
  - Attending lectures and labs is *not enough*.
  - **Doing** (writing bit of programs) is much more effective than just reading about it.
- Extra assistance — drop-in labs will commence soon.

## Values

**Values** are things such as 42 (an integer), "Hello, it's 2pm" (a string of characters) and 3.1412 (a floating point number).

Values may be passed to **functions** which return other values, eg:

- `(*)` takes two numbers and produces a number:  $13 * 5 \Rightarrow 65$
- `++` takes two strings and produces another string by concatenating them.
- `length` takes a string and produces a number: the length of the string.

We can combine values and functions by using the *result* of a function application as input to another function, e.g.

- `length ("Hello, " ++ "it's 2pm")`

# Types

What's the difference between `42` and `"Hello, it's 2pm"`?

What does `2 + "abc"` mean?

Values may naturally be grouped together in different categories.

We call these categories **types**. For example:

Type	Example values
Int	<code>...-3, -2, -1, 0, 1, 2, 3, ...</code>
Float	<code>1.0, 3.1412, ...</code>
String	<code>"Hello", "The time is 11 o'clock!", ...</code>
Char	<code>'a', 'A', 'x', 'S', '%', '8', ...</code>
Bool	<code>True, False</code>

GHCi can tell you the type of any Haskell expression with the command `:type` (or just `:t`)

```
Prelude> :type 7 + 3 < 5  
7 + 3 < 5 :: Bool
```

In Haskell we can also define **our own types**, and build **compound types** by collecting types together into **data structures**.

Much more on this later ...

## Functions — canned computations

The central activity in writing Haskell programs is 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  $f$ ) and we write

$$f : A \rightarrow B$$

If  $a \in A$  then  $f(a)$  is the associated member of  $B$ .

Haskell uses this same concept and similar notation.

How to define our own functions?

Let's start simple: a function whose result is the input number times 2.

We'll call it `double`.

The algebraic expression  $x + x$  computes `double x` for any  $x$ , so our function definition looks like:

```
double x = x + x
```

On the left of the = sign, `double` is the *name* of the function and `x` is the *name* of the argument.

On the right of the = sign is the *body* of the function definition.

The result of applying the function `double` to an argument *value* is computed by replacing all occurrences of `x` in the *body* by that value.

for example:

$$\text{double } 5 \Rightarrow 5 + 5 \Rightarrow 10$$

The arrows indicate steps in the computation.

Another example:

$$\begin{aligned} \text{double } (\text{double } 3) &\Rightarrow \text{double } (3 + 3) \Rightarrow \text{double } 6 \\ &\Rightarrow 6 + 6 \Rightarrow 12 \end{aligned}$$

## Parentheses around arguments

One notational difference to be aware of is that in Haskell we can write `f x` instead of `f (x)`

The brackets serve no real purpose. Leaving them out makes for less cluttered notation, but it can take some getting used to ...

Suppose you write: `double 3+1`

The spacing may suggest: `double (3+1)`

but in fact it means: `(double 3)+1`

The best way to remember this is that *function application* is just like any other operator, but it has *higher priority* than *all* other operators and it is *left associative*.

## Type signatures

Functions take arguments of certain types and give results of certain types. For example, `double` takes argument values of type `Int` and return values of type `Int`.

So functions have type, too. The type of `double` is `Int -> Int`.

The complete definition of the function is:

```
double :: Int -> Int
double x = x + x
```

[`double 3.1412` also works, but `3.1412` is not an `Int`. We'll get to that later.]

## Multiple arguments

In last week's lectures we wrote function definitions for resting metabolic rate which took 3 arguments: weight, height and age:

```
maleRestRate weight height age = ...
```

The type signature for functions with more than one argument separates the argument types with (->):

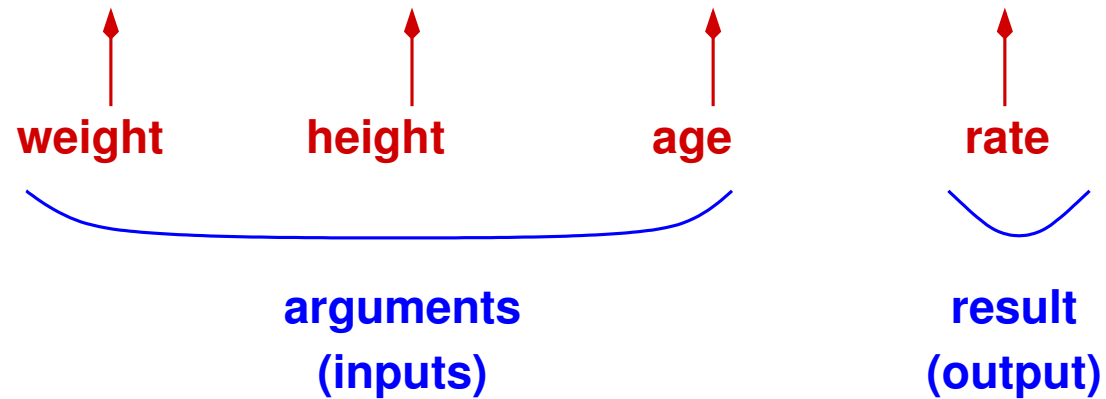
```
maleRestRate :: Float -> Float -> Float -> Float
```

and we write applications: `maleRestRate 85 190 21`

rather than: `maleRestRate(85,190,21)`

## Reading type signatures

`maleRestRate :: Float -> Float -> Float -> Float`



Another simple function definition:

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

We write applications: `add 3 4`

Haskell functions take their arguments *one at a time*.

It is legal to write an expression like `add 3`. (Try it.)

The value of the expression `add 3` is a **function** of type `Int -> Int` which adds 3 to its argument. (Try it.)

So `add 3 4` is the same as `(add 3) 4`.

(Function application is left-associative.)

## Type checking

GHC will work out the type of *every expression* and *every function* from its definition.

If you declare the type of a function, GHC will check whether you are right.

You should always declare the type of **every** functions you define in your programs:

- The *type* of a function is a *basic part of its design*.
- Type declarations are an important part of *program documentation*.
- Type declarations help you to find *errors*.

If GHC figures that the type of a function is different to what you expect, then *you* have made an error.

## Example

Suppose you want to design a function `isEven` such that `isEven x` returns `True` if `x` is divisible by 2 and `False` otherwise.

What's wrong with this:

```
isEven :: Int -> Bool
isEven x = x `div` 2
```

(What is a correct definition?)

## Introducing overloading

Values of type `Int` can be added using the `(+)` operator.

Values of type `Float` can be added using the `(+)` operator.

Using the **same symbol or name** for *different* operations is called **overloading**.

The function `(+)` *simultaneously* has types:

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

```
(+) :: Float -> Float -> Float
```

But every Haskell expression has **one** type.

If we had *type variables* (and we do) we might say:

(+) has type `a -> a -> a` where `a` is either `Int` or `Float`

In fact Haskell allows (+) to be applied to *any numeric type* (which includes `Double` and `Integer` as well as `Int` and `Float`)

The set of numeric types form a *type class* called `Num`

If we ask GHCi for the type of (+) we get:

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

which means (+) has type `a -> a -> a`

for any type `a` in class `Num`

## Comments in scripts

Haskell allows us to annotate scripts with **comments**. There are two kinds:

- everything on a line following the symbol `--`
- everything between the symbols `{-` and `-}`

The computer ignores comments but they may help humans reading our programs — *including ourselves* — to understand them.

***Programming is a human activity***

## Write your comments *as you code*

Write your comments at the program *design* phase and maintain them through the *development* phase.

- *Do not* add them to your scripts later
- They are for *your* benefit, too

Always include an identifying banner comment in every script, including:

- author's name
- date
- the purpose of the script

## Choosing names for functions, variables, etc.

**People read programs**, not just computers.

During development, maintenance, testing, review, etc. **you and other programmers** read your programs.

```
metabolicRestRate sex weight height age
```

is much better than:

```
rate sx wt ht yr
```

is much better than:

```
f x y z u
```

(The computer doesn't care but humans do.)

## Haskell lexical rules

- Haskell is **case sensitive** so `restRATE` is different from `restrate`.
- Names of functions, variables, type variables **must begin with a lower case letter**
- Names can contain letters (upper and lower case), digits, underscores `'_'` and apostrophes `' '`
- Names of types must begin with an upper case letter
- Names of “data constructors” must begin with an upper case letter **so far we have only seen data constructors in *enumerations*. For example, `True` and `False` are the data constructors of type `Bool`.**

## Suggestions

- The name of a function should describe **what is being calculated** (a noun), rather than **how it does it** (a verb).
- Often names are made up of several words (or obvious contractions). Make the name by stringing the words together, starting each new word with a capital.
- Look (on the web) for some coding standards to see what others do.
- Whatever you do, develop a good style, and **be consistent**.