Introduction to Programming, Aug-Dec 2006

Lecture 3, Friday 11 Aug 2006

Lists . . .

We can implicitly decompose a list into its head and tail by providing a pattern with two variables to denote the two components of a list, as follows:

```
length :: [Int] -> Int

length [] = 0
length (x:xs)  = 1 + (length xs)
```

Here, in the second definition, the input list `l` is implicitly decomposed so that `x` gets the value `head l` while `xs` gets the value `tail l`. The bracket around `(x:xs)` is needed; otherwise, Haskell will try to compute `(length x)` before dealing with the `:`. In this example, the list is broken up into a single value `x` and a list of values `xs`. This is to be read as "the list consists of an x followed by many x's" and is a useful convention for naming lists.

Notice that in the second inductive definition of length, `x` plays no role in the right hand side of the second definition, so we could also write:

```
length :: [Int] -> Int

length [] = 0
length (_:xs)  = 1 + (length xs)
```

We can rewrite `sum` using list pattern matching as follows.

```
sum :: [Int] -> Int
sum [] = 0
sum (x:xs) = x + (sum xs)
```

Here are some more examples of functions over lists: The function `concatenate` combines two lists into a single larger list.

```
concatenate :: [Int] -> [Int] -> [Int]
concatenate [] ys = ys
concatenate (x:xs) ys = x:(concatenate xs ys)
```

Concatenation is so useful that Haskell has a builtin binary operator `++` for this. Thus `[1,2,3] ++ [4,3]` $\leadsto$ `[1,2,3,4,3]`, etc.

We can reverse a list by first reversing the tail of the list and then appending the head of the list at the end, as follows.

```
reverse :: [Int] -> [Int]
reverse [] = []
reverse (x:xs) = (reverse xs)++[x]
```

The functions `sum`, `length` and `reverse` are actually basic list functions in Haskell, like the functions `head` and `tail`. Some other useful builtin functions are:

- `init l` returns all but the last element of `l`

  `init [1,2,3]` $\leadsto$ `[1,2]`

  `init [2]` $\leadsto$ `[]`

- last l returns the last element in l

  `last [1,2,3]` $\leadsto$ `3`

  `last [2]` $\leadsto$ `2`

An important builtin function is `concat`. This function takes a list of lists and "dissolves" one level of brackets, merging its contents into a single long list. For instance:

`concat [[1,2,3],[],[4,5],[],[6]]` $\leadsto$ `[1,2,3,4,5,6]`

We can write an inductive definition for `concat` in terms of `++`:

```
concat [[Int]] -> [Int]
concat [] = []
concat (l:ls) = l ++ (concat ls)
```

Lists are sequences of values, so the position of a value is important. In the list `[1,2,1]`, there are two copies of the value 1, at the first and third position. Haskell follows the convention that positions are numbered from 0, so the set of positions in a list of length `n` is {`0,1,...,(length n) - 1`}.

The notation `xs!!i` directly returns the element at position `i` in the list `xs`. Note that accessing the element at position `i` in a list takes time proportional to `i`.

Two more useful built-in functions in Haskell are `take` and `drop`. The expression `take n l` will return the first `n` values in `l` while `drop n l` will return the list obtained by omitting the first `n` values in `l`. For any list `l` and any integer `n` we are guaranteed that:

```
l == (take n l) ++ (drop n l)
```

In particular, this means that if `n < 0`, `take n l` is `[]` and `drop n l` is `l`, while if `n > (length l)`, `take n l` is `l` and `drop n l` is `[]`.

Polymorphism

Observe that functions such as `length` and `reverse` work in the same way for lists of any type. It would be wasteful to have to write a separate version of such functions for each different type of list. In Haskell, it is possible to say that a function works for multiple types by using type variables. For instance, we can write:

```
length :: [a] -> Int
length [] = 0
length (x:xs)  = 1 + (length xs)
```

Here, the letter `a` in the type `[a] -> Int` is a type variable. The type `[a] -> Int` is to be read as ""or any underlying type `a`, this function is of type `[a] -> Int`". It is conventional to use letters `a`, `b`, ...to denote types. Note that the variables used in type expressions are disjoint from those used in actual function definitions so one could, in principle, use the same variable in both parts, though it is probably not a good idea from the perspective of readability.

In the same way, we can generalize the types of reverse and concat to read:

```
reverse :: [a] -> [a]

concat  :: [[a]] -> [a]
```

Here it is significant that the same letter `a` appears on both sides of the `->`. This means, for instance, that the type of the list returned by `reverse` is the same as the type of the input list. In other words, all occurrences of a type variable `a` in a type declaration must be instantiated to the same actual type. (Notice that this type of implicit pattern matching was precisely what we disallowed when writing Haskell function definitions such as `isequal x x = True`, so the way type variables are instantiated differs from the way variables in function definitions are instantiated.)

Functions that work in the same way on different types are called *polymorphic*, which means (in Greek) "taking different forms".

We must be careful to distinguish polymorphism of the type we have seen with lists from the ad hoc variety associated with overloading operators. For instance, in most programming languages, we write `+` to denote addition for both integers and floating point numbers. However, since the underlying representations used for the two kinds of numbers are completely different, we are actually using the same name (`+`, in this case) to designate functions that are computed in a different manner for different base types. This type of situation is more properly referred to as *overloading*.

In a nutshell, overloading uses the same symbol to denote similar operations on different types, but the way the operation is evaluated for each type is different. On the other hand, polymorphism refers to a single function definition with a fixed computation rule that works for multiple types in the same way.

Note that we cannot assign a simple polymorphic type for `sum`. It would be wrong to write

```
sum :: [a] -> a
```

because `sum` will work only for lists whose underlying type supports addition. We will see later that it is possible to write a conditional type expression such as `sum :: [a] -> a` provided the type `a` supports the operation `+`.

Haskell allows us to pass any type to a function, including another function. Consider the function `apply`, that takes as input a function `f` and a value `x` and returns the value `(f x)`. In other words, this functions *applies* `f` to `x`. The definition of `apply` is very straightforward:

```
apply f x = f x
```

What is the type of `apply`? The first argument is any function, so we can denote its type as `a -> b` for some arbitrary types `a` and `b`. The second argument `x` has to be fed as an input to `f`, so its type must be `a`. The output of `apply` is `f x`, which has type `b`. Thus, we have,

```
apply :: (a -> b) -> a -> b
```

Notice that we must put brackets around `(a->b)` to ensure that this is not seen as a function of three variables.

What if we change the function to apply `f` twice to `x`?

```
twice f x = f (f x)
```

In this case, we see that the output `(f x)` is fed back as an input to `f`. This means that the input and output types `a` and `b` must be the same, so `f :: a -> a` and the type of `twice` is given by

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

The analysis we did by hand when trying to deduce the type of `apply` and `twice` is built in to Haskell. Thus, if we do not provide an explicit type for a function, Haskell will start with the most general assumption about the type and impose the constraints inferred from the function definitions to arrive at a final type.

As a last remark, recall that in our discussion of functions with multiple inputs we said that each input transforms the original function into a new one. Thus, when we write `plus m n = m+n`, after the input `m` is provided, we get a new function `(plus m)` which is of type `Int -> Int`. This intermediate function actually does exist in "real" life. For instance, if we write `apply (plus 7) 8`, we get the answer `15`.

The datatype `Char`

In Haskell, character constants are written within single quotes, like `'a'`, `'3'`, `'%'`, `'#'`, . . . Like all other datatypes, characters are encoded in a table.

Two functions are provided that allow us to interpret characters in terms of their internal position in the table and vice versa.

```
ord :: Char -> Int
chr :: Int -> Char
```

These are inverses of each other, so for each `Char c`, `c == chr(ord c)`, and for each `Int i` that is a valid encoding of a `Char`, `i == ord (chr i)`.

The actual way in which characters are organized in a table may vary from one system to another. In practice, most current systems use the encoding called ASCII in which characters are represented by positive binary numbers from 0 to 255. However, some languages use a larger range of encodings to accommodate characters from alphabets of different languages.

We shall assume only the following facts about the encoding of characters in Haskell.

- The characters `'a'`, `'b'`, . . . , `'z'` occur consecutively.
- The characters `'A'`, `'B'`, . . . , `'Z'` occur consecutively.
- The characters `'0'`, `'1'`, . . . , `'9'` occur consecutively.

This means that `ord 'b'` is always `(ord 'a') + 1`, and `(ord 'A' - ord 'a') == (ord 'B' - ord 'b')` etc.

Let us define a function `capitalize` that maps `'a'`, `'b'`, . . . , `'z'` to `'A'`, `'B'`, . . . , `'Z'` and leaves all other characters unchanged.

Here is a brute force definition of `capitalize` that makes use of the fact that we have 26 values to transform.

```
capitalize :: Char -> Char
capitalize 'a' = 'A'
capitalize 'b' = 'B'
.
.
.
capitalize 'y' = 'Y'
capitalize 'z' = 'Z'
capitalize c = c
```

The first 26 lines do the capitalization. The last line preserves all values other than `'a'`, `'b'`, . . . , `'z'`.

Here is a slightly smarter version of the same function that uses the fact that the displacement between a lower case letter and its capitalized version is a constant.

```
capitalize :: Char -> Char
capitalize c
  | ('a' <= c && c <= 'z') = chr (ord c + (ord 'A' - ord 'a'))
  | otherwise              = c
```

Notice that we are allowed compare the order of characters—this is used to check if `c` lies between `'a'` and `'z'`. Comparison is based on the `ord` value of a character : character `x` is less than character `y` if `ord x < ord y`.

However, we cannot perform arithmetic on characters. Expressions such as `'a' + 2` are illegal. Thus, while it is true that `'c' == chr (ord 'a' + 2)`, it is nonsensical to claim that `'c' = 'a' + 2`.

Strings

Programs that manipulate text deal with sequences of characters, not single characters. A sequence of characters is normally called a *string*. In Haskell, a sequence of characters is just a list of `Char`, or a value of type `[Char]`. The word `String` is a synonym for `[Char]`. Also, instead of writing strings using somewhat tedious list notation such as `['h','e','l','l','o']` we are allowed to directly write the string in double quotes, `"hello"` in this case.

Manipulating a `String` is easy—a `String` is just a list of `Char` so all list function work on `String` just like any other list. Thus, `length` can be used to get the length of a `String`, `concat` can be used to collapse a list of `String`s into a single long `String` etc.

```
length "hello" ⤳ 5

concat ["hello"," ","world"] ⤳ "hello world"
```

It is important to remember that single quotes denote character constants while double quotes denote strings. For instance, `'a'` is the character `a` while `"a"` is the list `['a']`.

Here is an example of a function on `String`. This function uses the function `capitalize` we wrote earlier to convert each letter in a `String` to uppercase.

```
touppercase :: String -> String

touppercase "" = ""
touppercase (c:cs) = (capitalize c):(touppercase cs)
```

Notice that in the inductive definition we use `""` to denote the empty string. This is translated to `[]`, the empty list of `Char`.

Here is another example of a function on `String`s—`exists` checks whether a given character occurs in a given string.

```
exists :: Char -> String -> Bool

exists c "" = False
exists c (x:xs)
  | c == x    = True
  | otherwise = exists c xs
```

Thus, as we march along the `String`, if we find a copy of the `Char` we are looking for, we report `True` and stop. Otherwise, we continue looking along the rest of the `String`. If we don't find a copy of what we are looking for, we eventually reach the empty string and return `False`.

Exercise Write a function `position :: Char -> String -> Int` such that `position c s` returns the first position in `s` where `c` occurs and returns `-1` if `c` does not occur in `s`. Note that a valid answer for this function must be either `-1` or a number in the range {`0,1,...,length s - 1`}.