36

0630386

THE UNIVERSITY *of York*

Degree Examinations 2003 - 2004

DEPARTMENT OF COMPUTER SCIENCE

Functional Programming

Time allowed: One and a half hours.

Calculators may **not** be used in this examination.

Candidates should 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? Alternatively, if `any` is defined by `any p = <3> . filter p` what could `<3>` 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] Define the function `diagonals :: [[a]] -> [[a]]` so that `diagonals g` lists all the ↘ diagonals in `g`. You may assume that `g` is list of m lists, each of n elements, where $m > 0$ and $n > 0$. For example, if

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

then `diagonals g = ["G","DH","AEI","BF","C"]` (or the same strings listed in any other order you find convenient).

(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**: any auxiliary functions that do not appear elsewhere in this question must be defined in full and briefly explained.)

2 (25 marks)

A datatype for *labelled trees* is defined by

```
data Tree a = T a [Tree a]
```

where `T x ts` represents a tree with a root node labelled `x` and immediate subtrees represented by the items of `ts`. For example:

```
      0
    / | \
   1  2  5
     / \
    3   4
```

```
example = T 0 [T 1 [], T 2 [T 3 [], T 4 []], T 5 []]
```

(i) [5 marks] Define a function `prune :: Int -> Tree a -> Tree a` so that the result of `prune n t`, for non-negative `n`, is a tree like `t` but with any nodes more than `n` generations from the root removed. Outline the reduction of `prune 1 example` to its result `T 0 [T 1 [], T 2 [], T 5 []]`.

(ii) [4 marks] A function `tree` is defined by:

```
tree f x = T x (map (tree f) (f x))
```

What is the *polymorphic type* of `tree`? Describe its result in terms of its arguments.

(iii) [3 marks] Draw a diagram of the tree represented by `prune 2 ham`, where `ham` is defined by:

```
ham = tree (\r -> filter asc (map (:r) [2,3,5])) []
      where
      asc (x:y:_) = x <= y
      asc _       = True
```

(iv) [6 marks] The function `foldTree` is defined by:

```
foldTree :: (a -> [b] -> b) -> Tree a -> b
foldTree f (T x ts) = f x (map (foldTree f) ts)
```

Give concise but informal specifications, including an illustrative example, for each of the following functions:

```
f1 = foldTree T
f2 = foldTree (\x ys -> 1 + sum ys)
f3 = foldTree (T . product)
```

(**Hint:** `f1` and `f2` are polymorphic in the label type, but `f3` can be applied only to trees with a specific type of label.)

(v) [7 marks] Finally, consider a *state-space search* problem specified by three parameters:

```
goal :: State -> Bool
init :: State
succ :: State -> [State]
```

A solution to the problem is a list of states: the first must be `init`, the last must satisfy `goal`, and for all consecutive states `x`, `y` the list `succ x` must contain `y`. Define a function

```
solve :: (State -> Bool) -> Tree State -> [State]
```

so that the result of `solve goal (tree init succ)` is a *shortest* solution. (**Note**: you may assume that a solution exists and that the tree is finite; any auxiliary functions that do not appear elsewhere in this question must be defined in full and briefly explained.)

3 (25 marks)

Consider the following definitions.

```
prodWith f []     ys = []
prodWith f (x:xs) ys = map (f x) ys ++ prodWith f xs ys

map f []     = []
map f (x:xs) = f x : map f xs

[]     ++ ys = ys
(x:xs) ++ ys = x : (xs ++ ys)

or []     = False
or (x:xs) = x || or xs

disj xs ys = not (or (prodWith (==) xs ys))
```

(i) [6 marks] For both `prodWith` and `disj` give *polymorphic types*, concise informal specifications and illustrative applications to short but non-empty list arguments.

(ii) [4 marks] Given the defining equations

```
not True = False     True || q = True     True && q = q
not False = True     False || q = q       False && q = False
```

show in detail that the law

```
not (p || q) = not p && not q
```

holds for all boolean expressions `p, q`.

(iii) [6 marks] Show by list induction ~~that~~ the following law holds for all boolean list expressions `xs`, `ys`. As before, full details of the proof are required.

```
or (xs ++ ys) = or xs || or ys
```

(iv) [1 mark] If the `elem` function is defined by the equations

```
elem x []     = False
elem x (y:ys) = x==y || elem x ys
```

how would you prove the law

```
or . map (x==) = elem x
```

by list induction? (Show only how to make list induction applicable; you need not give a detailed proof.)

(v) [8 marks] Use *fold/unfold transformation* with the laws from previous parts of the question to obtain a directly recursive definition of `disj` that no longer uses `or`, `map` or `prodWith`. (**Note**: give the derivation in full, not just the final program.)