The University of Nottingham

SCHOOL OF COMPUTER SCIENCE
AND INFORMATION TECHNOLOGY

A LEVEL 1 MODULE, SPRING SEMESTER 2006-2007

FUNCTIONAL PROGRAMMING

Time allowed TWO hours

Candidates must NOT start writing their answers until told to do so

Answer QUESTION ONE and THREE other questions

Marks available for sections of questions are shown in brackets in the right-hand margin.

No calculators are permitted in this examination.

Dictionaries are not allowed with one exception. Those whose first language is not English may use a dictionary to translate between that language and English provided that neither language is the subject of this examination.

No electronic devices capable of storing and retrieving text, including electronic dictionaries, may be used.

DO NOT turn examination paper over until instructed to do so

ADDITIONAL MATERIAL: Haskell Standard Prelude

Question 1 (Compulsory)

1. What are the values of the following expressions (9)

   (a) `[(x,y+2) | x <- [1 .. 4], y <- [1,5,7], y < 5]`

   (b) `filter k [1,2,5,7] where k x = x <= 5`

   (c) `phi 4 where phi x = if x == 1 then 1 else 1 + phi (x-1) * phi (x-1)`

2. The Altenkirch numbers were defined by the famous mathematician Altenkirch as follows. The first Altenkirch number is 1 and the second Altenkirch number is 1. Every other Altenkirch number is obtained by taking all the previous Altenkirch numbers, writing them down backwards, adding all the pairs and multiplying the elements of the resulting list. The first 3 Altenkirch numbers are thus 1, 1, 4 and hence the fourth Altenkirch number is 50 as can be seen by the following calculation:

```
      1  1  4
      +  +  +
      4  1  1
      --------
prod  5  2  5   = 50
      --------
```

   Use recursion to define a function `altenkirch:Int -> [Int]` which takes as input a positive integer n and returns a list containing the first n Altenkirch numbers (4)

3. Suppose that position in the plane of a robot and the moves of the robot are represented by the following types:

```
type Position = (Int,Int)
data Step     = Left | Right | Up | Down
```

   (a) Define a function `move :: Step -> Position -> Position` that returns the new position that results from taking a step from a starting position. (4)

   (b) Now suppose that a route is represented as a list of steps:

   `type Route = [Step]`

   Define a function `positions::Route->Position->[Position]` that returns all positions that arise by following a route from a starting position. (4)

(c) Define a function `equiv :: Route -> Route -> Bool` that decides if two routes starting from the origin have the same final position. The origin is the position `(0,0)`. (4)

Question 2:

1. Define a function `merge :: Ord a => [a] -> [a] -> [a]` and another function `msort :: Ord a => [a] -> [a]` which implement the merge sort algorithm. (6)

2. Define a function `mergeBy :: Ord b => (a -> b) -> [a] -> [a] -> [a]` and another function `msortBy :: Ord b => (a -> b) -> [a] -> [a]` which implement the higher order merge sort algorithm. (6)

3. A City finance firm is interested in keeping track of its portfolio of investments. Investments are placed in four different markets for which we have a datatype.

   ```
   data Market = Gold | Property | Shares | Bonds
   ```

   Each investment is recorded by a code which is an integer, the market it belongs to and a list of prices over successive time periods with the most recent price first in the list. All markets use the same time period. This is modelled by the type declarations

   ```
   type Code  = Int
   type Price = [Int]
   type Investment = (Code, Market, Price)
   ```

   Thus for example the investment `(102,Gold,[95, 106, 101, 100])` would represent an investment with code `102` in the market `Gold` which opened at `100`, rose to `101` in the next unit of trading, rose again to `106` at the end of the second period of trading and then fell to `95` at the end of the third period of trading.

   Use higher order sorting to define the following functions

   (a) `sortProfit :: [Investment] -> [Investment]` which takes as input a list of investments and returns that list sorted so that one investment comes before another if it has made more profit. The profit on an investment is the last trading price minus the original trading price. (3)

   (b) The total amount of profit ignores the length and scale of the investment. A better guide is the weighted profit which is defined to be the profit divided by the number of trading periods and by the original trading price of the investment

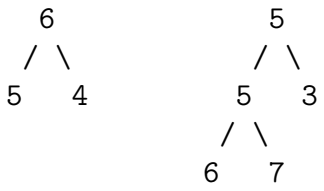
   Define `sortWtProfit :: [Investment] -> [Investment]` which takes as input a list of investments and returns that list sorted so that one investment comes before another if it has a higher weighted profit. (4)

(c) In order to assess which investments may have reached their peak, define a function `sortDown :: [Investment] -> [Investment]` which sorts a list of investments so that those investments whose value has gone down for the largest consecutive number of time periods until the present comes first. For example the investment `(102,Gold,[95, 106, 101, 100])` has gone down 1 time period, ie from `106` to `95`, while `(102,Gold,[97,95, 106, 101, 100])` has gone down for zero time periods since the last time period saw an increase in its price. (6)

Question 3:

1. Define the polymorphic type `Tree a` of binary trees which store data of type `a` in both the leaves and nodes of the tree. (3)

2. Define expressions `tree1` and `tree2` which represent the following trees.(3)

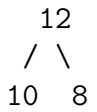


3. Define a function `sizeTree :: Tree a -> Int` which takes such a binary tree as input and returns the number of elements stored in the tree. For example `sizeTree tree1 = 3` and `sizeTree tree2 = 5` (3)

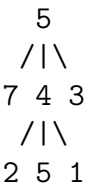
4. Define a function

```
mapTree :: (a -> b) -> Tree a -> Tree b
```

which takes a function and a binary tree as input and returns the tree obtained by applying the function to each piece of data stored in the tree. For example, `mapTree (2*) tree1` would be the tree whose graphical representation is (6)



5. The datatype `GTree a = GNode a [GTree a]` represents trees with an arbitrary number of branches. An expression of type `GTree a` is thus of the form `GNode x xs` where `x` is the data stored at the node and `xs` is the list of subtrees. Write an expression `gtree` of type `GTree Int` which represents the following tree. (4)



6. Define a function `sizeGTree :: GTree a -> Int` which takes a `GTree` as input and returns the the number of elements stored in the tree. For example, `sizeGTree gtree = 7`. (6)

Question 4:

Each video in a video store has a three digit number as its identification. Different copies of the same video have the same identification number. Each customer of the store can borrow at most three videos. Customers are represented as a string for their name and three numbers for the videos that they may borrow. If any of these numbers are zero, this indicates that no video has been borrowed. Thus a customer `‘‘Neil’’` with videos `271`, `311` and `345` on loan maybe represented by `cust1` while a customer `‘‘Fred’’` with only video `271` on loan may be represented by `cust2`.

```
cust1 = (‘‘Neil’’, 311, 271, 345)
cust2 = (‘‘Fred’’, 0,   271,   0)
```

Note that in `cust2`, the `271` could be in any position. The following type definitions are used to model these data structures.

```
type Name     = String
type Customer = (Name,Int,Int,Int)
type VStore   = [Customer]

vstore = [cust1, cust2]
```

1. Define a function `myloans :: VStore -> Name -> [Int]` which takes a video store and a members name as input and returns the list of video numbers that member has on loan. For example, (5)

   ```
   myloans vstore ‘‘Fred’’ = [271]
   ```

   Hint: Find the customers entry in the video store and put the video indentification numbers which are not zero into a list

2. Define a function `members :: VStore -> [Name]` which returns the list of names of members of the video store. For example, (3)

   ```
   members vstore = [‘‘Neil’’, ‘‘Fred’’]
   ```

   Hint: Use `map` *or list comprehensions to transform each 4-tuple in the video store into the name of the customer*

3. Define a function `owners :: VStore -> Int -> [Name]` which takes as input a video store and a video identification number and returns the names of members who have that video out on loan. For example, (5)

   ```
   owners vstore 271 = [‘‘Neil’’, ‘‘Fred’’]
   ```

   Hint: Use list comprehension

4. Define a function `borrow :: VStore -> Name -> Int -> Store` which takes as input the name of a video store, the name of a member and a video identification number.

   (a) If the member has less than three videos on loan, the result of `borrow` is the video store updated to show that the video is now on loan to the member.

   (b) Otherwise, the result of `borrow` is the original video store

   (6)

5. Define a function `return :: VStore -> Name -> Int -> Store` which takes as input a video store, the name of a member and a video identification number.

   (a) If the member has that video out on loan, the result of `return` is the video store is updated to show that the member no longer has that video on loan.

   (b) Otherwise, the result of `return` is the original video store.

   (6)

Question 5:

Write clear and precise descriptions of each of the following concepts that arise within functional programming. Make sure your answer explains the practical benefits of these concepts and also includes both simple and more complex examples.

1. Pattern Matching (eg, wildcards, tuple patterns etc) (8)
2. Higher Order Functions (7)
3. Algebraic types (10)

You may not use examples taken from other questions in this paper, your answers to them, or the standard prelude.

Answers
Question 1

```
1a. [(1,3), (2,3), (3,3), (4,3)]
1b. [1,2]
1c. 26

2.

altenkirch 1     = [1]
altenkirch (n+1) = prod (zipWith (+) l (reverse l)) : l
                   where l = altenkirch n

3.
move :: Step -> Position -> Position
move Up    (m,n) = (m,n+1)
move Down  (m,n) = (m,n-1)
move Left  (m,n) = (m-1,n)
move Right (m,n) = (m+1,n)

positions :: Route -> Position -> [Position]
position []     p = [p]
position (s:ss) p = p: position ss (move s p)

equiv :: Route -> Route -> Bool
equiv r1 r2 = last (position r1 (0,0)) == last (position r2 (0,0))
```

Question 2:

```
a)
merge :: Ord a => [a] -> [a] -> [a]
merge [] ys = ys
merge xs [] = xs
merge (x:xs) (y:ys)
    | x<y         = x : merge xs (y:ys)
    | otherwise   = y : merge (x:xs) ys

msort :: Ord a => [a] -> [a]
msort []  = []
msort [x] = [x]
msort xs  = merge (msort ws) (msort ys)
            where (ws,ys) = (take n xs, drop n xs)
                        n = length xs 'div' 2
```

```
b)
mergeBy :: Ord b => (a -> b) -> [a] -> [a] -> [a]
mergeBy f [] ys = ys
mergeBy f xs [] = xs
mergeBy f (x:xs) (y:ys)
    | f x< f y    = x : mergeBy f xs (y:ys)
    | otherwise   = y : mergeBy f (x:xs) ys

msortBy :: Ord b => (a -> b) -> [a] -> [a]
msortBy f []  = []
msortBy f [x] = [x]
msortBy f xs  = mergeBy f (msortBy f ws) (msortBy f ys)
            where (ws,ys) = (take n xs, drop n xs)
                        n = length xs 'div' 2

c)


sortProfit xs = msortBy profit xs
                where profit (c,m,ps) = last ps - head ps


sortWtProfit xs = msortBy wtprofit xs
                  where wtprofit (c,m,ps) =
                      (last ps - head ps)/(last ps * length ps)


sortDown xs = msortBy down xs
              where down (c,m,ps)     = longdown ps
                    longdown []       = 0
                    longdown [x]      = 0
                    longdown (a:b:cs) = if a < b then 1 + longdown (b:cs)
                                                 else 0
```

Question 3:

```
a) data Tree a = Leaf a | Node (Tree a) a (Tree a)

b)

tree1 = Node (Leaf 5) 6 (Leaf 4)
tree2 = Node (Node (Leaf 6) 5 (Leaf 7)) 5 (Leaf 3)
```

```
c)

sizeTree :: Tree Int -> Int
sizeTree (Leaf x) = 1
sizeTree (Node t1 x t2) = sizeTree t1 + 1 + sizeTree t2

d)

mapTree :: (a -> b) -> Tree a -> Btree b
mapTree f (Leaf x)       = Leaf (f x)
mapTree f (Node t1 x t2) = Node (mapTree f t1) (f x) (mapTree f t2)

e)

data GTree a = GNode a [GTree a]

gtree1 = GNode 5 [GNode 7 [],
                  GNode 4 [GNode 2 [], GNode 5 [], GNode 1 []],
                  GNode 3 []]

f)

sizeGTree :: GTree Int -> Int
sizeGTree (GNode x xs) = 1 + sum (map sizeGTree xs)


g)

mapGTree :: (a -> b) -> GTree a -> GTree b
mapGTree f (GNode x xs) = GNode (f x) (map (mapGTree f) xs)
```

Question 4:

```
myloans :: VStore -> Name -> [Int]
myloans [] x                = error (x ++ `` not a member of the video store'')
myloans ((n,v1,v2,v3):xs) x = if n == x then filter (/= 0) [v1,v2,v3]
                                        else myloans xs x

b)

members :: VStore -> [Name]
members = map name
          where name (n,v1,v2,v3) = n
```

c)

```
owners :: VStore -> Int -> [Name]
owners vs k = [n | (n, v1, v2, v3) <- vs, v1 == k || v2 == k || v3 == k]
```

d)

```
borrow :: VStore -> Name -> Int -> VStore
borrow ((n,v1,v2,v3):vs) n1 k1
  | n /= n1          =  (n,v1,v2,v3) : borrow vs n1 k1
  | v1 == 0          =  (n,k1,v2,v3) : vs
  | v2 == 0          =  (n,v1,k2,v3) : vs
  | v3 == 0          =  (n,v1,v2,k3) : vs
  | otherwise        =  (n,v1,v2,v3) : vs
```

e)

```
return :: VStore -> Name -> Int -> VStore
return ((n,v1,v2,v3):vs) n1 k1
  | n /= n1          =  (n,v1,v2,v3) : return vs n1 k1
  | v1 == k1         =  (n,0 ,v2,v3) : vs
  | v2 == k1         =  (n,v1,0 ,v3) : vs
  | v3 == k1         =  (n,v1,v2,0 ) : vs
  | otherwise        =  (n,v1,v2,v3) : vs
```

Question 5:

a) Pattern matching is a layout devise for defining functions on structured data. For example, the function which tests if an integer is zero maybe written

```
isZero :: Int -> Bool
isZero 0 = True
isZero - = False
```

The advantage is that code is shorter and clearer to read, write and understand. More complex forms of pattern matching arise in defining functions over algebraic types and when patterns are constructors and over tuple types. Recursion also supports patterns..

b) A higher order function is a function which takes as one of its inputs another function. A simple example is the function iterate which applies a function to its argument a given number of times

```
iterate :: (a -> a) -> Int -> a -> a
iterate f 0     x = x
iterate f (n+1) x = f (iterate f n x)
```

A more complex example is given by higher order sorting where various different sorts are achieved by making a function to describe the sort an input to the sorting algorithm

c) Algebraic types allow the user to define their own datatypes.
An element of an algebraic type is given by a constructor and certain inputs. Thus if we define

```
data Shape = Rect Int Int | Square Int | Circle Int
```

we have three possible shapes which are rectangles, squares and circles. To specify a rectangle one must supply two integers, eg Rect 3 4, while a circle requires only one, eg Circle 7

More advanced examples are recursive and polymorphic, eg the generalised trees

```
data GTree a = GNode a [GTree a]
```

Defining functions with algebraic types is aided by the fact that constructors are patters and so one may write

```
perim :: Shape -> Float
perim (Rect x y) = 2*(x+Y)
perim (Square x) = 4 * x
perim (Circle x) = pi * x
```