Simon Fraser University
School of Computing Science

CMPT 383

Assignment 4 (Functional Programming)

Due date: December 1, 2005

1. **(6 marks)** Imagine a language of expressions for representing integers defined by the syntax rules: (a) zero is an expression, (b) if `e` is an expression, then so are `succ(e)` and `pred(e)`.
   An evaluator reduces expressions in this language by applying the following rules repeatedly until no longer possible:

   ```
   succ(pred(e)) = e
   pred(succ(e)) = e
   ```

   Given the expression `succ(pred(succ(pred(pred(zero)))))`, write a **reduction sequence**. In how many ways can the reductions be applied to this expression? Do they all lead to the same final result?

   Outermost Reduction Sequence

   ```
   succ(pred(succ(pred(pred(zero))))
   { by succ(predd(e)) = e }
   succ(pred(pred(zero)))
   { by succ(pred(e)) = e }
   pred(zero)
   ```

   Innermost Reduction Sequence

   ```
   succ(pred(succ(pred(pred(zero))))
   { by succ(predd(e)) = e }
   succ(pred(pred(zero)))
   { by succ(pred(e)) = e }
   pred(zero)
   ```

   Other Reduction Sequence

   ```
   succ(pred(succ(pred(pred(zero))))
   { by pred(succ(e)) = e }
   succ(pred(pred(zero)))
   { by suc(pred(e)) = e }
   pred(zero)
   ```

   There are 3 different reduction sequences for this expression. All of them lead to the same final canonical form (result).

2. **(6 marks)** Suppose a date is represented by a triple `(d,m,y)` of three integers, where `d` is the day, `m` is the month, and `y` is the year. Define a function **`age`** that takes two dates, the first being the current date, and the second being the birthdate of some person `P`, and return the age of `P` as a whole number of years.

   ```
   age :: (Integer,Integer,Integer) -> (Integer,Integer,Integer) -> Integer
   age (d1,m1,y1) (d2,m2,y2)
          | years >= 0 && m2 < m1 = years
          | years >= 0 && m2 == m1 && d2 <= d1 = years
          | years > 0 && m2 == m1 && d2 > d1 = years - 1
          | years > 0 && m2 > m1 = years - 1
          | otherwise = error "Incorrect dates"

          where years = y1 - y2
   ```

3. **(6 marks)** Define a function **`convert`**`::Nat→Integer` that converts a natural number to an integer.

```
data Nat = Zero | Succ Nat

convert :: Nat -> Integer
convert Zero = 0
convert (Succ n) = 1 + convert n
```

4. **(6 marks)** Define the function that **`split`**s a list of numbers into two lists: positive ones (including zero) and negative ones. For example

```
? split [3,-1,0,5,-2]
([3,0,5],[-1,-2])
```

```
split :: [Integer] -> ( [Integer], [Integer] )
split [] = ( [], [] )
split (x:xs)
      | x >=0 = (x:fst(split xs),snd(split xs))
      | x < 0 = (fst(split xs),x:snd(split xs))
```

5. **(5 marks)** The function `filter` takes a Boolean function `p` and a list `xs` and return that sublist of `xs` whose elements satisfy `p`. For example,

```
? filter even [1,2,4,5,32]
[2,4,32]
```

This function `filter` can be defined in terms of `concat` and `map`:

```
filter p xs = concat · map box
              where box = …
```

Give the definition of **`box`**.

```
filter :: ( a -> Bool ) -> [a] -> [a]
filter p xs = concat ( map box xs )
      where box x = if p x then [x] else []
```

6. **(5 marks)** The functions `takeWhile` and `dropWhile` are similar to `take` and `drop` except that they both take a boolean function as first argument instead of a natural number. The value `takeWhile p xs` is the longest initial segment of `xs` all of whose elements satisfy `p`. For example:

```
? takeWhile even [2,4,6,1,5,6]
[2,4,6]
```

The value `dropWhile p xs` gives what remains; for example:

```
? dropWhile even [2,4,6,1,5,6]
[1,5,6]
```

Give recursive definitions of **`takeWhile`** and **`dropWhile`**.

```
takeWhile :: ( a -> Bool ) -> [a] -> [a]
takeWhile p [] = []
takeWhile p (x:xs) = if p x then x:takeWhile p xs else []

dropWhile :: ( a -> Bool ) -> [a] -> [a]
dropWhile p [] = []
dropWhile p (x:xs) = if p x then dropWhile p xs else x:xs
```

7. **(5 marks)** Define the function **`palindrome`** that verifies if a string is palindrome. A string is a palindrome if it reads the same in the forward and in the backward direction. For example:

```
? palindrome “madam”
True
```

```
palindrome :: String -> Bool
```

```
palindrome x = if x == reverse x then True else False
```

8. **(10 marks)** Write a program to **`convert`** a whole number of pence into words. For example, the number 3649 should convert to "thirty-six pounds and forty-nine pence".

```
? convert 3649
Thirty-six pounds and forty-nine pence
```

```
units, teens, tens :: [String]
units = ["one", "two", "three", "four", "five", "six", "seven", "eight", "nine"]
teens = ["ten", "eleven", "twelve", "thirteen", "fourteen", "fifteen", "sixteen",
"seventeen", "eighteen", "nineteen"]
tens = ["twenty", "thirty", "forty", "fifty", "sixty", "seventy", "eighty", "ninety"]

digits2 :: Int -> (Int,Int)
digits2 n = (div n 10, mod n 10)

combine2 :: (Int,Int) -> String
combine2 (0,0)     = "zero"
combine2 (0,u+1)   = units!!u
combine2 (1,u)     = teens!!u
combine2 (t+2,0)   = tens!!t
combine2 (t+2,u+1) = (tens!!t) ++ "-" ++ (units!!u)

convert2     :: Int -> String
convert2 n   = combine2 (digits2 n)

digits3 :: Int -> (Int,Int)
digits3 n = (div n 100, mod n 100)

combine3 :: (Int,Int) -> String
combine3 (0,0)     = "zero"
combine3 (0,t+1)   = convert2 (t+1)
combine3 (h+1,0)   = units!!h ++ " hundred"
combine3 (h+1,t+1) = units!!h ++ " hundred and " ++ convert2 (t+1)

convert3     :: Int -> String
convert3 n   = combine3 (digits3 n)

link :: Int -> String
link n = if n < 100 then " and " else " "

digits6 :: Int -> (Int,Int)
digits6 n = (div n 1000, mod n 1000)

combine6 :: (Int,Int) -> String
combine6 (0,0)     = "zero"
combine6 (0,h+1)   = convert3 (h+1)
combine6 (m+1,0)   = convert3 (m+1) ++ " thousand"
combine6 (m+1,h+1) = convert3 (m+1) ++ " thousand" ++ link (h+1) ++ convert3 (h+1)

convert6     :: Int -> String
convert6 n   = combine6 (digits6 n)

convert :: Int -> String
convert n = convert6 pounds ++ " pounds " ++ convert6 pence ++ " pence"
          where pounds = div n 100
                pence = mod n 100
```

9. **(6 marks)** An integer `x` can be represented by a pair of integers `(y,z)` with `x=10*y+z`. For example, `27` can be represented by `(2,7)`, `(3,-3)`, and `(1,17)`, among others. Among possible representations we can choose one in which `abs z<5` and `abs y` is as small as possible (subject to `abs z`≤`5`). Define a function **`reprint`**, so that `reprint x` returns this canonical representation.

```
      ? reprint 27
      (3,-3)

reprint :: Integer -> (Integer, Integer)
reprint x
      | abs (a) > 5 = (b+1, a-10)
      | abs (a) <= 5 = (b,a)
      where  a = mod x 10
             b = div x 10
```

10. **(15 marks)** Suppose that there are tab stops at every 8 spaces. Write a function that will take a string as an argument and return as a result a string that is equivalent to the input string, except that whenever two or more spaces can be replaced by a tab, this is done. You may assume that the input string contains no tabs, newlines, or other whitespace characters other than spaces.
    For example, "1234␣␣␣␣␣01␣␣4␣␣7890123␣␣6" should be transformed into "1234\t␣01␣␣4\t7890123␣␣6" where "\t" denotes a tab and "␣" represents a space. (If we show the position of tab stops by "^", then the input is "1234␣␣␣␣^␣␣01␣␣4␣␣^7890123␣^␣6".)
    The first tab replaces 4 spaces while the other tab replaces 2 spaces. A third tab could replace the single space in the 24th position of the input, except that we only use a tab to replace 2 or more spaces. The 2 spaces between 1 and the 4 cannot be replaced with a tab, because the 4 is not positioned after a tab stop. The other 2 spaces, near the end of the input, remain because only 1 can be replaced by a tab.

```
makeGroups8 :: String -> [String]
makeGroups8 [] = []
makeGroups8 s = take 8 s:makeGroups8 (drop 8 s)

returnSpaces :: String -> String
returnSpaces [] = []
returnSpaces (x:xs) = if x == ' ' then x:returnSpaces xs
                      else []

dropSpaces :: [String] -> [String]
dropSpaces [] = []
dropSpaces (x:xs) = if (spaces >= 2) then
                                ("\t" ++ drop spaces x):dropSpaces xs
                    else x:dropSpaces xs
                  where spaces = length (returnSpaces x)

flatten :: [String] -> String
flatten [] = ""
flatten (x:xs) = x ++ flatten xs

replace :: String -> String
replace [] = []
replace x = flatten(map reverse (dropSpaces (map reverse (makeGroups8 x))))
```