

# Lists

Reading: Thompson Ch.5

**COMP1100 — Introduction to Programming and Algorithms**

Clem Baker-Finch

Australian National University

Semester 1, 2006

## Lists

**Tuples** allow us to combine a **fixed number** of values of **various types**.

**Lists** allow us to combine a **varying number** of values of **the same type**.

1, 2, 3 are all of type Int, so [1, 2, 3] is of type [Int].

(Say [Int] as “list of Int.”)

String is a synonym for [Char].

"dog" is another way to write ['d', 'o', 'g'].

[[1,2,3], [1,99], [0], []] has type [[Int]].

That is, “list of list of Int”.

We can have lists of **ANY** type, so long as all elements are the **same** type.

## Which of the following are valid lists?

### What are their types?

[1, '2']

**Invalid** because the elements of a list must all be of the same type.

$1 :: \text{Int}$  but  $'2' :: \text{Char}$

[[1,2],3,4]

**Invalid** because the first element is  $[1,2] :: [\text{Int}]$  and the other elements have type  $\text{Int}$ . This is not a valid list because the elements do not all have the same type.

["cat", "sat", "on", "mat"]

**Valid** and has type  $[\text{String}]$  or equivalently,  $[[\text{Char}]]$ .

`[(1,2), (1,2,3)]`

**Invalid** because the elements are of different types: `(Int, Int)` and `(Int, Int, Int)` respectively.

`[(10, 'a'), (3, 'x'), (42, 'm')]`

**Valid** and has type `[(Int, Char)]`.

`[True, [False]]`

**Invalid** because `Bool` and `[Bool]` are different types.

`[[1], []]`

**Valid** expression of type `[[Int]]`.

Try some experiments for yourself.

# Constructing Lists

## The *cons* function

Lists are **constructed** by adding elements to the *front*:

$$42 : [7, 65, 3] \implies [42, 7, 65, 33]$$

*EVERY LIST* is constructed by a sequence of elements *cons*-ed onto the front of the **empty list**: `[]`.

`[42, 7, 65, 3]` is another notation for `42 : 7 : 65 : 3 : []`

**The *cons* operator can only add elements at the front.**

$$[7, 65, 3] : 42 \implies \text{error!}$$

## Other Useful List Functions

**Concatenate:** Join lists together using (++):

`[3,5,7] ++ [7,8,9] ==> [3,5,7,7,8,9]`

**Index:** Select elements from a list using (!!):

`['a','b','c','d'] !! 2 ==> 'c'`

*Indexes start from 0.*

**Head and tail:** Together, the dual of (:)

`head ['a','b','c','d'] ==> 'a'`

`tail ['a','b','c','d'] ==> ['b','c','d']`

Remember "abcd" is the same as `['a','b','c','d']` .

## More Useful List Functions

**Length:** The length function:

`length [42,7,65,3] ==> 4`

**Sum, product:** Add or multiply the elements of a list of numbers:

`sum [2,3,4] ==> 9`

`product [2,3,4] ==> 18`

**Reverse:** the order of list elements:

`reverse [42,7,65,3] ==> [3,65,7,42]`

Lots more in the Prelude and the List library.

## Arithmetic Progressions

We sometimes want a list of values in an *arithmetic progression*.

Haskell provides a special syntax:

$$[5..10] \implies [5, 6, 7, 8, 9, 10]$$

To specify step sizes other than 1, give the first 2 numbers. For example:

$$[1, 3..10] \implies [1, 3, 5, 7, 9]$$
$$[0, 10..50] \implies [0, 10, 20, 30, 40, 50]$$

Remember that only *arithmetic progressions* work.

$[2, 4, 8..128]$  is *illegal*.

The notation works for other types in Haskell such as `Float` and `Char`.

## Example: roots of a quadratic, revisited

A quadratic may have 0, 1 or 2 real roots.

In the earlier version, we raised an error if there was 0, and returned a pair containing the same value twice, if there was 1.

Alternatively, we can return a *list* of 0, 1 or 2 elements, depending on the discriminant.

```
roots :: Float -> Float -> Float -> [Float]
```

```
roots a b c
```

```
  | discriminant == 0 = [ -b/(2*a) ]
```

```
  | discriminant > 0 = [ (-b + (sqrt discriminant))/(2*a),  
                        (-b - (sqrt discriminant))/(2*a) ]
```

```
  | otherwise        = []
```

```
  where
```

```
    discriminant = b^2 - 4*a*c
```

## Polymorphism

What are the *types* of `(:)`, `(++)`, `(!!)`, `length` ...?

Since a list can have elements of *any type*, `(:)` has to work for any type.  
(Infinitely many of them.)

Notice that all these functions change or inspect the **structure of the list** without touching the **elements**.

In other words, you don't need to know what the list elements are to work out its length:

$$\text{length } [\square, \square, \square, \square, \square] \implies 5$$

Functions like these are *generic* as they work on lists of *any type*. We call them **polymorphic** functions, and we write their types using *type variables*.

A *single definition* of a polymorphic function is sufficient for all types.

For example, a function to return the first element of a pair can be defined like this:

$$\text{fst } (x, y) = x$$

and works in exactly the same way, no matter what are the types of  $x$  and  $y$ .

(In languages without polymorphism, you would have to write separate definitions for `fst` for each different pair of element types that you wanted to handle. Each time the code would be identical, possibly apart from a type declaration.)

## Polymorphic Types

How do we write the types of polymorphic functions? With **type variables**:

```
(: )      :: a -> [a] -> [a]
(++ )     :: [a] -> [a] -> [a]
length    :: [a] -> Int
fst       :: (a, b) -> a
```

We say `length` has type `[a] -> Int` for all types `a`.

When we *apply* a polymorphic function all occurrences are **instantiated** to *the same type*, so `(++)` can have types like:

```
(++)     :: [Int] -> [Int] -> [Int]
(++ )    :: [Char] -> [Char] -> [Char]
```

## Polymorphism or Overloading?

Recall that (+) works on different types: Int, Float, etc.

We said (+) was **overloaded** and had type

$$(+)\ ::\ \text{Num } a \Rightarrow a \rightarrow a \rightarrow a$$

What's the difference between **polymorphism** and **overloading**?

**A polymorphic function has a *single definition* that works on all instances.**

**An overloaded function has *different definitions* for different types.**

## Polymorphism or Overloading? (ctd)

The definition of `fst` above works for any types `a` and `b`.

In contrast, the equality operator (`==`) has a different definition for different types. Equality on `Int` is a built in operation, but equality on *pairs* depends on there being an equality operation on its *component types*, and has a definition:

$$(x, y) == (a, b) = (x == a) \ \&\& \ (y == b)$$

There is another **different** definition of (`==`) for lists, triples, and so on.

*Not every type has an equality operation.*

There is a **class of types** that have equality operations. The type of (`==`) is:

$$(==) \ :: \ Eq \ a \ => \ a \ -> \ a \ -> \ Bool$$