

COM 2001: Advanced Programming Techniques

Semester 1: Functional Programming

Phil Green

1. Introduction

1.1 Programming Paradigms

In computer programming there are a number of distinct **paradigms** (i.e. ways of expressing a programme)

- **Imperative** programming: the programme is a recipe - instructions or **statements** to be executed in sequence. The computer carries out the orders. Some statements result in changes to the **state** of the system, e.g. the data stored in the computer changes. Results are produced by statements which produce output.
- **Object-Oriented** programming: the programme expresses the behaviour of some internally-defined class of **objects** (e.g. windows in a graphical user interface). The code is still imperative but rather than a single recipe we have a number of **methods** each one giving a recipe for performing one task associated with the object class (re-sizing a window, or iconising it, or closing it).
- **Applicative** (or **declarative**) programming: the programme expresses knowledge about some problem domain. To use the programme we supply a problem, to which the knowledge is then applied, yielding an answer to the problem. *The state of the system is unchanged.* Applicative programming in turn subdivides into
- **Logic** programming in which the scheme for representing the knowledge is based on mathematical logic, e.g. predicate calculus.
- **Functional** programming, in which the knowledge representation scheme is based on functions (which we are about to meet).

Particular programming languages are designed for work within particular paradigms¹, for instance

- Pascal, Modula-2, ADA, C and BASIC are imperative languages
- Java and C++ are object-oriented languages
- Prolog is a logic language
- ML, Miranda, CLEAN and Haskell are functional languages

The different paradigms are suited to different types of programming problem.

1.2 Advantages of functional programming

It is claimed that...

- FP allows the programmer to work at a higher level of abstraction,
- FP enforces clean programme design,
- FP has important links to formal methods in computer science ,
- Functional programmes are generally shorter, more elegant and more robust than imperative programmes.
- In FP, the system can do more to spot errors in your code before it runs.
- Programmers using FP beat those using other paradigms in competition (see www.crista.inria.fr/ICFP2001/prog-contest)

At the end of the course, we'll revisit these claims.

1.3 A brief history of Functional Programming²

- Mathematical foundations developed by Alonzo Church (1930s) - Lambda Calculus, Haskell Curry (1950s).
- Lisp first implemented by John McCarthy, 1960.
- Lisp became the 'language of AI'. Industry standard Common Lisp agreed ~1985. Object system CLOS added ~1990.
- Academic alternatives to Lisp developed 1980...2001: ML, Miranda, CLEAN
- COM2001 will use Haskell, first specified late 1980s, last standard 1998, now the prevailing functional language.
- Nowadays Haskell is probably the most popular Functional language in academia, and it is used increasingly in industry: https://wiki.haskell.org/Haskell_in_industry lists over 100 companies using Haskell.

¹The nearest thing to a language which supports all the paradigms is LISP, which is a functional language with an object system and imperative capability.

²For more details, see MacLennan, FP - Practice and Theory, Addison-Wesley, pp23-26

- Some features of Functional Programming have been built in to other languages, e.g. Python & java.

1.4 Learning Resources

Textbooks:

- **Haskell: the craft of functional programming (2nd edition)**, Simon Thompson, Addison Wesley
- **Real World Haskell**: Bryan O'Sullivan, Don Stewart, and John Goerzen, on-line at <http://book.realworldhaskell.org/>
- **The Haskell WikiBook**, on line at <https://en.wikibooks.org/wiki/Haskell>
- Short Introduction to Functional Programming and Lambda-calculus, Franco Barbanera, Universita Catalonia
<http://www.dipmat.unict.it/~barba/PROG-LANG/PROGRAMMI-TESTI/READING-MATERIAL/ShortIntroFPprog-lang.htm>

Software

- The Haskell web site: www.haskell.org from where you can download the Haskell Platform
- The software we will use from the Haskell Platform is **GHCI**, an interpreter . It's copied onto the departmental teachnet image & the CICs desktop. It comes complete with documentation for the Haskell language. There is a guide on the web site.
- **Hoogle** (<http://www.haskell.org/hoogle/>) is a useful search engine for the Haskell API.

2. Fundamentals of FP and Haskell

2.1 Definitions, Expressions and Evaluation

A programme in Haskell consists of a number of **definitions**. A definition associates a **name** or **identifier** with a **value** of a particular **type**. A definition will usually have two components.

- The **type definition** specifies the type associated with the name.
- One or more **equations** specify the **expressions** which define how the value is to be computed.

Examples

```
num :: Int -- type defn: the name num is of type Int
num = 3 -- equation: the value of num is 3
```

- names begin with a lower case letter
- `::` is the typing operator: read it as ‘*is of type*’
- comments start with `--` (ignore rest of line). Longer comments look like so:

```
{- ignore this -}
```
- **Int** is the type of integers³. Type names must start with an upper case letter.
- 3 is a **Literal**: an instance of a basic type.
- The **expression** following `=` is **evaluated** to find the value of **num**. Trivially, literals (and other data structures) evaluate to themselves.
- This definition makes **num** a constant. Unlike in imperative languages we can’t change its value (*because this would be a change of state*).

```
num2 :: Int
num2 = quot 28 (num1 + 4)
```

- This expression uses the system-defined function **quot** (quotient). To use (or **call** a function you simply give its name followed by its **arguments**, separated by spaces⁴.
- The arguments might themselves be expressions, which must be evaluated first. `+` is an **operator**: like a function but placed between its arguments rather than after them.
- The usual rules of precedence (BODMAS) hold, so to set **num2** to 4 we need the brackets.
- We write programmes by defining our own functions, which make use of existing functions...

2.2 Functions

A function can be thought of as a definition which expresses a solution to a set of problems.

We write programmes by defining functions. For instance, we’ll define a function which expresses Pythagoras’ Law: it will compute the length of the hypotenuse of a right-angled triangle if we give it the lengths of the other two sides. In English, we might say

³There is another type **Integer**, if you need greater precision

⁴so-called ‘quiet’ syntax

`pyth` is the function which, when given two numbers `a` and `b`, delivers

The result of the square root function when it is given

The result of the plus function when it is given

The result of the square function when it is given `a`,
and

The result of the square function when it is given `b`.

Functional programming languages, like Haskell, allow us to say this much more concisely.

2.2.1 What's in a Function?

function name <i>pyth</i>				
arguments	parameters	environment	body	value
3	<i>a</i>	<i>a</i> <-- 3		5
4	<i>b</i>	<i>b</i> <-- 4		

A function is defined by

A set of parameters (`a` and `b` above), and

A function body, expressed in terms of the parameters.

The function body defines the computation which the function performs to return its **value**.

A function usually (but not necessarily) has a **name**. (`pyth` in this example).

2.2.2 Calling a Function

To make use of an existing function (to **call** or **apply** it), we create an **expression** which identifies the function & supplies **arguments**, which are associated with the function's parameters. We say that **the parameters are bound to the arguments**. The function body is then **evaluated**, and the result is returned as the **value** of the function.

e.g. if `pyth` is called with arguments 3 and 4,
 `a` is bound to 3
 `b` is bound to 4
 The body evaluates to 5
 the value 5 is returned.

Note that

1. A **function body** is expressed in terms of **function calls using the parameters**.
2. When a function is called, the **value is returned to the caller**: When `plus` is called in `pyth`, its value (25) is returned to the call of `square-root`, which returns its value (5) to `pyth`, which returns it to the outside world.
3. **Bindings persist only as long as the function call which creates them**. Thus there are **no side effects**: when our call of `pyth` terminates, everything is in the state it was at the time the call was made. This property, called **referential transparency**, has important consequences (see later).
4. **In a function call, we normally supply as many arguments as there are parameters in the function definition, and they will be bound in the order given**. The set of bindings in force at a particular time is called the **environment**. Supplying too many arguments is an error. Supplying too few is interesting...see later.
5. **Any sensible computation can be expressed using the above scheme**: this can be proved theoretically (proof related to the Church-Turing theorem).
6. Most programming languages allow the user to write functions, but do not make it easy to express everything in a functional way.
7. The above constitutes an **informal description of pure functional programming**. Haskell (and other functional languages) allow you to express yourself in a variety of ways, but the system translates what you write into code which fits this model.

2.2.3 TYPES

It only makes sense to supply `pyth` with two numbers as its arguments, i.e. every argument to a function must be of the appropriate **type**. A type can also be associated with the value a function returns - `pyth` will always return a number.

Haskell is a **strongly-typed language**. This means that every construct you define in your programme has to be given a type. The language **type system** checks that your typing makes sense before the programme runs.

2.3 An Example Programme

Here's Haskell code for the **pyth** function:

```
{-*****  
    pyth.hs  
    pythagoras law  
*****-}  
module Pyth where  
-- type definition  
    pyth :: Float -> Float -> Float  
-- function definition  
    pyth a b = sqrt(a*a + b*b)
```

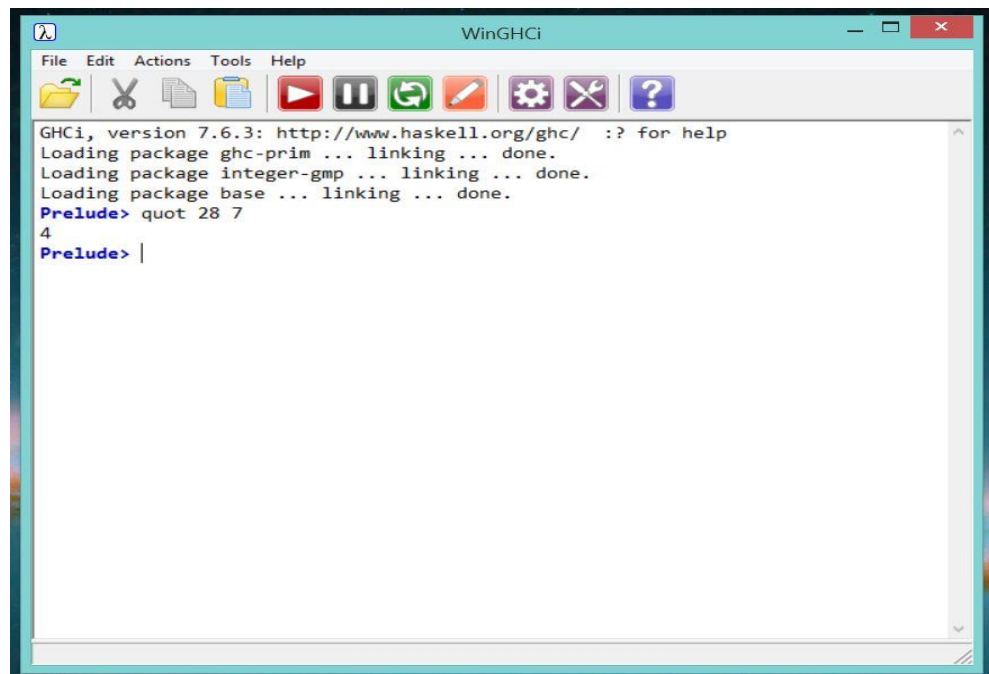
Let's take each line in turn:

1. A Haskell programme consists of a number of **modules**, each containing code for one or more functions. Here we are defining a module **Pyth**. The above code is the content of a file **pyth.hs** *Module names must begin with a capital letter but file names don't have to.*
2. **pyth :: Float ->Float -> Float** is s the **type declaration** or **signature** for the **pyth** function. In this,
 - **Float** is the type of floating point numbers (real numbers, loosely).
 - The type definition gives the types of the arguments in turn and finally the value type. **->** separates the elements of the definition (you'll see the reason for this syntax later).
 - The whole type declaration can be read as '*pyth takes a Float and a Float and returns a Float*'.
 - If you miss out the type declaration, Haskell will try to infer what it should be. In simple cases you may get away with this but it's good practice to always give a type declaration.
3. **pyth a b = sqrt (a*a + b*b)** is the function definition or **equation**. Each function is defined by one or more equations, which closely resemble equations in algebra. In this case we only need one.
 - **pyth a b** means '*we are defining the function pyth with parameters a and b*'. The function name is followed by the parameters, separated by spaces.
 - **=** means '*is defined to be*'.
 - mathematical expressions are written in a way which resembles familiar algebra, so **sqrt (....)** means '*apply the square root function (from Prelude) to the value of the expression inside the brackets*'

- `*` and `+` do multiplication and addition respectively, and are examples of **operators**. An operator is like a function but is placed **between** its arguments rather than **before** them. We could have used `(a**2 + b**2)` - `**` raises to a power⁵
- We have written an **equation** which defines the **pyth** function. In functional programming, we concentrate on this **declarative** style. We don't need to think about giving orders to the computer.

2.4 Loading and Running Haskell Programmes

- The Haskell Platform includes the Glasgow Haskell Compiler, GHC and an interpreter, GHCi. The windows version of this is WinGHCi
- I'll use WinGHCi in these notes. You can use the compiler if you like but there's no need for what we'll cover. Here's the WinGHCi window...



- After loading the kernel of the Haskell language (Prelude.hs), the interpreter produces a prompt (**Prelude>**) and waits to be given something to evaluate.
- **quot 28 7** is provided for evaluation. When we hit **return** the expression is evaluated, its value is returned and Haskell waits for the next expression.
- The interpreter allows us to evaluate expressions, but not to type in definitions. To do that we must load a file containing our definitions. Here's what happens when we load **pyth.hs** (using **open** in the **file** menu or typing the `:load` command as shown below):

⁵as does ^

- When a module is loaded, the prompt changes to its name.
- We can then call the **pyth** function.
- With the interpreter you can test the function you've loaded as many times as you wish: there's no need to write a separate programme with the tests in.
- As an alternative to the drop-down menus you can supply commands (starting with :) directly to the interpreter - say :? to see a list of them. For instance

:load <module name> loads a module

:type <expr> gives you the type of the expression ,expr>, so

```
*Pyth> :type pyth
pyth :: Float -> Float -> Float
*Pyth>
```

- To debug, edit the code and reload (one click from the panel - the icon with 2 arrows). If you pick the file off winGHCi's **edit** menu or from the module manager icon it will come up for editing in **Notepad**, but you can use any text editor you like.
- In functional programming, *nothing else ever happens*, We define functions, use them (and the standard functions) to evaluate expressions & get the values back.

3. Some Haskell details

Before moving on, let's fill in some detail:

3.1 Naming conventions

- Function names and parameter names must start with a lower case letter, which can be followed by zero or more upper or lower case letters, digits or the symbol `_`. e.g. `pyth1` or `pyth_1` would be OK but `Pyth` would not.
- Type names start with an upper case letter e.g **Float** not **float**.
- Module names start with an upper case letter.

3.2 Layout

Haskell obeys the ‘*offside rule*’: a definition is ended by the first piece of text which lies at the same indentation or to the left of the start of the definition.

So

```
pyth a b =  
    sqrt (a*a + b*b)
```

Is OK, but

```
pyth a b =  
sqrt (a*a + b*b)
```

will produce an error.

You can also separate expressions with `;-`

```
pyth 3 4; pyth 5 12
```

3.3 Numeric Types

The basic numeric datatypes in Haskell are

- **Int** - integer numbers
- **Float** - real numbers.

Haskell will treat a whole number as a float if a function requires it, i.e you can say `pyth 3 4`, you don’t have to say `pyth 3.0 4.0`.

Numbers can be negative but beware of situations like

```
fun 1 -2 3
```

Here we are trying to supply the function **fun** with 3 arguments 1, -2 and 3, but the above will be interpreted as **fun (1-2) 3**. We need to say **fun 1 (-2) 3**.

3.4 Numeric Functions and Operators

- As you'd expect, the arithmetic operators `+`, `-` and `*` work for both integer and real arithmetic. However, different versions are invoked dependent on the argument type: the operators are said to be **overloaded**.
- Other operators and functions for integer arithmetic are **quot**, **rem**, **mod** etc.
- For real arithmetic there are **/**, **log**, **exp**, **sqrt** and all the trigonometric functions.
- You can't mix integer and real arithmetic:

```
an_Int+a_Float
```

```
ERROR - Type error in application
```

```
*** Expression      : an_Int + a_Float
```

```
*** Term            : an_Int
```

```
*** Type            : Int
```

```
*** Does not match : Float
```

- To get over this there are type conversion functions:

```
fromInt :: Int->Float
```

```
ceiling :: Float->Int --round up
```

```
floor :: Float->Int -- round down
```

```
round :: Float->Int -- round to nearest
```

3.5 Constants

We can define constants as well as functions, and then refer to them in function definitions, or in defining other constants:

```
module TempConversion where

freezePtF :: Float
freezePtF=32

bloodTempF :: Float
bloodTempF=98.4

ftoC :: Float -> Float
ftoC f = (f-freezePtF)*5/9

bloodTempC :: Float
bloodTempC=ftoC bloodTempF
```

Remember **we can't change the value of a constant** (i.e. unlike imperative languages, there are no assignment statements).

A constant is like a function which takes no arguments.

4. Taking Decisions

4.1 Booleans and Predicates

As in most other languages, conditionality in Haskell is based on truth values.

The basic type **Bool** takes the Boolean values **True** and **False**. Functions or operators which perform tests return **Bools**. For example, there are numeric comparison operators **<**, **>**, **<=**, **>=** :

```
four_figure_Int :: Int -> Bool
four_figure_Int i = i >= 1000

four_figure_Int 1001 -- returns True
```

A function which performs a test and returns a **Bool** is called a **predicate**.

Another important predicate is **==** , the test for identity [**2==2 is True**].

The following logical operators are available for **Bools**: **not**, **||** (or) , **&&** (and).

4.2 If

We want functions which define their return values differently for different cases. Haskell provides several ways of doing this, but the basic one is the **if** construction:

```
my_min :: Int -> Int -> Int
my_min n1 n2 = if (n1<n2) then n1 else n2
```

- I've called this function **my_min** because there already is a system function **min**.
- **if** is followed by a predicate, a then-clause, defining the answer when the predicate returns **True**, and an else-clause, the answer when the predicate returns **False**

- **if** is **lazy**: it will **either** evaluate the then-clause or the else-clause, but not both. This contrasts with **greedy** function evaluation, where **all** the arguments are evaluated and the function body is then evaluated in the resulting environment.
- Apart from efficiency, there is a good reason why **if** should be lazy: if the predicate returns **True** the else-clause might not make sense, and vice-versa. Consider trying to avoid dividing by zero.
- In general, Haskell has a lazy evaluation strategy: evaluations are only done when they are needed. More about this later.
- For taking **n-way decisions**, we can use nested **ifs**:

```
min_3 n1 n2 n3 = if n1<n2 then
                    (if n1<n3 then n1 else n3)
                  else
                    (if n2<n3 then n2 else n3)
```

The brackets round the inner ifs could be left out in this case but they make the expression easier to read.

There is also a **case** construction, but we'll see neater ways of writing n-way decisions later.

5. Polymorphism and Type Classes

The **min** example above was defined for **Ints**, but the same logic should apply to finding the min of **Floats** - can we avoid having to write a separate function? Yes, we can, using **type variables**:

```
my_min :: Ord a => a -> a -> a
my_min n1 n2 = if n1<n2 then n1 else n2
```

Here **a** is a **type variable**. We are saying that **my_min** accepts two arguments which must be of the same type. It returns a result of that type, whatever it is.

Functions which have type variables are said to be **polymorphic**. When such a function is called, the type variables are bound to a **concrete type** for the duration of the call. i.e. in

```
my_min 3.3 3.2
```

the concrete type of **a** is **Float**.

However, there is a restriction. **my_min** only makes sense if the operator **<** works for the concrete type. The construction **Ord a =>** does this: it says that the concrete type of **a** must

be a member of the **type class Ord**. The part before the `=>` is called the **context**, and we can read the whole definition as *'if the type `a` is in the class `Ord` then `my_min` has type `a->a->a`'*.

Ord is the class of ordered types, which includes **Int** and **Float**, and for which `>`, `<`, `<=` etc. are defined. Similarly,

- **Eq** is the type class supporting the identity operator `==` and 'not equal', `/=`. Types of class **Ord** will also support the functions defined for **Eq**, i.e. **Ord** inherits from **Eq**.
- **Enum** allows the type to be enumerated.
- **Show** allows instances of the type to be converted to strings (for printing).
- **Read** allows instances to be read from strings.

Show and **Read** are illustrated in the next section

6. Characters and Strings

Char, the type of individual characters as they appear on the keyboard, is the basic type for textual processing. **Chars** are written in single quotes e.g. `'v'`.

Identity (`==`) and comparison operators are overloaded to work with **Chars** as well as **Ints**, **Floats** and **Bools**. Comparison uses alphabetic ordering:

```
my_min 'v' 'w' ~> 'v'
```

A **String** is a sequence of **Char**, written in double quotes e.g. `"querty"`. Again, identity and comparison operators work for **Strings**, and you can concatenate (join together) two strings with the operator `++` i.e.

```
(my_min "abc" "abd") ++ (my_min "de" "dx")
~> "abcde"
```

show converts other types into Strings:

```
show(2+3) ~> "5"
show(True||False) ~> "True"
```

`~>` is my shorthand for 'returns' - not part of Haskell.

read converts from **String** to other types, but you have to say what the other type is:

```
(read "2") :: Int ~> 2
```

7. Lists

Int, **Float**, **Bool** and **Char** are basic types. **Structured types** allow you to build data structures from the basic types. **String** is an example of a structured type, **List** is another. A **List** is a sequence of items, **which are all of the same type**. The length of a list does not have to be pre-specified. **Lists** are written like so:

```
[item1,item2,.....]
```

So the following are all valid **Lists**, with their type declarations:

```
li :: [Int] -- i.e. a List of Ints
li=  [1,2,3]
lb :: [Bool]
lb=[True, False, False, True]
ls :: [String]
ls= ["qwer"] -- a list of one item
lli :: [[Int]] -- a List of Lists of Int
lli=[[1,2,3],[4,5],[6]]
```

Notice from the last example that we are allowed to embed lists to any level that we like.

The **empty List** (a **List** with zero items) is written `[]`.

The predicate **null** tests for an empty **List**.

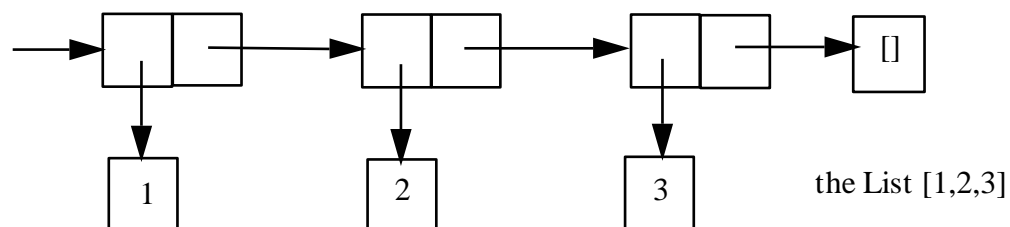
7.1 List Construction Shorthand

The following are convenient ways of defining list sequences:

```
[1..5] ~> [1,2,3,4,5]
['a'..'d'] ~> ['a','b','c','d'] i.e. "abcd"
[1,4,..12] ~> [1,4,7,10]
```

7.2 List Storage Structure

Internally, **Lists** are represented like so:



The arrows represent **pointers**, i.e. memory addresses. These structures are called *linked lists* in other programming languages. The incoming pointer is to the first item or **head** of the list. From this item we point to the rest or **tail** of this list.

7.3 Operations on Lists

Here are some common operations on **Lists**. There are more

- `==` can be used to test List identity
- `<` etc. make list comparisons (they compare heads)
- **length** returns the number of items in a list e.g. `length lb ~> 4` (read `~>` as 'returns').
- **!!** returns the nth item in a list, counting from 0 i.e. `[2,3,4,5]!!2 ~> 4`
- `++` concatenates lists (of the same type) i.e. `[2,3,4] ++ [5,6] ~> [2,3,4,5,6]`
- **head** and **tail** return the head and tail

```
head li ~> 1
```

```
tail li ~> [2,3]
```

8. Recursion

Many programming problems involve repetitive computation (for instance processing each item in a **List** in turn). In imperative programming, the usual way of doing this is by **iteration** - arranging for statements to be repeated. In functional languages we use **recursion** - defining the solution to a problem in terms of the solution to a simpler version of the same problem. *We think about how to define the answer, rather than what the computer will do.*

The general design technique is

1. **define the trivial (or base) cases**, where the answer is immediately available.
2. **express the general case using recursion.**

Often there will be 2 trivial cases, one representing **success** and one representing **failure**.

8.1 Tail Recursion

For instance, let's define a function **member** whose purpose is to tell us whether a given item is present in a given **List**⁶:

```
member 'b' ['a', 'b', 'c'] ~> True
```

⁶There already is a system function to do this, **elem**. In many cases we will be re-defining existing functions in order to understand how they work


```
member 6 [3,5,2,4] ~> False
```

First we write the type definition. Clearly it doesn't matter what the type of the **List** items is, so **member** should be polymorphic. Its arguments are an **a** and a **List** of **as**, written **[a]**. However, we will need to use **==** to check whether a given item is the one we want. Hence the concrete class of **a** must be **Eq**. **member** will return a **Bool**, so it's a predicate

```
member :: Eq a => a -> [a] -> Bool
```

Now the equation which expresses the function definition. It will start

```
member x lis = if (null lis) then False ...
```

i.e. if the second argument (**lis**) is the empty **List**, the answer is **False**

If the second argument is not empty, then trivially the answer is **True** if the thing we're looking for (**x**) - is identical to the first item in **lis**. So the else-clause to the above **if** is another **if**:

```
(if (x == head lis) then True else ....)
```

This leaves us to deal with the general case, the else-clause to this **if**. If the item we want is not the first in the list, we want to go on and consider the remaining items in the same way. We can define that by saying

```
member x (tail lis)
```

Here's the whole definition:

```
member x lis = if null lis
                then False
                else (if x==head lis
                      then True
                      else member x (tail lis))
```

In English, **member** is given **x** and a **List** of **xs** and delivers:

False if **lis** is empty

True if **x** is identical to the first item in **lis**

otherwise **True** or **False**

depending on whether **x** is a **member** of the rest of **lis**

member x (tail lis) is a **recursive call** to the **member** function. As we said above, the general idea of recursive calls is to express the problem in terms of simpler problems. In

List processing, the simpler problems are usually obtained by considering the **head** and **tail** of the **List** you are given. In writing the code, the question to ask yourself is

If I have the solution to the simpler problem, how do I define the answer to the original problem?

In this case, the answer is easy:

The answer to the original problem is the answer to the simpler problem.

This kind of definition (where the recursive call delivers the answer directly) is called **tail recursion**⁷, because the answer is what we get when we process the tail of the list.

Here's a second example of a tail-recursive function, which tests whether 1 list is a **prefix** of another list:

```
prefix [1,2,3] [1,2,3,4,5] ~> True
prefix "qw" "qwerty" ~> True
prefix "qw" "qsd" ~> False
```

Trivial cases:

- if lis1 is empty return **True**.
- if lis2 is empty return **False**
- if heads are not identical then **False**

General Case: if heads are identical then return **True** if tail of lis1 is a prefix of tail of lis2

```
prefix :: Eq a => [a]-> [a]-> Bool
prefix lis1 lis2 =
  if (null lis1)
    then True
    else (if (null lis2)
            then False
            else (if (head lis1)==(head lis2)
```

⁷It can be shown that any tail-recursive function can be written as an iteration, but this isn't the case for recursion in general.

```
then prefix (tail lis1) (tail lis2)
else False))
```

Nested if-then-else constructions become tedious. We're about to address that..

8.2 General Recursion

As an example of a recursive function which isn't tail-recursive, let's define the **length** of a **List**. Again, this is a system function, so we have to rename:

```
my_length :: [a] -> Int
my_length lis = if (null lis)
                  then 0
                  else (1 + my_length (tail lis))
```

Now the answer to

If I have the solution to the simpler problem, how do I define the answer to the original problem?

Is

I add 1 to it

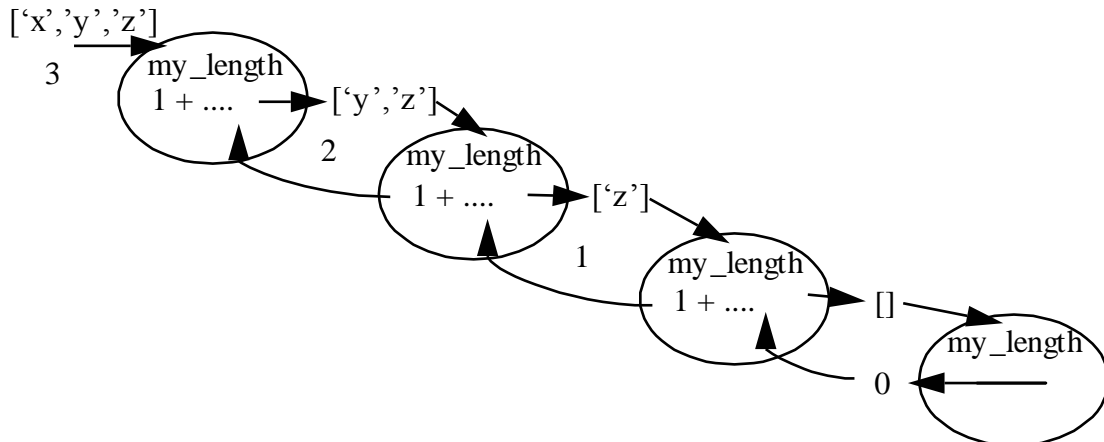
In general recursion the values delivered by recursive calls are themselves part of further expressions.

8.3 Understanding Recursion

There are several ways in which you can get your head round recursion:

8.3.1 Tracing Recursive Calls

Suppose we evaluate `my_length ['x','y','z']`. Internally, something like this will happen:



Each oval represents a recursive call. *Each call is its own ‘world’, independent of everything else.* The arrows indicate the arguments to each call and what each call returns.

This is called **tracing** the execution of a programme. Most functional programming environments provide the programmer with a trace facility. In Haskell the tracing tools are in a library **Debug.Trace** (i.e. you say **import Debug.Trace**).

8.3.2 Algebraic Substitution

Because equations in Haskell are very much like equations in algebra, we can manipulate them symbolically. So to work out `my_length ['x','y','z']` we can substitute as follows

```

my_length ['x','y','z'] = 1+my_length(tail ['x','y','z'] )
                        = 1 + my_length ['y','z']
                        = 1 + 1 + my_length ['z']
                        = 1 + 1 + 1 + my_length []
                        = 1 + 1 + 1 + 0
                        = 3

```

Haskell evaluates expressions in much this way. The technique is called **rewriting** or **graph reduction**.

As you become more familiar with this style, you will find less and less need to think in terms of what the programme will do (tracing) rather than the logic of the definition (algebra).

8.4 Practice with Recursive Functions

You need to practice writing recursive functions until you are confident. Here are some system functions for which you should be able to write your own definition. We will do some of them in class

```

nth                nth item in a List (!!)

```

append	joining operator (++)
subst	substitution e.g. subst 'x' 'y' ['q','x','r','x','s'] ~> ['q','y','r','y','s']
intersection	e.g. intersection [2,5,7] [9,7,3,5] ~>[5,7]
union	e.g. union [2,5,7] [9,7,3,5] ~> [2,5,7,9,3]
reverse	e.g. reverse [4,5,6,7] ~> [7,6,5,4]

SOME TIPS

- Think of **defining what the answer is**, rather than what will happen when the function is called.
- Write the **type declaration first**.
- Write the **trivial cases next**.
- In the general case the **recursion must break the problem down into 'simpler' cases**.
- In functions which process lists, this usually involves taking the **head** and **tail** of the list.
- **Imagine that the function is already working**.
- Ask yourself **how the answer should be defined if the recursive call returns what it should**.
- If the arguments are not how you want them for the recursion, use an **auxiliary function**.

9. Function definition using multiple equations and Patterns

Up to now, we have used only one equation to define each function, and the parameters have simply been variable names. In fact we are allowed to use several equations, where each equation triggers on a different parameter **pattern** (a variable name is a simple form of pattern). Here are our three exemplar functions written in this style:

```
-- member
member :: Eq a => a [a] -> Bool
member x [] = False
member x (h:t) = if (x==h) then True else (member x t)

-- prefix
prefix :: Eq a => [a] -> [a] -> Bool
prefix [] _ = True
prefix _ [] = False
prefix (h1:t1) (h2:t2) =
    if (h1==h2) then prefix t1 t2
    else False

-- length
my_length :: [a] -> Int
my_length [] = 0
my_length (_,rest) = 1 + my_length rest
```

- The operator **:** (sometimes called **cons**) is a constructor for lists. Provided its arguments are compatible, **h:t** returns a list whose head is **h** and whose tail is **t**. Any list can be constructed using cons: **1:2:3:[~]>[1,2,3]**
- When used in equations, it is necessary to put the pattern in parenthesis e.g. **(h:t)** to override the binding to the function application.
- Haskell searches through all the equations for a function **in the order they have been defined** until it finds one whose parameter pattern fits the arguments given.
- It then binds the variables in the pattern accordingly, so that if we call **member 2 [1,2,3]**, **h** is bound to **1** and **t** to **[2,3]**.
- We can then use these bindings in the r.h.s of the equation (rather than calling **hd** and **tl** as before).
- In identical, the equation

```
identical [] _ = False
```

means ‘if the first argument is the empty list then the answer is False no matter what the second argument is’. `_` is the anonymous variable: it matches with anything. Similarly, in `length`, we don’t need to refer to the `hd` in the r.h.s. Hence the pattern `(_:rest)`.

- **You can use any pattern that makes sense:** for instance the following will sum the items in any numeric list of length up to 3

```
sumlist :: Num n => [n] -> n
```

```
sumlist [] = 0
```

```
sumlist [x] = x
```

```
sumlist [x,y]=x+y
```

```
sumlist [x,y,z] =x+y+z
```

but it will fail for longer lists.

- The advantage of using this style is that the pattern matching takes care of decision making for us. The idea is to have a separate pattern for each case we have to consider.
- However, you can’t use the same variable name twice in a pattern (e.g. `member x [x,t]=True` won’t work).

10. Function definition using Guards

Another alternative to `if` is to split the r.h.s of an equation into a number of **cases**, each one defining the answer in some condition that we test for. The tests associated with each case are called **guards**. Here are our three functions again, in this style and retaining pattern matching:

```
-- member
```

```
member :: Eq a => a-> [a] -> Bool
```

```
member x [] = False
```

```
member x (h:t)
```

```
  | x==h = True
```

```
  | otherwise = member x t
```

```
-- prefix
```

```
prefix :: Eq a => [a]-> [a]-> Bool
```

```
prefix [] _ = True
```

```
prefix _ [] = False
```



```

prefix (h1:t1) (h2:t2)
  | (h1==h2) = prefix t1 t2
  | otherwise = False

-- length
my_length :: [a] -> Int
my_length lis
  | null lis = 0
  | otherwise = 1 + my_length t
                    where (_:t) = lis

```

- each case has a new line, indented at least 1 place
- *note that there is no initial =*
- each case looks like | {**predicate**} = {**expression**}
- **otherwise** is a dummy predicate which returns **True** (we could just use **True**)
- in the last example,
 - the test for the empty list has been expressed by a guard
 - **where** allows a local addition to the environment. Notice that a pattern is used here too. More about this later.

It's good style to make as much use of patterns as you can, trying to have a separate equation to deal with each case of interest.

11. Examples of Recursive Functions on Lists

- **nth**

Returns the *nth* item in a List, counting from 0 i.e. **nth 2 [4,5,6,7] ~> 6**

```
my_nth :: Int->[a] -> a
my_nth n lis
  |length lis <= n = error "list too short"
  |otherwise = my_nthA n lis
```

```
my_nthA :: Int->[a] -> a
my_nthA 0 (first : _) = first
my_nthA n (_ : rest) = my_nthA (n-1) rest
```

- We should check that **n** is not greater than the length of the list. The question is what to do when this kind of thing happens. **my_nth** has to return an **a**, but that's not appropriate here.
- One thing we can do is to abort the computation. **error** stops everything & prints the string you give it in the interpreter window. That will do for now but in general it's unsatisfactory: **my_nth** might be called from within some other function and we don't want to abandon everything. We'll see a better method later.
- We only want to make this test on the length of a list once. Hence the **auxiliary function my_nthA** is called to do the work if the test is OK.

11.2 Remove

remove removes all occurrences of a given item in a given list, e.g. **remove 'q' ['a','q','b','q','c'] ~> ['a','b','c']**

```
remove :: Eq a => a->[a] -> [a]
remove _ [] = []
remove x (h:t)
  | x==h = remove x t
  |otherwise = (h:remove x t)
```

This illustrates the use of a pattern in the r.h.s. of an equation. The answer in the last case is **[h:remove x t]** i.e. the List whose head is **h** and whose tail is **remove x t**, the result of removing **x** from the remainder of the List, **t**.

- **Reverse**

reverse reverses the elements of a **List** i.e. **reverse** [1,2,3,4] ~> [4,3,2,1]

Here's how to do it using an auxiliary function:

```
my_reverse :: [a] -> [a]
my_reverse lis = my_reverseA lis []

my_reverseA :: [a]->[a] -> [a]
my_reverseA [] out = out
my_reverseA (first:rest) out =
    my_reverseA rest (first:out)
```

Here we want two parameters to manage the recursion, but our user should only have to give one. We hide the other with an auxiliary function.

Here's an alternative, illustrating the use of the list concatenation operator ++

```
rev :: [a] -> [a]
rev [] = []
rev (h:t) = rev t ++ [h]
```

- We couldn't say (**rev t : h**) there because the tail of a **List** must be a **List**, and **h** isn't.
- Similarly, (**rev t : [h]**) is wrong: **rev t** is a **List** of **as**, not the head of such a **List**.

- **Append**

++ is called **append**. We could define our own:

```
append :: [a]->[a] -> [a]
append [] lis2 = lis2
append (f:r) lis2 = (f:append r lis2)
```

- this shows that **append** copies its first argument. It returns a **List** with the concatenation of the elements of its arguments, but their values remain unchanged, preserving referential transparency.
- Because of this, it's more efficient to use **cons** to add elements to the head of a list than to use **append** to add elements to the end of a list, and this effect will increase with the length of the lists that have to be copied:

```

> my_reverse [1,2,3,4,5,6,7,8,9,10,11,12,13,14,15]
[15,14,13,12,11,10,9,8,7,6,5,4,3,2,1]
(180 reductions, 313 cells)
> rev [1,2,3,4,5,6,7,8,9,10,11,12,13,14,15]
[15,14,13,12,11,10,9,8,7,6,5,4,3,2,1]
(299 reductions, 642 cells)

```

11.5 Intersection

This is the intersection of two Lists representing sets, i.e. with no repeated elements:
intersection [1,2,3] [2,3,4,5] ~> [2,3]

```

intersection :: Eq a => [a]->[a] -> [a]

intersection [] _ = []
intersection _ [] = []

intersection (h1:t1) lis2
  | elem h1 lis2 = (h1:intersection t1 lis2)
  | otherwise = intersection t1 lis2

```

TABLE 1. Haskell List Functions in Prelude.hs

Function/Operator	Type Definition	Effect	Example
:	$a \rightarrow [a] \rightarrow a$	cons operator: add item to head	$3:[2,1] \rightsquigarrow [3,2,1]$
++	$[a] \rightarrow [a] \rightarrow [a]$	append operator: concatenate 2 lists	$[1,2]++[3,4,5] \rightsquigarrow [1,2,3,4,5]$
!!	$[a] \rightarrow \text{Int} \rightarrow a$	nth operator	$['a','b','c']!!1 \rightsquigarrow 'b'$
concat	$[[a]] \rightarrow [a]$	flattens a list of lists	$\text{concat } [[1,2],[3],[4,5]] \rightsquigarrow [1,2,3,4,5]$
length	$[a] \rightarrow \text{Int}$	length of a list	$\text{length } [[1,2],[3],[4,5]] \rightsquigarrow 3$
head	$[a] \rightarrow a$	first item in a list	$\text{head } [1,2,3] \rightsquigarrow 1$
last	$[a] \rightarrow a$	last item in a list	$\text{last } [1,2,3] \rightsquigarrow 3$
tail	$[a] \rightarrow [a]$	tail of a list	$\text{tail } [1,2,3] \rightsquigarrow [2,3]$
init	$[a] \rightarrow [a]$	list of all but the last item in a list	$\text{init } [1,2,3] \rightsquigarrow [1,2]$
replicate	$\text{Int} \rightarrow a \rightarrow [a]$	form a list of n copies of arg 2	$\text{replicate } 3 \text{ 'x'} \rightsquigarrow ['x','x','x']$
take	$\text{Int} \rightarrow [a] \rightarrow [a]$	take first n elements from the head	$\text{take } 2 [1,2,3] \rightsquigarrow [1,2]$
drop	$\text{Int} \rightarrow [a] \rightarrow [a]$	drop n elements from the head	$\text{drop } 2 [1,2,3] \rightsquigarrow [3]$
splitat	$\text{Int} \rightarrow [a] \rightarrow ([a],[a])$	split a list at nth position, return pair of lists	$\text{splitat } 3 [1,2,3,4,5] \rightsquigarrow ([1,2,3],[4,5])$
reverse	$[a] \rightsquigarrow [a]$	return list reversed	$[1,2,3] \rightsquigarrow [3,2,1]$
zip	$[a] \rightarrow [b] \rightarrow [(a,b)]$	take a pair of lists to a list of pairs	$\text{zip } [1,2] ['a','b','c'] \rightsquigarrow [(1,'a'), (2,'b')]$
unzip	$[(a,b)] \rightsquigarrow ([a],[b])$	take a list of pairs to a pair of lists	$\text{unzip } [(('a',1), ('b',2))] \rightsquigarrow (('a','b'), [1,2])$
and	$[\text{Bool}] \rightarrow \text{Bool}$	conjunction of a list of Bools	$\text{and } [\text{True}, \text{False}, \text{True}] \rightsquigarrow \text{False}$
or	$[\text{Bool}] \rightarrow \text{Bool}$	disjunction of a list of Bools	$\text{or } [\text{True}, \text{False}, \text{True}] \rightsquigarrow \text{True}$

sum	[Int]->Int [Float]->Float	sum of a numeric list	sum [4,5,6]~>15
product	[Int]->Int [Float]->Float	product of a numeric list	product [4,5,6]~>120

- **Local Definitions**

In imperative programming, computations are often split into a sequence of intermediate steps:

```
x = some computation
y = some other computation
z = some computation which involves x and y
....
```

This is done not only to improve clarity but, crucially, to **avoid repeated work**. The values of **x** & **y** are still available for subsequent computation: they needn't be re-computed.

The functional equivalent of this is to **add new bindings to the environment** that was created by the function call. Haskell provides **let** and **where** constructions for doing this. For instance, let's compute the roots of a quadratic:

$$ax^2 + bx + c = 0$$

$$x = \frac{-b \mp \sqrt{b^2 - 4ac}}{2a}$$

```
roots :: Float->Float->Float -> [Float]
roots a b c =
  let s = sqrt (b*b -4.0*a*c)
      d = 2.0*a
  in [ (-b+s)/d, (-b-s)/d]
```

- **let** is followed by any number of **local definitions**.
- **in** separates these from the main expression.
- **roots** doesn't deal properly with imaginary roots.

Here's the alternative using **where**

```
roots a b c = [ (-b+s)/d, (-b-s)/d]
              where s = sqrt (b*b -4.0*a*c)
                    d = 2.0*a
```

- where you have a sequence of expressions within a **let**, earlier bindings are available later on:

```
try_let :: Int->Int -> Int
  try_let p q = let r = p+q --binding for r
                  s = p*q --binding for s
                  t = r+s   --binding for t uses r & s
                  in t --returning t
```

```
try_let 2 3 ~> 11
```

- this looks like imperative programming but it isn't. Each expression in the **let** is adding a new binding to the environment. Behind the scenes everything is functional.
- Therefore the following won't work:

```
try_let p q = let r = p+q
                s = p*q
                r = r+s --reusing r
                in r
```

because **r=r+s** would be destructive. The error message is *"r" multiply defined*.

13. Tuples

Tuples are the counterpart to **Lists**. **Lists** have any number of elements, all of the same type. **Tuples** have a fixed number of elements of different types. **Tuples** are written with **()**s rather than **[]**s. e.g.

(1,'a') is a tuple of type **(Int,Char)**

([1,2], True, sqrt) is a tuple of type **([Int],Bool, Float -> Float)**

It's better to use tuples rather than lists when you know how many elements there are going to be: for instance in **roots** example above there are always two results:

```
roots :: Float->Float->Float -> (Float, Float)
roots a b c = let s = sqrt (b*b -4.0*a*c)
                d = 2.0*a
                in ((-b+s)/d, (-b-s)/d)
```

For tuples of length 2 (**pairs**) there are functions **fst** (first item) and **snd** (second item). More generally, you use patterns to extract what you want, e.g.

```
let
```



```
(a,b,c) = fun .....
```

Here fun returns a 3-tuple whose components will be bound to a,b and c.

14. Defining your own Types

14.1 Type Synonyms

We can define our own types to represent structures built from existing types.

- **An Example: Property Lists**

Suppose we want to work with information about some well-known people:

```
"name" "Green" "room" "105" "phone" "21828" ....  
"name" "Fairtlough" "phone" "21826" "group" "V&T" ...  
"name" "Mendler" "grade" "reader" "email" "m.mendler" ...
```

Each of these lines is a **property list (plist)**: it contains a number of **items** of information, each introduced by a property (**prop**), followed by its **value**.

- For simplicity, we'll assume that all **props** and **values** are **Strings** (but we could use **read** to convert into other types).
- Different **plists** may contain different items, and they can come in any order.
- In a given list, **properties are unique**: we can only have 1 item for a given **prop**.

We will write a module **proplists** to handle property lists. We assume that users want the following functions:

- **pcreate** returns a new, empty **plist**
- **pput** adds a **prop**, and its **value**, to an existing **plist**, returning the new **plist**.
- **pdelete** deletes a **prop** and its **value** from a **plist** and returns the resulting **plist**.
- **pget** retrieves data from **plists**: i.e. given a **prop** and a **plist**, it returns the corresponding **value** (if the **prop** is present: otherwise we should return an indication that the search failed).

The natural way to represent a property list is by a **List** of **Pairs**. We use the type construct to say this:

```
type Plis = [(String,String)]
```

Here's the proplists module: in this version the computation is aborted if **pget** cannot find the required item.

```

module Proplists where
  --datatype for a proplist
  type Plis = [(String,String)]
  --create an empty Plis

  pcreate :: Plis
  pcreate = []

  --pput

  pput :: String->String->Plis -> Plis
  pput p v pl = ((p,v):pdelete p pl)

  --pdelete
  pdelete :: String->Plis -> Plis
  pdelete p pl
    | null pl = []
    | p==q = rpl --found the item, answer is the rest
    | otherwise = ((q,v):pdelete p rpl)
    where ((q,v):rpl)=pl

  -- pget

  pget :: String->Plis -> String
  pget q pl
    | null pl = error "item not present"
    | q==p = v --found it
    | otherwise = pget q rpl --try the rest of the proplist
    where ((p,v):rpl) = pl

  --example - creating a plist from scratch
  plis1 :: plis
  plis1 = (pput "name" "green"
           (pput "phone" "21828"
                (pput "room" "107" pcreate)))

```

now

```
pget "room" plis1~> "107" etc.
```

- The easy way to write **pput** is to use **pdelete** to remove any old item with the same property, then add the **prop** and **value** tuple to what is returned.
- **pdelete** is similar to **remove**.
- We can now import **proplists**

14.1.2 Type Synonyms for Functions

Type synonyms can be used as shorthand for function type definitions as well as for conventional data structures. Suppose we are writing game-playing programmes, for instance to play chess. We want to investigate different strategies by having programmes play each other. We can define

```
type ChessPlayer = Board->History->Move
```

This says that **ChessPlayer** is a function which takes a board position and the game history (a list of moves) and returns a move (i.e. which piece to move where). We can now write different **ChessPlayers**:

```
simplePlayer :: ChessPlayer
simpleplayer board history = .....
kasparov :: ChessPlayer
kasparov board history = .....
```

14.1.3 Polymorphic Type Synonyms

You can use type variables in the definition of a type synonym. For example, here's an alternative structure for the information in our property list:

```
type Name = String
type Dlis a = [(Name,a)] -polymorphic type synonym

phlis :: Dlis Int -- a is Int
phlis = [("Green",21828),("Simons",21838)]
grlis :: Dlis String -- a is String
grlis = [("Green", "SPandH"), ("Gaizauskas", "NLP")]
tlis :: Dlis [Int] -- a is [Int]
tlis = [("Moore", [4502,6502]), ("Green", [1005, 2001])]
deptData = (phlis, grlis, tlis)
```

14.2 Algebraic Types

Algebraic data type definitions are introduced by the keyword **data**, which is followed by the name of the type, = and then a number of alternative **constructors**, separated by |. The type name and the constructors must start with an upper case letter.

The simplest form of algebraic type just defines a number of literals: for instance, the built-in **Boolean** type is defined as

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

i.e. The literals **False** and **True** are the only valid constructions allowed by this type. Each alternative must begin with a literal.

We can define our own...

```
data Temp = Cold | Hot  
data Season = Spring | Summer | Autumn | Winter
```

... and functions which use them:

```
weather :: Season -> Temp  
weather Summer = Hot  
weather _ = Cold
```

Haskell can associate the new type with existing type classes:

```
data Season = Spring | Summer | Autumn | Winter  
deriving (Eq,Show)
```

Says that Seasons can be compared for identity (**Eq** type class):

```
Summer == Autumn ~> False
```

and print them out (**Show**)

14.2.1 Enumerated types

Enumerated types provide an ordering over the alternatives: if we say

```
data Season = Spring | Summer | Autumn | Winter  
deriving (Eq,Show,Enum)
```

then each constructor except the first has a predecessor and each constructor except the last has a successor:

```
succ Spring ~> Summer
pred Winter ~> Autumn
```

14.3 Data Fields

We can specify information to be associated with each constructor:

```
data Anniversary = Birthday String Int Int Int
                  -- name, year, month, day
                  | Wedding String String Int Int Int
                  -- spouse name 1, spouse name 2, year, month, day
                  deriving Show
```

The **deriving** construction tells Haskell to make the **Anniversary** class an instance of the type class **Show**, i.e. to provide a way of printing **Anniversaries**.

The constructors (**Birthday** and **Wedding** here) are really functions which create **Anniversaries**. To see this, look at the type of **Birthday**:

```
*Main> :type Birthday
Birthday :: String -> Int -> Int -> Int -> Anniversary
```

We can call the constructors directly:

```
johnSmith :: Anniversary
johnSmith = Birthday "John Smith" 1968 7 3
```

Show allows us to print what we've made:

```
> johnSmith
Birthday "John Smith" 1968 7 3
```

We can access data in these structures using pattern matching:

```
JohnSmithBirthYear = y
                    where (Birthday _ y _ _) = johnSmith
```

or more generally

```
birthYear :: Anniversary->Int
birthYear (Birthday _ y _ _) = y
```

This could get awkward if there are a lot of fields, so Haskell has ‘record syntax’:

14.4 Record Syntax (named fields)

Here’s an alternative definition for **Anniversary**

```
data Anniversary = Birthday
    {bName :: String,
     bYear :: Int,
     bMonth :: Int,
     bDay :: Int}
| Wedding
    {wName1 :: String,
     wName2 :: String,
     wYear :: Int,
     wMonth :: Int,
     wDay :: Int}
```

This notation makes it easier to handle data structures with many fields.

This code creates an **Anniversary**:

```
johnSmithWedding :: Anniversary

johnSmithWedding = Wedding
    {Name1= "johnSmith",
     Name2 ="mrsSmith",
     Year = 1988,
     Month = 1,
     Day = 12}
```

We can access fields like so:

```
wName2 johnSmithWedding ~> "mrsSmith"
```

and make modified structures like so:

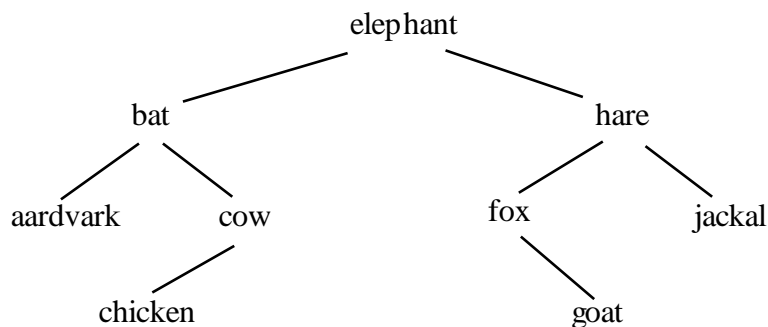
```
johnSmithWedding2 = johnSmithWedding {wMonth = 2}
```

*note that having done that **johnSmithWedding** is unchanged .. it’s not assignment!*

15. TREES

As a further example of defining our own types, and recursive processing with these types, we'll look at **binary trees** (which you may already have met).

In a binary tree, each **node** holds some data item and has a **left-successor** and a **right-successor**, either or both of which might be empty. A node with no successors is called a **leaf**. The 'topmost' node in a tree is called its **root**. Suppose we are using a binary tree to represent a dictionary:



For every node, the left sub-tree contains words which are alphabetically-earlier than the word at the root; the right subtree contains words which are alphabetically-later.

We can define a polymorphic algebraic datatype for binary trees thus:

```
data Tree a = Empty | Leaf a | Node a (Tree a) (Tree a)
```

So a Tree of **a**s is either **Empty**, or a **Leaf** holding a data item, or a **Node** holding a data item and having left and right successors which are both **Trees of a**. So the type definition is itself recursive.

The keywords introducing the different forms of a datatype (**Empty**, **Leaf**, **Node** above) are called **constructors** because you can think of them as fns which create examples of the datatype e.g. **Leaf** takes an **a** and produces a **Tree**:

```
Leaf :: a->Tree
```

Note that we could easily add a second data item to each node, to allow us to store an English-French dictionary, for instance. In each node we'd have a **key** (the English word) and some data associated with the key (the French word). We'd then have something equivalent to a property list, from the user's viewpoint, except that the tree would (usually) be ordered by its keys whereas property lists (usually) are unordered.

The above tree could be constructed directly:

```
animals :: Tree String
```

```

animals = Node "elephant"
          (Node "bat"
              (Leaf "aardvark")
              (Node "cow"
                  (Leaf "chicken")
                  Empty))
          (Node "hare"
              (Node "fox"
                  Empty
                  (Leaf "goat"))
              (Leaf "jackal"))

```

We'll write a family of functions to handle binary trees. *Functions accepting algebraic types will generally require a separate equation for each allowed alternative.*

15.1 Tree Traversal

'*Traversing*' a tree means visiting each node once, in some specified order. For instance, suppose we want to produce a list of all the words in our dictionary, in alphabetic order. From any node **n**, initially the root node, we want to

1. Traverse **n**'s left sub-tree, returning a list of all the words in it in order
2. Append the word at **n** to this list
3. Append the result of traversing **n**'s right sub-tree to this list.

```

traverse :: (Tree a) -> [a]
traverse Empty = []
traverse (Leaf x) = [x] --leaf returns list of 1 item
traverse (Node x left_sub_tree right_sub_tree) =
    (traverse left_sub_tree) ++
    [x] ++
    (traverse right_sub_tree)

traverse animals ~>
["aardvark","bat","chicken","cow","elephant","fox","goat",
,"hare","jackal"]

```

This kind of traverse (visit the left subtree, then the root, then the right subtree) is called an **in-order** traverse.

15.2 Searching a Tree

The following predicate tells us whether a given item is present in a given tree: the tree is assumed to be ordered:

```
tree_member :: Ord a => a -> (Tree a) -> Bool
tree_member _ Empty = False
tree_member y (Leaf x) = (x==y)
tree_member y (Node x left_sub_tree right_sub_tree)
    | x==y = True
    | x>y = tree_member y left_sub_tree
    | otherwise = tree_member y right_sub_tree
```

now

```
tree_member "aardvark" animals ~> True
```

So each recursive call limits the search to either the left or the right subtree. If the tree is ‘balanced’ i.e. there are equal numbers of nodes in left and right subtrees from any node, we will **halve** the number of nodes remaining to search on each call.

We could easily adapt this, for trees with keys and data, to return the data item associated with a given key (i.e. the French translation for an English word). This would be another case where we’d use a **Maybe** type for the result.

15.3 Tree Insertion

To insert a new item into a given tree, we successively recurse left or right finding where it belongs. We wind up either at an **Empty** or at a **Leaf**:

```
tree_insert :: Ord a => a -> (Tree a) -> Tree a

--insert into an Empty tree produces a Leaf
tree_insert y Empty = Leaf y

--insert into a Leaf produces a Node
tree_insert y (Leaf x)
    | x==y = (Leaf x) --already there - leave as it is
    | y<x = (Node x (Leaf y) Empty) --y in left_sub_tree
    | otherwise = (Node x Empty (Leaf y)) --in right_sub_tree

--insert into a Node - insert in left or right sub_tree
```

```

tree_insert y tree
  | x==y = (Node x left_sub_tree right_sub_tree)
  | y<x  = (Node x
              (tree_insert y left_sub_tree)
              right_sub_tree)
  |otherwise = (Node x
                left_sub_tree
                (tree_insert y right_sub_tree))
where (Node x left_sub_tree right_sub_tree) = tree

```

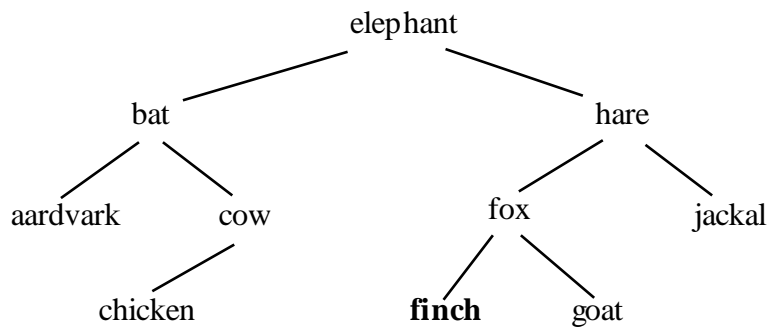
now

```

traverse (tree_insert "finch" animals)~>
["aardvark","bat","chicken","cow","elephant","finch","fox",
,"goat","hare","jackal"]

```

and the tree looks like



16. The Maybe type

The polymorphic enumerated type **Maybe** provides a neat way of dealing with error conditions:

```
data Maybe a = Nothing | Just a
              deriving (Eq,Ord,Read,Show)
```

The last line says that this type should inherit from the **Eq**, **Ord**, **Read** and **Show** type classes.

We can use **Maybe** in cases where a function might not return its usual value because of an error condition:

```
my_nth :: Int->[a] -> Maybe a
my_nth n lis
    |length lis < n = Nothing
    |otherwise = Just (my_nthA n lis)

my_nthA :: Int->[a] -> a
my_nthA 0 (first : _) = first
my_nthA n (_ : rest) = my_nthA (n-1) rest
```

Now

```
my_nth 5 [1,2,3,4] ~> Nothing
my_nth 2 [1,2,3,4] ~> Just 3
```

We can tidy up pget now:

```
pget :: String->Plis -> Maybe String
pget q pl
    |null pl = Nothing --it's not there
    |q==p = Just v --found it
    |otherwise = pget q rpl --try the rest of the proplist
    where ((p,v):rpl) = pl
```

Now

```
pget "room" plis1 ~> Just "107"
pget "age" plis1 ~> Nothing
```

Maybe helper functions

The **Maybe** type is pre-defined but it is worth defining the following to make it easy to deal with results from a function whose return type is a **Maybe**:

```
-- predicate testing for success
isJust :: (Maybe a) -> Bool
isJust (Just _) = True
isJust Nothing = False
-- extract the result from a successful Maybe
resMaybe :: (Maybe a) -> a
resMaybe Just x = x
```

Using these we can write in the following style:

```
fun .....
  | isJust r = fun2 .. (resMaybe r) ..
  | otherwise = ...
  where r = foo ...
```

Here **foo** is a function that returns a **Maybe**. In the outer function **fun** we want to define the answer differently dependent on whether **foo** succeeds or fails. If it succeeds we want to make use of the result it returns.

We achieve this by calling **foo** in the **where** and then using **isJust** to find out whether it worked. If so the answer is defined by a call of **fun2** which uses **resMaybe**. In this way we avoid having to call **foo** twice (once to find out if it finds a result and again to make use of the result).

An example of this is coming up.

Annoyingly, you can't do the equivalent with **let** (i.e. have guards after **let... in**).

There are more Maybe helpers in the library **Data.Maybe**

17.Higher Order Functions

What we have looked at so far is '**First-Order Functional Programming**': functions are viewed as effecting static transformations of input structures to produce output structures. The idea in '**Higher-Order Functional Programming**' is that functions themselves have the same 'first-class status' as other data objects, and can thus be **passed as inputs to** and **returned from** other functions. So the data transformation performed by a higher-order function will depend on what function was passed to it.

17.1 Mapping Functions

A mapping function embodies a **common pattern of recursion**. Once we have the mapping function we don't have to programme this recursion pattern again. Haskell supplies a number of mapping functions: we'll look at **map**, **filter** and **foldr**.

17.1.1 map

We often want to take a list and perform the same operation on each of its members. The mapping function **map** does this for us:

```
map sqrt [9.0,16.0,25.0] ~> [3.0,4.0,5.0]
map length [[1,2,3],[4,5],[6,7,8,9]] ~> [3,2,4]
```

So the first argument to **map** is a function of one argument. This function must accept the type of the list which is the second argument. **map** returns a list of the results of calling the function on each item in the list.

We can easily define our own version of **map**:

```
my_map :: (a->b) -> [a] -> [b]
my_map _ [] = []
my_map f (h:t) = (f h : my_map f t)
```

The type of the first argument is the type of the function (**a->b**)

17.1.2 Anonymous Functions

Suppose we want to add a constant, say 5, to each item in an integer list. We could say

```
add5 :: Int -> Int
add5 n = n+5
```

,and then

```
map add5 [1,2,3] ~> [6,7,8]
```

but it seems pointless to create **add5** if this is all we are using it for. Instead we can use an **anonymous function**:

```
map ( \n -> n + 5) [1,2,3] ~> [6,7,8]
```

here `\` introduces an anonymous function, which is then applied to the following arguments. It's meant to be read as **lambda**, and comes from the **lambda calculus**, the mathematical notation for functions developed by Church and Curry. `\` is followed by the parameters, `->`, and then the body of the function. **Lambdas** are used a lot in conjunction with mapping functions.

17.1.3 filter

filter is what you use when you want to perform a test on each item in a list, and return only those items which pass the test:

```
filter (\n -> (n > 100)) [87, 112, 54, 225] ~> [112,225]
filter (\l->(length l)==2) [[1,2], [3,4,5], [6,7], [8]]
~> [[1,2], [6,7]]
```

Here's a definition of **filter**:

```
my_filter :: (a->Bool)->[a] -> [a]
my_filter _ [] = []
my_filter f (h:t)
    | f h = (h:my_filter f t)
    | otherwise = my_filter f t
```

17.1.4 foldr

A third common requirement is to take a list and produce a single result, as when we want to add up a numeric list, or find its maximum. This is called **folding**⁸. Here's the behaviour of **foldr** (fold to the right):

```
foldr (+) 0 [1,2,3,4] ~> 10
foldr (*) 10 [1,2,3,4] ~> 240
```

- Operators like `+` and `*` can be used as functions by putting `()`s round them⁹.
- The second argument gives the 'base value', what **foldr** should return for an empty list:

⁸or sometimes **reducing**

⁹conversely, functions can be used as operators by putting them in backquotes:

```
'd' `elem` "qwder" ~> True
```

```

my_foldr :: (a->b->b)->b->[a] -> b
my_foldr _ base [] = base
my_foldr f base (h:t) = f h (my_foldr f base t)

```

i.e. on each step, we apply the function **f** to the first in the list and the result of folding the rest of the list:

```

my_foldr (+) 0 [1,2,3]
    = (+) 1 (my_foldr (+) 0 [2,3])
    = (+) 1 ((+) 2 (my_foldr (+) 0 [3]))
    = (+) 1 ((+) 2 ((+) 3 (my_foldr (+) 0 [])))
    = (+) 1 ((+) 2 ((+) 3 0))
    = (+) 1 ((+) 2 3)
    = (+) 1 5
    = 6

```

there is a corresponding **foldl**

17.2 Programming with Mapping Functions

map, **filter** and **foldr** are an extremely useful family of functions, especially when used in combination:

- If you want to perform the same operation on each member of a list, use **map**.
- If you want to perform a test on each member of a list, use **filter**.
- If you want to process a list and return a single result, use **foldr**.

Here are some examples:

17.2.1 Exam Mark Analysis

Suppose we have a list of exam results organised into tuples like so:

```

[("able",55), ("baker", 67), ("charles",37), ("dogbreath",
32) ...]

```

We want to find the average mark and the names of all the students below average:

```

type Mark_list = [(String,Int)] -- define a type

below_av :: Mark_list->(Float,[String])
below_av mlis =

```

```

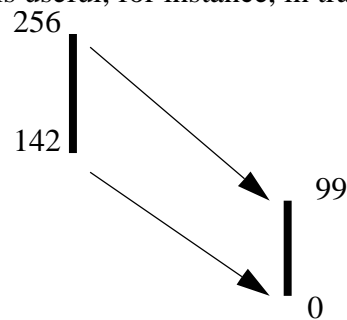
let marks = map snd mlis --use snd to extract the marks
-- use foldr to add the marks up
    av = (fromInt (foldr (+) 0 marks))/
          (fromInt (length mlis))
-- filter the below-av items from mlis
blis = filter (\e -> (fromInt (snd e)) < av)
        mlis

in
    (av,map fst blis) --map with fst to get the names below av

```

17.2.2 Rescale

Here we have a list of numbers. We want to scale these numbers so that they fit within a given range. e.g. if the smallest in the original list was **142** and the largest was **256** and the required range is **0** to **99**, the resulting list should be such that **142~>0**, **256~>99** and the rest are scaled accordingly. This is useful, for instance, in training neural nets, where data



must be scaled to a range **0** to **1**.

```

{- function rescale (rescale numeric data to fit in a given
range)
  parameters
  lo  lower limit of range
  hi  higher .... ..
  dlis the data - list of numbers
  each number n in dlis changes to
    lo + (n - dmin) * (hi - lo) / (dmax - dmin)
  where dmax & dmin are largest & smallest is dlis
-}

rescale :: Float->Float->[Float] -> ([Float],Float,Float)
rescale lo hi dlis =

```



```

let dmax = foldr (\ next max_so_far ->
    if (next > max_so_far)
    then next else max_so_far)
    (head dlis) (tail dlis)
dmin = foldr (\ next min_so_far ->
    if (next < min_so_far)
    then next else min_so_far)
    (head dlis) (tail dlis)
old_range = dmax - dmin
new_range = hi - lo
mfact = new_range/old_range

in ((map (\ n -> lo + (n - dmin)*mfact) dlis), dmax,dmin)

```

17.2.3 Merge

If the mapping function we need isn't already there, we can easily define it, as illustrated by the following example:

Merging data is a common need in computer science. We have two (or, in general, more) lists of data which are already ordered in some way. We want to produce a new list containing all the items in the two original lists, in order, i.e.

[4,9, 15, 21] merged with [3, 11, 18, 25, 33] gives [3,4, 9, 11, 15, 18, 21, 25, 33]

["cat", "fox", "goose"] merged with ["bear", "dog", "hare"] gives ["bear", "cat", "dog", "fox", "goose", "hare"]

The logic of merging is the same no matter what the ordering metric is.. Can we capture it in a mapping function?

```

my_merge :: Ord a => (a->a -> Bool)->[a]->[a] -> [a]

my_merge compfn [] lis2 = lis2
my_merge compfn lis1 [] = lis1
my_merge compfn (h1:t1) (h2:t2)
    | compfn h1 h2 = (h1:my_merge compfn t1 (h2:t2))
    | otherwise = (h2:my_merge compfn (h1:t1) t2)

my_merge (>) [5,4] [3,2,1] ~> [5,4,3,2,1]
my_merge (<) [1,2,3] [4,5] ~> [1,2,3,4,5]

```

```

my_merge (\str1 str2 -> (length str1) < (length str2))
    ["qw","erty"]
    ["q","wer","qwert"]
~> ["q", "qw", "wer", "erty", "qwert"]

```

We could avoid splitting the lists up and then reassembling them using **let** or **where** or using **@** notation (coming later)

17.2.4 MergeSort

Another common requirement is to **sort data into order**. There are many sorting algorithms. One of the best sorting algorithms is **mergesort**. Suppose we start with a list of unordered data (in this case numbers to be sorted in ascending order):

```
[5, 3, 6, 2, 5, 9, 7]
```

We start by forming **subLists one element long**, e.g.

```
[[5], [3], [6], [2], [5], [9], [7]]
```

We then **merge successive pairs** of subLists, giving

```
[[3, 5], [2, 6], [5, 9], [7]]
```

We continue this merging until only 1 subList is left. Here, the next stage gives

```
[[2, 3, 5, 6], [5, 7, 9]]
```

& the last stage gives

```
[[2, 3, 5, 5, 6, 7, 9]]
```

mergesort, the top-level function, looks like

```

mergesort :: (a->a -> Bool)->[a] -> [a]
mergesort compfn [] = [] --check this once only
mergesort compfn dlis =
    (mergesortA compfn (map (\ e -> [e]) dlis))

```

mergesortA will do the work. We give it the initial sublists of length one, produced by the **map** over the original data.

mergesortA will return a list of length 1, like **[[1,2,3,4]]**. The **head** of this is what **mergesort** returns.

Here's **mergesortA**: by recursive calls it transforms sublists of length 1 to sublists of length 2, then 4 .. until there's only one sublist left.

```

mergesortA _ [lis] = lis -- one list only, it's the answer
-- general case - merge list pairs & repeat

```

```
mergesortA compfn mlis= mergesortA compfn
                        (mergesortpass compfn mlis)
```

The final function **mergesortpass** uses **my_merge** to perform one such step:

```
mergesortpass :: (a->a -> Bool)->[[a]] -> [[a]]
mergesortpass _ [] = []
mergesortpass _ [l]= [l] -- one element, return list
                        -- unchanged

-- general case - merge first two lists, cons to remainder

mergesortpass compfn (lis1:(lis2:rest)) =
    (merge compfn lis1 lis2): mergesortpass compfn rest
```

The general case merges the first two sublists using the **compfn** and makes this result the **head** of a list whose **tail** is the result of the remainder of the **mergesortpass**.

```
mergesort (\str1 str2 -> (length str1) < (length str2))
    ["qw","erty", "q","wer","qwert"]
~> ["q", "qw", "wer", "erty", "qwert"]
```

You can make this a bit neater using **@-notation** and **curried functions** (see later).

This implementation of mergesort *builds up* from sublists of length 1,2,4,8... Alternatively, you can define mergesort in a *break down* style.

17.3 Randomising with Mapping functions

It's often necessary to arrange a list of data into random order. For instance, in coding, a simple substitution cypher replaces each letter with another, randomly chosen.

Haskell provides random number generators in a library **System.Random** which we can import.

Then

- **mkStdGen** will return a random number generator
- You can give **mStdGen** a seed – an **Int**

- **randoms** is given a generator & returns a list of random numbers:

so

```
rlis = take 26 (randoms (mkStdGen 42)) :: [Int])
```

returns a list of 26 random **Ints**. We then

- zip this with the alphabet string

```
zlis = zip "abcdefghijklmnopqrstuvwxyz" rlis
```

producing a list of (**Char**,**Int**) pairs e.g.

```
zlis ~> [ ('a',15), ('b',7), ('c',21) ..]
```

- sort this list according to the Ints

```
slis = mergesort \(_,n1) (_,n2)->n1<n2) zlis
```

- use map to extract the letters, now in random order

```
cypher = map fst slis)
```

18. Libraries

Standard libraries are available which you can import into Haskell programmes (just put `import` followed by the library name in your code). You can find out about them from the GHCi help menu or Hoogle.

The libraries are arranged in a hierarchy, e.g library names starting with **Data.** deal with different datatypes, those starting with **Debug.** are debugging facilities.

18.1 Data.List

A particularly useful library is **Data.List**¹⁰. Here are a few Data.List goodies:

- The datatype **Ordering** is for 3 way comparisons. Its values are **LT**, **EQ** and **GT**
- **compare** does a comparison and returns an ordering, e.g **compare 2 3 ~> LT**
- **sort** and **sortBy** do sorting: **sortBy**'s first argument must return an ordering

```
sort "qwerty" ~> "eqrtwy"
```

```
sortBy
```

```
  (\x y -> compare (length x) (length y))
```

```
  [[1,2],[3],[4,5,6]]
```

```
  ~> [[3],[1,2],[4,5,6]]
```

```
sortBy
```

```
  (\x y -> compare (length y) (length x))
```

```
  [[1,2],[3],[4,5,6]]
```

```
  ~> [[4,5,6],[1,2],[3]]
```

- **maximumBy** finds the maximum according to an ordering fn which you supply:

```
maximumBy compare [5,3,2,7,4] ~> 7
```

- **flip f x y**, where **f** is a function and **x** & **y** are arguments to **f**, presents **f** with its arguments in reverse order, so

```
maximumBy (flip compare) [5,3,2,7,4] ~> 2
```

- **group** groups identical consecutive elements of a list into sublists:

```
group "Mississippi"
```

```
~> ["M","i","ss","i","ss","i","pp","i"]
```

18.2 Debugging with Debug.Trace

¹⁰Some functions originally in **Data.List** have been moved to **Prelude** in modern versions of Haskell

The function **trace** in the library `Debug.Trace` takes 2 arguments. It prints the first argument (which must be a `String`) & returns the second. **traceShow** is like that but converts the first arg to a `String` automatically. For instance, to show the working of **mergesort** we can put a trace on **mergesortA**:

```
mergesortA :: (Ord a, Show a) => (a->a -> Bool)->[[a]] -> [a]
mergesortA _ [lis] = lis -- one list only, it's the answer
mergesortA compfn mlis=
    traceShow mlis mergesortA compfn (mergesortpass compfn mlis)
```

Now

```
mergesort (<) "qwertyuiop"
["q","w","e","r","t","y","u","i","o","p"]
["qw","er","ty","iu","op"]
["eqrw","ituy","op"]
["eiqrtywy","op"]
"eiopqrtywy"
```

19. AS-Patterns

It's often the case that we want to use a pattern to decompose a list but we would also like to keep a handle on the whole of the list. Consider our **length** example:

```
my_length :: [a] -> Int
my_length lis
    | null lis = 0
    | otherwise = 1 + my_length t
    where (_,t) = lis
```

The reason we needed a **where** there was because we wanted bindings both to the whole list (**lis**) and its tail **t**.

We can achieve this using 'as-notation' as follows:

```
my_length :: [a] -> Int
my_length [] = 0
my_length lis@(_:t) = 1 + my_length t
```

Here @ means ‘bind the variable on the left of the @ to the value on the right’

Another example we could use this in is **merge**:

```
my_merge :: (a->a -> Bool)->[a]->[a] -> [a]
my_merge compfn [] lis2 = lis2
my_merge compfn lis1 [] = lis1
my_merge compfn (h1:t1) (h2:t2)
    | compfn h1 h2 = (h1:my_merge compfn t1 (h2:t2))
    | otherwise = (h2:my_merge compfn (h1:t1) t2)
```

In the last 2 lines we’ve got (h2:t2) and (h1:t1) , but we just want to refer to the original lists, so we can say

```
my_merge :: (a->a -> Bool)->[a]->[a] -> [a]
my_merge compfn [] lis2 = lis2
my_merge compfn lis1 [] = lis1
my_merge compfn lis1@(h1:t1) lis2@(h2:t2)
    | compfn h1 h2 = (h1:my_merge compfn t1 lis2)
    | otherwise = (h2:my_merge compfn lis1 t2)
```

20. List Comprehensions

A notation for list manipulation which is found only in functional programming. Related to mapping functions but even more concise. *One comprehension can encapsulate many filters and maps.*

A list comprehension **defines a new list in terms of the elements of a given list**, by **testing** and **transforming** them. The given list is called the **generator**¹¹.

e.g. suppose we have a list of Int `ex~>[2,3,4,7]` then

```
[2*n | n<-ex] ~> [4,6,8,14]
```

```
[n>3 | n<-ex] ~> [False,False,True,True]
```

`<-` is meant to resemble the mathematical symbol \in . On the left of `<-` is a pattern (just a symbol `n` here) which matches elements of the list produced by the generator. The `|` separates the generator from the test/transform of its elements. You can read the first example as ‘Take all $2*n$ where n comes from `ex` (or exists in `ex`)’

The generator can include one or more tests:

```
[2*n | n<-ex, n>3] ~> [8,14]
```

We can use comprehensions as an alternative to **filter**:

```
[ch | ch<-"ab123", isDigit ch]
```

Here’s an example with a non-trivial pattern:




```
[m+n | (m,n)<-[(2,3), (4,5), (6,7)]] ~> [5,9,13]
```

A list comprehension can form part of a function definition: the following checks whether all Ints in a list are greater than a given threshold value `t`:

```
allAbove :: [Int]->Int->Bool
```

```
allAbove nlis t = null [n | n<-nlis, n<t]
```

¹¹We will only consider the case of a single generator. Comprehensions can be more general (see Thompson p 344)


```
mlis=[ (Able,64) , (Baker,75) , (Charles,77) , (Dogbreath 46)  
        
[ans pattern|gen pattern<-generator,test,test..]  
        
[name| (name,mark)<- mlis,mark>70]  
        
[Baker, Charles]
```

21. Functions as Values

Mapping functions take other functions as their arguments. Now we look at the other kind of higher-order functions: **functions which return other functions**.

A function of type

a -> (b -> c)

takes an **a** and returns a function which takes a **b** and returns a **c**

Notice that *-> associates to the right*, hence we could dispense with the brackets there.

21.1 Partial Function Application or Currying

Suppose we define a multiply function

```
multiply :: Int->Int->Int  
multiply x y = x*y
```

and then call it with only one argument: a **partial application**

```
multiply 6
```

This is not an error. What it does is return a function which requires 1 more **Int** to return a value. We could give it a name:

```
times6=multiply 6  
times6 7 ~> 42
```

note the type of times6:

```
times6 :: Int->Int
```

Partial application is also called **currying**, after Haskell Curry. It's very useful, e.g. when combined with mapping functions: in a previous example we had

```
map ( \n -> n + 5) [1,2,3] ~> [6,7,8]
```

But we can equally say

```
map (+) 5 [1,2,3] ~> [6,7,8]
```

again,

```
filter (\n -> (n > 100)) [87, 112, 54, 225] ~> [112,225]
```

can be written

```
filter ((<) 100) [87, 112, 54, 225] ~> [112,225]
```

so by using partial application here we have avoided having to define those throwaway lambdas.

We can be even more succinct in this case:

```
map (5+) [1,2,3] ~> [6,7,8]
filter (>100) [87, 112, 54, 225]
```

because if *op* is an operator then (*xop*) is a fn requiring the second arg to op & (*opx*) is a fn requiring the first arg. This kind of partial application is called a **section**.

As another example, suppose we have

```
countElems :: Eq a => a ->([a]->Int)
countElems x lis = length (filter (\y->x==y) lis)
```

countElems returns a function which counts how many times a given element appears in a given list, so this is a fn which counts 6s:

```
countSixes = countElems 6
countSixes [3,2,6,5,6,7] ~> 2
```

.. and we don't need to give a name to the final fn:

```
(countElems 'q') "quertyqq" ~> 3
```

Curried functions take their arguments one at a time, in contrast to functions which take a single tuple which contains all the arguments.

21.2 Arity

How many arguments do Haskell functions really have (i.e. what is their *arity*)?

In fact **every Haskell function takes one argument**, the one before the initial -> in its type declaration.

```
multiply :: Int->Int->Int
```

is shorthand for

```
multiply :: Int->(Int->Int)
```

because the **function space symbol** -> is right-associative, and therefore the definition doesn't really mean

'multiply takes 2 Ints and returns an Int'

but rather

'multiply takes an Int and returns a function which takes an Int and returns an Int'

When we say

```
multiply 5 7
```

Haskell will convert this into

```
(multiply 5) 7
```

which is then evaluated to give **35**

The meaning of the function space symbol \rightarrow in type declarations is that it denotes a function which takes the type on its left and returns the type on its right.

21.3 The rule of cancellation

If the type of a function f is

$$t_1 \rightarrow t_2 \rightarrow \dots t_n \rightarrow t$$

and it is applied to arguments

$$e_1 :: t_1, e_2 :: t_2, \dots e_k :: t_k \text{ where } k \leq n$$

then the result type is given by cancelling the types t_1 to t_k

$$t_1 \rightarrow t_2 \rightarrow \dots t_k \rightarrow t_{k+1} \rightarrow \dots t_n \rightarrow t$$

21.4 Function-level definition

We can define functions in terms of other functions:

Conventional definitions like

```
double :: Int -> Int  
double x = 2*x
```

define the function behaviour *pointwise* i.e. it defines what to return for any given x . But we could have just said

```
double = (2*)
```

which uses partial application to define *double* *wholesale* in terms of another function.

Here's another example, involving 2 partial applications:

```
doubleAll = map (*2)  
doubleAll [1,2,3] ~> [2,4,6]
```

21.5 Function composition

The composition of 2 fns **f** and **g** is what happens when **g** is applied to an argument and then **f** is applied to the result. We can define a function to do composition thus:

```
fcomp :: (b->c) -> (a->b) ->a->c
```

```
fcomp f g x = f (g x)
```

But it is more common to use the infix composition operator .

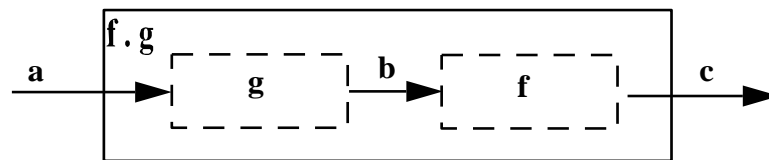
The composition of **f** and **g** is written as

```
f . g
```

and its effect is given by the definition

```
(f . g) x = f (g x)
```

i.e. **g** is applied to **x** and **f** is applied to the result



some examples:

1. Multiple successors

```
(succ . succ) 3 ~> 5  (succ is the successor function)
```

2. A compositional version of pyth:

```
(sqrt.sum.(map (**2))) [3,4] ~> 5
```

3. Composition in function definitions:

```
twice f = (f.f)
```

```
(twice succ) 12 ~> 14
```

note the type of **twice** here:

```
twice :: (a->a) -> (a->a)
```

4. The mode of a data set

The mode is the most common item, i.e. **mode [3,2,5,5,2,5,1,8] ~> 5**

```
import Data.List -- need compare
```

```
-- useful fn specifying what we want to compare by
comparing :: Ord b=>(a->b)->a->a->Ordering
comparing f l r = compare (f l) (f r)

mode :: Ord a => [a] -> a
mode = head.maximumBy (comparing length).group.sort
```

22 Lazy Programming

Haskell has a **lazy evaluation strategy**: arguments to a function are only evaluated when they are needed - *on demand*. The opposite strategy is called **eager evaluation**.

Lazy evaluation makes sense for a number of reasons: efficiency, avoiding calculations which don't make sense, allowing infinite data structures. It also effects programming style.

22.1 Efficiency

If you ask for

```
head [1^1,2^2,3^3,4^4,5^5]
```

only 1^1 will be evaluated. An eager strategy would work out $2^2, 3^3$ etc. even though these results are never used.

If you have a function with guards:

```
switch :: Int -> a -> a -> a
switch n x y
  | n > 0 = x
  | otherwise = y
```

then to evaluate

```
switch p q r
```

p will be evaluated and then either **q** or **r**

If a function has multiple equations triggered by different patterns the same principle applies: Haskell works its way through the equations in order, doing just enough evaluation to be able to decide whether the patterns fit.

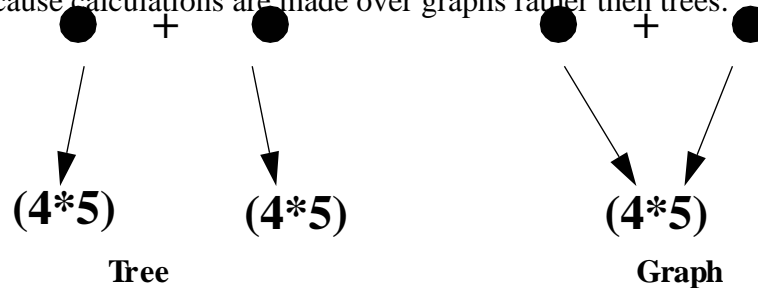
If the function has local bindings defined in a *where*, then these too will only be evaluated if they are needed.

Haskell also ensures that *duplicated arguments are only evaluated once* so with this definition of double

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

```
double (4*5) will only evaluate 4*5 once.
```

This is because calculations are made over graphs rather than trees:



22.2 Invalid evaluations

Lazy evaluation also helps you to avoid doing computations which don't make sense, e.g. dividing by zero.

```
safeDivide :: Float->Float->Maybe Float
safeDivide x y
  | y==0 = Nothing
  | otherwise = Just x/y
```

Here's our definition of **length**

```
length :: [a] -> Int
length [] = 0
length (_:rest) = 1 + length rest
```

If we give **length** an empty list then the first equation will trigger & we won't try to compute **rest**, which would produce an error.

22.3 Infinite Computation

It is easy to define endless computations in Haskell:

```
loop :: Int->Int
loop n = loop (n+1)
```

but because Haskell is lazy we can have infinite computations in expressions provided they don't get evaluated:

```
head [3, loop 4]
```

22.4 Infinite Lists

Given **n**, the following constructs $[n^2, (n+1)^2, (n+2)^2, \dots]$

```
squares :: Int -> [Int]
squares n = n^2:(squares (n+1))
```


but lazy evaluation allows us to take a finite part of an infinite computation:

```
take12 5 (squares 4) ~> [16,25,36,49,64]
```

It turns out to be useful to be able to work with infinite lists. Haskell extends the `[..]` notation to allow us to define them easily:

```
[1..7] ~> [1,2,3,4,5,6,7]
[1..] ~> [1,2,3,...] -- & will fill your window if you
                    -- evaluate it
[1,4..16]~>[1,4,7,10,13,16]
[1,4..] ~> [1,4,7,10,...]
```

The following prelude fn allows us to construct infinite lists:

```
iterate (a->a)->a->[a]
iterate f x = x:iterate f (f x)

iterate (2*) 3 ~> [3,6,12,24,...]
```

Here's a fn using **map** over an infinite list to construct a graph of $f(x)$ against x , for $x=0,1,\dots$

```
mkGraph :: (Int -> a) -> [(Int,a)]
mkGraph f = map(\n->(n, f n)) [0..]

take 5 (mkGraph (3^))~>
[(0,1), (1,3), (2,9), (3,27), (4,81)] -- f(x)=3x

take 5 (mkGraph (^3))
[(0,0), (1,1), (2,8), (3,27), (4,64)] -- f(x)=x3
```

Infinite lists can be used in comprehensions:

```
[(x,y,z) | z<-[2..], y<-[2..z-1], x<-[2..y-1], x*x+y*y==z*z]
```

generates the pythagorean triples

```
[(3,4,5), (6,8,10), (5,12,13), (9,12,15), (8,15,17) ...]
```

here's a version of **mkGraph** using comprehensions:

```
mkGraph f = [(x, f x) | x <- [0..]]
```

¹²**take n lis** returns the first **n** elements of **lis**

23. Classes and Types

We've seen how to use type classes like Eq:

```
member :: Eq a => a -> [a] -> Bool
```

Here is how Eq is defined:

```
class Eq a where
    (==), (/=) :: a -> a -> Bool

    -- Minimal complete definition:
    --      (==) or (/=)
    x /= y      = not (x == y)
    x == y      = not (x /= y)
```

i.e. if a type `a` is to be made an instance of the type class `Eq` it must support the functions `(==)` and `(/=)` (the **class methods**).

The class also defines default definitions for `(==)` and `(/=)` *in terms of each other*.. so if you define one, you've got the other.

We can make our own instances of type classes:

```
data Foo = Foo {x :: Integer, str :: String}

instance Eq Foo where
    (Foo x1 str1) == (Foo x2 str2) = (x1 == x2) &&
                                      (str1 == str2)
```

now

```
Foo 3 "orange" == Foo 6 "apple" ~> False
Foo 3 "orange" /= Foo 6 "apple" ~> True
```

Note

- **deriving** prevents us from having to write the obvious instance declarations for prelude classes such as `Eq`, `Ord` and `Show`
- Classes can inherit from other classes. Here's the start of the definition for `Ord`, which inherits from `Eq`:

```
class (Eq a) => Ord a where
    compare          :: a -> a -> Ordering
    (<), (<=), (>=), (>) :: a -> a -> Bool
    max, min         :: a -> a -> a
```

Functors

Mapping a function over a data structure is a powerful concept and can be generalised to structures other than lists. For instance, recall our definition of a binary tree:

```
Data Tree a :: Empty | Leaf a | Node (Tree a) (Tree a)
```

We can define **treeMap**, which traverses the tree applying a function to each data item & returning a tree of the results:

```
treeMap :: (a->b)->(Tree a)->(Tree b)

treeMap _ Empty = Empty

treeMap fn (Leaf x) = Leaf (fn x)

treeMap fn (Node x lst rst) = Node (fn x)
                                (treeMap fn lst)
                                (treeMap fn rst)

traverse (treeMap length animals) ~> [8,3,7,3,8,3,4,4,6]
```

Haskell provides a class **Functor** which can be used to define a mapping over a given structure. Instances of **Functor** must define the mapping function **fmap**. To implement **treeMap** this way we say

```
instance Functor Tree where

    fmap f Empty = Empty

    fmap f (Leaf x) = Leaf (f x)

    fmap f (Node x left right) = Node (f x) (fmap f left)
                                     (fmap f right)
```

24. Input/Output

24.1 I/O and Functional Programming

Everything we've done so far has been *self-contained*: the **.hs** files contain all the data we need, we call fns to perform evaluations using this data & they return results to the terminal.

This isn't sufficient when we require significant interaction with the 'world outside' - reading from and writing to files, interface to graphics, operating under a GUI etc.

We have to accept the fact that interactions often happen in sequence, though there's no direct way in FP to specify *'do x then y then z'*.

The problem is how to achieve better interaction without breaking the functional programming rules: e.g.

- reading in the value of a variable is a change of state.
- If you have a programme which works its way through a file reading line-by-line as in java:

```
str = rdr.readString()
```

then every time you do this you're destructively overwriting the value of **str**

- if you allow operations like this inside a function body then you can never be sure that the function will always return the same value given the same arguments.

The Haskell way of addressing this issue is to make a clean separation between the functions which handle i/o: which allow **actions** and the functions which are 'pure': which are **definitions**.

24.2 Basic IO in Haskell

Haskell provides the type

```
IO a
```

an object of this type is a programme which will perform some IO and then return a value of type **a**. The IO might be a single action or a number of actions.

Haskell also provides a language for IO actions, including a way of sequencing these actions. Here are some simple IO commands for terminal-based interaction:

```
getLine :: IO String -- reads a line, returns it as a string  
putStr :: String -> IO () -- prints a string. Return type is  
'void'
```

We can use these in functions with return type **IO a**:

```
sayHello :: IO ()
sayHello=putStrLn "Hello world"
```

now

```
> sayHello
Hello world
```

IO actions often need to be done in sequence, so there is the **do** construct:

```
greeting :: IO ()
greeting = do
  putStrLn "Greetings!  What is your name?"
  inpStr <- getLine
  putStrLn ("Welcome to Haskell, " ++ inpStr ++ "!")
```

greeting behaves like so:

```
Haskell_IO> greeting
Greetings!  What is your name?
Phil
Welcome to Haskell, Phil!
```

but there is a clean break between ‘pure’ functions and functions which do IO:

- You can only use **do** inside a fn which has return type **IO a**
- You can only use existing fns with return type **IO a** (like **getLine**) within a function with type **IO a**
- Within a **do**, **<-** binds a variable to the result of an IO action

24.3 The interface between IO actions and ‘Pure’ code

You can call ‘pure’ functions from within a **do**, but you use **let** to bind the values which they return: this example uses **take** :

```
takesome :: IO ()
takesome = do
  putStrLn "  give me a string please"
  listr<-getLine
  putStrLn "  and an index"
  istr<-getLine
  let resstr = (take (read istr ::Int) listr)
```

```
putStrLn $ "    the first "++istr++
           " chars in your string are "++resstr
```

this behaves like so:

```
> takesome
    give me a string please
quert
    and an index
3
    the first 3 chars in your string are que
```

24.4 Other IO actions

We've only looked at terminal IO (`getLine`, `putStrLn`) but the same idea is extended in obvious ways to IO via files, e.g.

```
inpStr <- hGetLine inh
```

reads a string from a file you've opened. `inh` is a 'handle' to that file, which you get when you open it:

```
inh <- openFile "input.txt" ReadMode
```

Haskell extends its 'lazy evaluation' philosophy to IO, e.g if you ask to read in a whole file Haskell only reads data from it as and when its needed.

24.5 Monads

do makes explicit the order in which things will take place, whereas in pure functional programming order doesn't matter: if we call **f** with args **a** and **b** it doesn't matter whether we evaluate **a** or **b** first, the result will be the same.

Monads are a means of incorporating sequential computations which may have side effects into a pure functional framework. We have been using the IO Monad, but there are more, and you can define your own. Monads support a whole range of things like exceptions, state, non-determinism, continuations, coroutines,

Here's the definition of the Monad type class:

```
class Monad m where
    return :: a -> m a
    (>=)    :: m a -> (a -> m b) -> m b

    (>>)    :: m a -> m b -> m b
    fail    :: String -> m a
```

