HW 7 **Prof. Caldwell**
Due: 27 September 2011 **COSC 3015**

Problem 0.1. Read Chapters 4, 5 and 6 of LYAHFGG

1 Highlights of the 9/25/12 Lecture

1.1 Prefix operators, Infix operators and Sections

If f is a function of type $a \to b \to c$ we reminded the class that it can be used in an infix position by enclosing it in back-quotes (‘). Thus, $f\ x\ y \quad = \quad x\ ‘f‘\ y$.

If $\otimes$ is an infix binary operator symbol, then enclosing that symbol in parenthesis transforms it into a Curried operator. You can also add an argument to the left or right side to create a unary function. Thus, if $(\otimes) :: a \to b \to c$, $x :: a$ and $y :: b$ then $(x\otimes)$ has type $b \to c$ and $(\otimes y)$ has type $a \to c$.

For example, consider the the infix cons operator $(:) :: a \to [a] \to [a]$. Then $(42\ :)$ has type $(Num\, a) \Rightarrow [a] \to [a]$ and $(:\ [2,3])$ has type $(Num\, a) \Rightarrow a \to [a]$. Recall that $flip$ has type $(a \to b \to c) \to b \to a \to c$, thus $(flip\,(:)) :: [a] \to a \to [a]$ and $(”bc”\ (flip\,(:)))$ has type $Char \to [Char]$ and $(flip\,(:)\ ‘z‘)$ has type $[Char] \to [Char]$.

1.2 Patterns of recursion and *foldr*

In class we presented the following two functions defined by recursion on the structure of their list arguments:

```
sum' [] = 0
sum' (x:xs) = x + sum' xs

prod' [] = 1
prod' (x:xs) = x * prod' xs
```

We noted that these functions look pretty much the same – they only differ in their names, the identity element used (call it *id*) and the operator (call it *op*) – then they have the following shared form:

```
name [] = id
name (x:xs) = x ‘op‘ name xs
```

By abstracting *id* and *op* we implemented a function (called *foldr* that captures this pattern of recursion as follows:

```
foldr :: (a → b → b) → b → [a] → b
foldr op id [] = id
foldr op id (x:xs) = x ‘op‘ foldr op id xs
```

We can then reimplement the *sum'* and *prod'* functions as follows:

```
sum' = foldr (+) 0
prod' = foldr (*) 1
```

Not that the following identity holds:

Theorem 1.1 () $\forall xs : [a].\ foldr(:)[] \ \ = \ \ (\backslash x \rightarrow x)$

We will provide the means to prove theorems like this one which require a form of induction next week.

1.3 *map* and *filter*

We defined the *map* function as follows:

```
map :: (a → b) → [a] → [b]
map f [] = []
map f (x:xs) = f x : map f xs
```

The *map* function is higher order (its first argument is a function of type $a \rightarrow b$). If the list is empty it returns the empty list and if not it decomposes the list and builds a new one whose head is obtained by applying the function f to the head of the input list and consing that onto the result of recursively calling *map f* on the tail of the list.

The function *filter* takes a predicate (a function of type $a \rightarrow Bool$ and a list (say xs) of type $[a]$ and returns a new list containing only those elements of xs which satisfy the predicate p *i.e.* it keeps the elements x in xs for which $p\ x == True$.

```
filter':: (a → Bool) → [a] → [a]
filter' p [] = []
filter' p (x:xs) = if p x then (x : filter' p xs) else filter' p xs
```

2 Problems

Problem 2.1. Write a recursive function that behaves something like *filter'* but which returns two lists:

```
partition :: (a → Bool) → [a] → ([a],[a])
```

If *partition p xs == (ys,zs)* then the first list ys should contain those elements of xs for which p is *False* and the zs should contain those elements of xs for which p is *True*. warning: if you copy code form the web – be careful.

Problem 2.2. Rewrite *map'*, *filter'* and *partition'* using *foldr* (and thereby not directly using recursion.) Hint: This is tricky but a careful reading of Chapter 6 of LYAHFGG should make it doable.