# Introduction to Programming, Aug-Dec 2008

Lecture 1, Monday 4 Aug 2008

## Administrative matters

### Resource material

Textbooks and other resource material for the course:

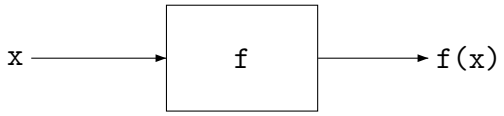
- *The Craft of Functional Programming* by Simon Thompson
- *Introduction to Functional Programming using Haskell* by Richard Bird
- *Programming in Haskell* by Graham Hutton
- *Introduction to Functional Programming* by Richard Bird and Philip Wadler
- Instructor's lecture notes, to be made available as the course progresses
- Online archive at `http://www.haskell.org`

### Evaluation

- Approximately 8–10 assignments, 50%
- Midsemester examination, 20%
- Final examination, 30%

## Introduction to Haskell

Programs in Haskell are functions that transform inputs to outputs. Viewed externally, a function is a black box:




The internal description of the function $f$ has two parts:

1. The types of inputs and outputs

2. The rule for computing the output from the input

In mathematics, the type of a function is often implicit: Consider $sqr(x) = x^2$ which maps each input to its square. We could have $sqr : \mathbb{Z} \to \mathbb{Z}$ or $sqr : \mathbb{R} \to \mathbb{R}$ or $sqr : \mathbb{C} \to \mathbb{C}$, depending on the context.

Here is a corresponding definition in Haskell.

```
sqr :: Int -> Int
sqr x = x^2
```

The first line gives the type of `sqr`: it says that `sqr` reads an `Int` as input and produces an `Int` as output. In general, a function that takes inputs of type `A` and produces outputs of type `B` has the type `A -> B`. The second line gives the rule: it says that `sqr x` is `x^2`, where `^` is the symbol for exponentiation.

## Basic types in Haskell

`Int` is a type that Haskell understands and roughly corresponds to the set of integers $\mathbb{Z}$ in mathematics. "Roughly", because every integer in Haskell is represented in a fixed and bounded amount of space, so there is a limit on the magnitude of the integers that Haskell can manipulate (think of what would happen if you had to do arithmetic with pencil and paper but could not write a number that was longer than one line on the page).

Here are (some of) the types that Haskell understands by default:

**Int** Integers. Integers are represented internally in binary. Typically, one binary digit (or bit) needs to be used to denote the sign (+/-) of the integer, and the remaining bits denote its magnitude. The exact representation is not important, but we should realize that the size of the representation is fixed (that is, how many binary digits are used to represent an `Int`), so the magnitude is bounded.

**Float** "Real" numbers. The word `Float` is derived from *floating point*, a reference to the fact that when writing down a real number in decimal notation, the position of the decimal point is not "fixed", but "floating". Internally, a Float is represented in *scientific notation* (for example, $1.987x10^{23}$) using two binary quantities: the mantissa and the exponent. For each of these, we reserve one bit for the sign and use the rest for the magnitude. Thus, we can represent numbers that are both very large and very small in magnitude: for instance, $1.987x10^{23}$ and $1.987x10^{-23}$.

Once again, the exact representation is not important but we should realize that `Float` is only an approximation of the set of real numbers, just as `Int` is only an approximation of the integers. In fact, the approximation in `Float` has two dimensions—there is a limit on magnitude and precision. Thus, real numbers are dense (between any two real numbers we can find a third) but floating point numbers are not.

**Char** Used to represent text characters. These are the symbols that we can type on the keyboard. A value of this type is written in single quotes: for instance, `'z'` is the character representing the letter z, `'&'` is the character representing ampersand etc.

A significant amount of computation involves manipulating characters (think of word processers) so this is an important type for programming.

**Bool** This type has two values, `True` and `False`. As we shall see, these are used frequently in programming.

## Compilers vs interpreters

The languages that we use to program computers are typically "high level". These have to be converted into a "low level" set of instructions that can be directly executed by the electronic hardware inside the computer.

Normally, each program we write is translated into a corresponding low level program by a *compiler*.

Another option is to write a program that directly "understands" the high level programming language and executes it. Such a program is called an *interpreter*.

For much of the course, we will run Haskell programs through an interpreter, which is invoked by the command `hugs` or `ghci`. Within `hugs`/`ghci` you can type the following commands:

`:load filename` — Loads a Haskell file

`:type expression` — Print the type of a Haskell expression

`:quit` — exit from hugs

`:?` — Print "help" about more hugs commands

## Functions with multiple inputs

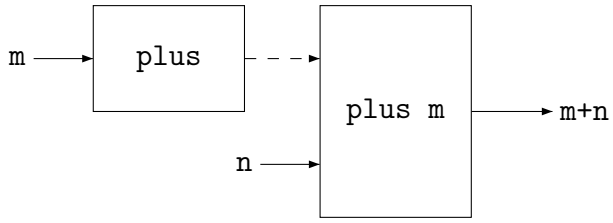
One feature of function definitions that we have not emphasized is the the number of inputs. For instance, the function sqr that we saw earlier has only one input. On the other hand, we could write a function on two inputs, such as the mathematical function *plus*, below

$$plus(m, n) = m + n$$

Mathematically, the type of *plus* would be $\mathbb{Z} \times \mathbb{Z} \to \mathbb{Z}$ (or $\mathbb{R} \times \mathbb{R} \to \mathbb{R}$). This means that, in addition to the types of the input and the rule for computation, we also need to include information about the *arity* of the function, or how many inputs it takes.

This complication can be avoided by taking the somewhat drastic step of assuming that all functions take only one argument. How then can we define a function such as *plus* that needs to operate on two arguments? We say that *plus* first picks up the argument $m$ and

becomes a new function *plus* $m$, that adds the number $m$ to its argument $n$. Thus, we break up a function on two arguments into a sequence of functions of one argument.




What is the type of *plus*? It takes in an integer $m$ and yields a new function *plus* $m$ that is like *sqr* above: it reads an integer and generates an output of the same type, so its type is $\mathbb{Z} \to \mathbb{Z}$ or, in Haskell notation, `(Int -> Int)`. Thus, in Haskell notation, `plus` reads an `Int` and generates a function of type `(Int -> Int)`, so the type of `plus` is `Int -> (Int -> Int)`. Here is a complete definition of plus in Haskell:

```
plus :: Int -> (Int -> Int)
plus m n = m + n
```

Notice that we write `plus m n` and not `plus(m,n)`—there are no parentheses around the arguments to a function. In fact, the correct bracketing for `plus m n` is `(plus m) n`. This tells us to first feed `m` to `plus` to get a function `(plus m)` to which we then feed the argument `n`.

What if we had a function of three arguments, such as `plus3(m,n,p) = m+n+p`? Once again, we assume that `plus3` consumes its arguments one at a time. Having read `m`, `plus3` becomes a function like `plus` that we defined earlier, except it adds on `m` to the sum of its two arguments. Since the type of `plus` was `Int -> (Int -> Int)`, this is the output type of `plus3`. The input to `plus3` is an `Int`, so the overall type of `plus3` is `Int -> (Int -> (Int -> Int))`.

Here is a complete definition of `plus3` in Haskell:

```
plus3 :: Int -> (Int -> (Int -> Int))
plus m n p = m + n + p
```

Once again, note the lack of brackets in `plus m n p`, which is implicitly bracketed `((plus m) n) p`.

In general, suppose we have a function `f` that reads `n` inputs `x_1`, `x_2`,..., `x_n` of types `t_1`,`t_2`,...,`t_n` and produces an output `y` of type `t`. The notation `::` introduced earlier to denote the type of a function is read as "is of type" and we can use here as well to write `x_1::t_1`, `x_2::t_2`, ..., `y::t` to denote that `x_1` is of type `t_1`, `x_2` is of type `t_2`, ..., `y` is of type `t`.

We can define the type of `f` by induction on `n`.

The base case is when `n` is 1, so `f` reads one input `x_1::t_1` and produces the output `y::t`. In this case, `f :: t_1 -> t`, as we have discussed earlier.

For the inductive step, we have a function that reads its first input `x_1::t_1` and then transforms itself into another function `g` that reads inputs `x_2::t_2`,`x_3::t_3`,...,`x_n::t_n` and produces an output `y::t`. Let the type of `g` be `T`. Then, `f : :t_1 -> T`. If we unravel the structure of `T` inductively, we find that

```
f::t_1 -> (t_2 -> (... -> (t_n -> t)...))
```

In this expression, the brackets are introduced uniformly from the right, so we can omit the brackets and unambiguously write

```
f::t_1 -> t_2 -> ... -> t_n -> t
```

## More on defining functions

The simplest form of definition is the one we have seen in `sqr`, `plus` and `plus3`, where we just write a defining equation using an arithmetic expression involving the arguments to the function.

The arithmetic operators that we can use in writing such an expression are `+,-,*,/` signifying addition, subtraction, multiplication and division. As usual, we can also use `-` in front of an expression to negate its value, as in `-(x+y)`. In addition, the function `div` and `mod` signify integer division and remainder, respectively. So `div 3 2` is `1`, `div 7 3` is 2, ... while `mod 10 6` is `4`, `mod 17 12` is `5`, ... Note that `div` and `mod` are functions, so they are written before their arguments rather than between them: it is `div 3 2` and `mod 17 12`, not `3 div 2` and `17 mod 12`.

We can also write expressions involving other types. For instance, for values of type `Bool`, the operator `&&` denotes the *and* operation, which returns `True` precisely when both its arguments are `True`. Dually, the operator `||`, pronounced `or`, returns `True` when at least one of its arguments is `True` (or, equivalently, `||` returns `False` precisely when both its arguments are `False`). The unary operator `not` inverts its argument. For instance, here is a definition of the function `xor` which returns `True` provided exactly one of its arguments is `True`.

```
xor :: Bool -> Bool -> Bool
xor b1 b2 = (b1 && (not b2)) || ((not b1) && b2)
```

We can also use operators to compare quantities. The result of such an operation is of type `Bool`. Here is a function that determines if the middle of its three arguments is larger than the other two arguments.

```
middle :: Int -> Int -> Int -> Bool
middle x y z = (x <= y) && (z <= y)
```

The comparison operators are `==` (equal to), `/=` (not equal to), `<` (less than), `<=` (less than or equal to), `>` (greater than), `>=` (greater than or equal to).