0630386

THE UNIVERSITY of York

Degree Examinations 2010–2011

DEPARTMENT OF COMPUTER SCIENCE

Functional Programming

Time allowed: **one and a half hours**

Answer any **two** questions.
Do not use red ink.

1 (25 marks)

Some puzzles require the solver to find words in rectangular grids of letters. For example, WORD appears in the ↗ direction in the grid:

```
A B C D
E F R G
H O I J
W K L M
```

Suppose there is only one word to find, and it can occur only in one of the directions →, ↓, ↘ or ↗. We might define:

```
type Word = String
type Grid = [String]
data Direction = Right | Down | Downright | Upright | None

find :: Grid -> Word -> Direction
find g w =
  if      w `within` g then Right
  else if w `within` transpose g then Down
  else if w `within` diagonals g then Downright
  else if w `within` diagonals (reverse g) then Upright
  else None
  where
  w `within` ss = any (subString w) ss
```

(i) [4 marks] The function `any :: (a -> Bool) -> [a] -> Bool` is used to define `` `within` ``. The result of `any p xs` is `True` if some element of `xs` satisfies `p` and `False` otherwise. If `any` is defined by

```
any p = foldr <1> <2>
```

what could the expressions <1> and <2> be?

(ii) [6 marks] Now define `subString :: Eq a => [a] -> [a] -> Bool` so that `subString w s` is `True` exactly if `s` takes the form *prefix*++w++*suffix* (where *prefix* or *suffix* or both could be empty).

(iii) [3 marks] Recall that the function `transpose` can be defined by:

```
transpose [r]    = map (:[]) r
transpose (r:rs) = zipWith (:) r (transpose rs)
```

What is the *type* of the expression `(:[])` in the first equation? This expression is an example of a *section*; what is that? It is also a function; specify its result in terms of its argument.

(iv) [6 marks] If `g` is a list of $m > 0$ lists, each of $n > 0$ elements, `diagonals g` should be a list of all the $\searrow$ diagonals in `g`. For example, if

```
g = ["ABC",
     "DEF",
     "GHI"]
```

then `diagonals g = ["G","DH","AEI","BF","C"]` (or the same strings listed some other order). Consider this definition:

```
diagonals :: [[a]] -> [[a]]
diagonals g  =  [diag h | h <- g : ends g ++ ends g']
  where
  g'  =  transpose g

ends :: [a] -> [[a]]
ends [_]     =  []
ends (x:xs)  =  xs : ends xs
```

What must be the type of `diag`? Describe its result in terms of its argument, giving an example. Define `diag`.

(v) [6 marks] Finally, suppose the `Direction` type is extended to include the constructors `Left | Up | Upleft | Downleft`. Define a function

```
findAll :: Grid -> [Words] -> [Direction]
```

so that `findAll g ws` lists the directions in which the words in `ws` occur in the grid `g`. (**Note**: you may use `reverse` if you wish, but must define any other auxiliary functions used that do not appear elsewhere in this question.)

2 (25 marks)

An integer *association list* has type `[(Int,Int)]`, and each item `(i,v)` pairs a unique index `i` with an associated value `v`. In this question, such lists represent a *sparse vector*, in which many of the values are zero. Explicit association pairs are only needed for the non-zero values. Study the following definitions.

```
lookUp :: Int -> [(Int,Int)] -> Int
lookUp j []              =  0
lookUp j ((i,x):ixs)     =  if i==j then x
                            else lookUp j ixs

update :: Int -> Int -> [(Int,Int)] -> [(Int,Int)]
update j y []            =  [(j,y)]
update j y ((i,x):ixs)   =  if i==j then (i,y):ixs
                            else (i,x):update j y ixs
```

(i) [4 marks] The `update` function records a list entry whatever the value of `y`. Revise the definition so that zero values are never recorded — an update with a zero value should remove any non-zero entry for the same index.

(ii) [4 marks] It might be more efficient to hold the list items in index order. Further revise the definitions of `lookUp` and `update` to implement this idea.

(iii) [5 marks] The original definition of `lookUp` could have been expressed as an application of `foldr`. The following definition has expressions missing at points <1> and <2>.

```
lookUp j vec  =  foldr f <1> vec
                 where
                 f (i,x) lu  =  <2>
```

What must be the type of the function `f`? What expressions could be provided at points <1> and <2> to complete a correct definition?

(iv) [1 mark] The original definition of `update` could *not* have been expressed as a `foldr` application in a similar way. Explain why not.

(v) [8 marks] For the original definitions, prove by list induction on `vec` the equivalence:

```
lookUp k (update k z vec) = z
```

You may ignore the possibility of $\perp$ values. Carefully justify each step in your proof. (**Hints**: Follow the structure of the definitions. The inductive case can be split into two sub-cases, one where the result of an equality test is `True` and the other where it is `False`.)

(vi) [3 marks] For the original definitions, we might also expect (again ignoring the possibility of $\perp$ values)

```
update k (lookUp k vec) vec = vec
```

but this equation does *not* hold. Give an example for which it fails, specifying `k`, `vec` and the vector computed by the left-hand side.

3 (25 marks)

A function `pt :: Int->Int->Int` is so defined that for any positive arguments `r` and `c`, if $c \leq r$ then `pt r c` is the element in row `r` and column `c` of Pascal's Triangle.

```
pt r c  =  if c==1 || c==r then 1
           else pt (r-1) (c-1) + pt (r-1) c
```

(i) [4 marks] If `r` is positive and `c>r` then we may assert that

$$\texttt{pt r c} \Longrightarrow \bot .$$

What does this assertion mean and why is it true?

(ii) [4 marks] If `pt r c` $\Longrightarrow$ `n`, where `n` is a well-defined number, express in terms of `n` the number of `pt` applications reduced during the evaluation. Justify your answer.

(iii) [4 marks] Using `pt` as an auxiliary, Pascal's Triangle can be defined as an infinite list of rows in which row n contains n elements. Complete the following definition, by supplying an appropriate expression for the position marked `<1>`. Explain how the completed definition works.

```
pascal  =  map row [1..]
  where
  row r  =  <1>
```

(iv) [4 marks] Greater efficiency can be achieved by making `pascal` and `pt` *mutually* recursive — using `pascal` as a *lazy tabulation* of `pt` results. Give a revised definition of `pt` implementing this idea. (Recall that `!!` indexes lists from 0.)

(v) [5 marks] What is it about the `pt` function and its applications when computing Pascal's Triangle that makes tabulation an effective way to save computational work? (**Hint:** How are the reduction counts associated with each number `n = pt r c` in the triangle altered as a result of tabulation?)

(vi) [4 marks] Give example arguments `r` and `c` for which the original `pt` and the memoised version yield different results. State the result in each case and explain the discrepancy. (**Hint**: think about strictness.)