Haskell Unit 2: Lists

Antoni Diller

26 July 2011

Introduction

The most important datatype in a functional language is the *list*. A list is a linearly ordered collection of elements. All elements of a list must be of the same type. Some examples:

```
        [3, 7, 5, 88] :: [Int]
['t', 'i', 'm', 'e'] :: [Char]
               "time" :: [Char]
 [[2, 3], [4, 8, 17]] :: [[Int]]
```

Haskell provides many list operators. Some are:

`:`	binary infix	sticks an element at the front of a list
`head`	unary prefix	extracts the first element of a non-empty list
`tail`	unary prefix	returns the tail of a non-empty list
`length`	unary prefix	returns the length of a list
`!!`	binary infix	extracts an element of a list

A function to sum the elements of a list of integers can be defined like this:

```
sum :: Integral a => [a] -> [a]
sum ys
  | ys == []  = 0
  | otherwise = head ys + sum (tail ys)
```

It is better, however, to use pattern-matching thus:

```
sum :: Integral a => [a] -> [a]
sum []     = 0
sum (y:ys) = y + sum ys
```

List addition and subtraction

Two useful binary infix functions on lists are ++ (list addition) and \\ (list subtraction). List addition takes two lists as its arguments and sticks them together. List subtraction removes elements from a list, for example:

`[1, 2, 3, 4, 5] \\ [1, 4]`	is equivalent to	`[2, 3, 5]`
`[1, 1, 1, 1] \\ [1, 4]`	is equivalent to	`[1, 1, 1]`
`[1, 1, 1, 1] \\ [1, 1]`	is equivalent to	`[1, 1]`

List subtraction is not predefined in the version of Haskell used here, but it can be defined like this:

```
(\\) :: Eq a => [a] -> [a] -> [a]
[] \\ _  = []
xs \\ [] = xs
(x:xs) \\ (y:ys)
  | x == y    = xs \\ ys
  | otherwise = (x : (xs \\ [y])) \\ ys
```

Local definition

Haskell supports local definitions, for example:

```
foo x
  | x > 0  = p + q
  | x <= 0 = p - q
             where
             p = x^2 + 1
             q = 3*x^3 - 5
```

Local defintions obey Landin's offside rule:

> The southeast quadrant that just contains the phrase's first symbol must contain the entire phrase, except possibly for bracketted subexpressions.

Programming style

The following two definitions of a leap year illustrate bad and good programming style:

```
leap1 y = (y ‘mod‘ 4 == 0) &&
          (y ‘mod‘ 100 /= 0 ||
           y ‘mod‘ 400 == 0)

leap2 y
  | y ‘mod‘ 100 == 0 = y ‘mod‘ 400 == 0
  | otherwise        = y ‘mod‘ 4 == 0
```

In Haskell `leap2` is considered more elegant than `leap1`.