The University of Nottingham

SCHOOL OF COMPUTER SCIENCE

A LEVEL 1 MODULE, SPRING SEMESTER 2007-2008

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)

The objective of the game *Hangman* is to guess an unknown word. At each turn the guesser chooses a letter and all occurrences of this letter in the unknown word are made known to the guesser. If there are no occurrences of the letter in the word, then the guesser looses a life.

We model a game of *Hangman* by a triple consisting of the unknown word to be guessed, the list of guesses that have already been made, and a list of guesses that will be made. For example, the triple `(``Newcastle'', ``ae'', ``sdqkrlopmbvxz'')` represents the game of `Hangman` where the unknown word is `Newcastle`, two guesses have been made, namely `a` and `e`, the next guess will be `s`, the following guess will be `d` and so on. Thus a game of *Hangman* is modelled as an expression of type `Hang` which is defined as follows:

```
type Word = String
type CGuess = String
type FGuess = String
type Hang = (Word,CGuess,FGuess)
```

where `Word` represents the word to be guessed, `CGuess` is the list of letters that have been guessed so far and `FGuess` is the list of future guesses.

Each game of `Hangman` is in one of three states: (i) `Win` if the guesser has won the game, that is the guesser has lost less than 11 lives and guessed all of the letters in the unknown word; (ii) `Lose` if the guesser has lost more than 10 lives; and (iii) `Alive` otherwise. These states are represented by the datatype declaration

```
data Result = Win | Lose | Alive
              deriving (Show,Eq)
```

1. Define a function `diff::Eq a => [a]->[a]->[a]` which takes as input two lists and returns the list consisting of all elements in the first list which are not in the second list. (4)

2. Define a function `status::Hang->Result` which returns the status of a game of Hangman. You may wish to use the function `diff`. (4)

3. One way to implement *Hangman* is to repeatedly make a guess until the game is no longer `Alive` or there are no more guesses. As the first step in implementing this algorithm, define a function `repeatUntil` which takes three inputs: a function, a test (that is a function returning a boolean) and an expression. If the expression satisfies the test, then `repeatUntil` returns the expression. Otherwise, `repeatUntil` repeatedly applies the function to the expression until the test is satisfied. For example, if `double x = 2*x` then

```
repeatUntil double (>10) 2 = 16
```

(5)

4. Define a function `noMoreGuess::Hang->Bool` which takes as input a game of *Hangman* and returns `True` if no more guesses can be made. This occurs whenever the list of guesses to be made is empty or the game is no longer `Alive`. (2)

5. Define a function `makeGuess::Hang->Hang` which makes a guess, that is `makeGuess` takes the next guess to be made and adds it to the list of guesses which have been made. (2)

6. Hence define a function `hangman::(Word,Word)->Hang` which takes a pair of words as input, treats the first word as the unknown word and the second as a list of guesses and returns the state of the Hangman game after the guesser has either won or lost or there are no further guesses. (4)

7. Define a variant `hangman2::(Word,Word)->Hang` which behaves exactly as `hangman` does, except that if the guesser chooses a letter which has already been guessed, then that letter is ignored and not counted as a lost life. (4)

Question 2:

1. Give an implementation of the higher order mergesort algorithm, that is, define functions

```
mergeBy :: Ord b => (a -> b) -> [a] -> [a] -> [a]
msortBy :: Ord b => (a -> b) -> [a] -> [a]
```

   such that `msortby` sorts a list so that an element `x` comes before an element `y` if `f x <= f y`. (6)

2. In an election the result of each constituency is a list of the parties standing in that constituency together with the number of votes that party secured. The overall election result consists of a list of such results. Such data is represented using the type definitions

```
type Party        = String
type Result       = [(Party,Int)]
type ElectionRes  = [Result]
```

   Use the higher order mergesort algorithm to answer the following questions

   (a) Define a function `decl :: Result -> Result` which takes as input the results of a constituency and sorts the results so that the parties occur in descending number of votes. (2)

   (b) Define functions

```
winVotes,secVotes,totVotes :: Result -> Int
```

   which take as input the result of a constituency and return the number of votes obtained by the winner, the number of votes obtained by the party coming second and the total number of votes cast in the constituency. You may wish to use `decl` to calculate which party won the election and which party came second. (6)

   (c) Define a function `sortWinVotes::ElectionRes->ElectionRes` which takes as input the results of an election and sorts the results into descending order of the number of votes obtained by the winning party in each constituency. (2)

   (d) Define a function `sortMaj::ElectionRes->ElectionRes` which takes as input the results of an election and sorts the results into descending order of the majority in each constituency. The majority in a constituency is the difference between the number of votes of the winning party and the number of votes of the second party. (2)

(e) Define a function `sortPerCent::ElectionRes->ElectionRes` which takes as input the results of an election and sorts the results into descending order of the percentage share of the total vote obtained by the winner in each constituency. (3)

(f) Define a function

```
sortMarginal::ElectionRes->Party->Party->ElectionRes
```

such that `sortMarginal e p1 p2` is the result of sorting those constituencies where `p1` won and `p2` came second, by the majority of `p1` over `p2` . (4)

Question 3: The paper-rock-scissors game is played by two people. Behind their back, each player forms a hand into the shape of either a piece of paper (a flat palm), a rock (a clenched fist) or a pair of scissors (two fingers extended). Simultaneously, both players show their hands and the winner is determined by the rules

- paper beats rock,
- rock beats scissors, and
- scissors beats paper

If both players produce the same object, then neither player wins. This question models a series of rounds of the paper-rock-scissors game. To do so we first introduce some relevant datatypes

```
data Move  = Paper | Rock | Scissors
           deriving Show
type Round = (Move, Move)
```

1. Define a function `beats :: Move -> Move -> Bool` which returns `True` if the first move beats the second move and `False` otherwise. (3)
2. Define a function `score :: Round -> (Int,Int)` which returns `(1,0)` if the result of the round is a win for the first player, `(0,1)` if the result of the round is a win for the second player and `(0,0)` otherwise. (3)
3. Each player in the game will be represented by a strategy which is a function taking as input a list of moves and returning the next move for the player. We think of the input list of moves as being the moves previously made by the opponent. Thus we make the type definition

   ```
   type Strategy = [Move] -> Move
   ```

   (a) One strategy, called `follow :: Strategy`, looks at the list of the opponent's moves and plays the last move. The first move of `follow` is unspecified and so can be chosen to be any move. Define the function `follow`.

   (b) Another strategy called `smart` firstly calculates how many times the opponent has chosen paper, rock and scissors and then secondly chooses a move based upon their relative frequency. Thus smart is defined by

   ```
   smart :: Strategy
   smart xs = choose (count xs)

   count xs :: [Move] -> (Int,Int,Int)
   ```

Define the function `count` which takes as input a list of moves and returns a triple consisting of the number of times paper, rock and scissors were chosen respectively. You do not need to define the function `choose`.

(5)

4. Define a function `nextround::(Strategy,Strategy)->[Round]->Round` which takes as input two players, represented by their strategies, and a list of rounds played so far and returns the next round, ie the pair consisting of the next move for each player as determined by their strategies. For example

```
nextround (follow,follow) [(Rock,Scissors),(Rock,Paper)]
                                                = (Paper,Rock)
```

since the first player looks at the second player's moves and chooses the last move, and similarly for the second player. (4)

5. Define a function `rounds::(Strategy,Strategy)->Int->[Round]` which takes a pair of strategies as input and a number and returns the result of playing that number of rounds. For example

```
rounds (follow, follow) 0 = []
rounds (follow, follow) 3 = [(Rock,Paper),
                             (Paper,Rock),
                             (Rock, Paper)]
```

where the first move of player one was chosen (by some random process) to be `Rock` and that of the second player was `Paper`. (4)

6. Finally, define a function `match::(Strategy,Strategy)->Int->(Int,Int)` which takes as input a pair of strategies and a number `n` and returns a pair consisting of the number of rounds won by the first player and the number of rounds won by the second player where the total number of rounds played is `n`. For example

```
match (follow, follow) 3 = (1,2)
```

since the first player won 1 round, namely the second round, while the second player won two rounds, namely the first and third. *Hint: Proceed as follows. First calculate the result of playing n rounds using the function* `rounds`. *Then work out who won each round using the function* `score`. *Then calculate the total number of rounds won by each player.* (6)

Question 4:

This question concerns the following datatype for representing arithmetic expressions.

```
data Exp = Val Int
         | Add Exp Exp
         | Mul Exp Exp
         | Sub Exp Exp
         | Div Exp Exp
         | Try Exp Exp
         deriving Show
```

The first five constructors should be familiar to you from the lectures. The `Try`-constructor is required in part (2b) and hence you can ignore it before.

1. Define a function `eval :: Exp -> Int` which takes an expression as input and returns the value of the expression. For example

   ```
   eval (Add (Sub (Val 7) (Val 2)) (Mul (Val 7) (Val 1))) = 35
   ```

   (8)

2. Assume we want to use the arithmetic expressions to model computation of positive integers with a maximum value where the only divisions that are allowed are ones where there is no remainder. In doing so a number of exceptions may arise

   - The exception `NegNumberException` occurs if part of the evaluation process produces a negative number.
   - The exception `MaxNumberException` occurs if part of the evaluation process produces a number larger than a given constant `maxInt`.
   - The exception `DivByZeroException` occurs if part of the evaluation process attempts to divide a number by `0`.
   - The exception `RemainderException` occurs if part of the evaluation process attempts to divide a number by another number if a remainder is produced.

   These exceptions are modelled by the datatype

   ```
   data Exception = NegNumberException
                  | MaxNumberException
                  | DivByZeroException
                  | RemainderException
                  deriving Show
   ```

The rest of this question defines a new evaluator which, if such an exception arises, will throw the exception. Thus we shall define a new evaluation function with type `eval2: Exp -> Excep Int` where

```
data Excep a = Throw Exception
             | Ok a
             deriving Show
```

(a) In evaluating expressions formed by the `Val`-constructor, we must check whether an integer lies between `0` and `maxInt` and throw the appropriate exception if not. Define a function `checkIntOk :: Int -> Excep Int` which does this.

(10)

(b) In evaluating a recursive expression such as `Add e1 e2`, we shall want to recursively evaluate `e1` and `e2`. If either of these computations throws an exception then so should the overall computation - it doesn't matter which exception is thrown. On the other hand, if these computations produce `Ok`-values, then the addition function should be applied to these values.

To capture this situation, define a function

```
applyIfOk :: (a -> b -> c) -> Excep a -> Excep b -> Excep c
```

which throws an exception if both of the second and third inputs are exceptions and applies the first input appropriately if not. (7)

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. Recursion (8)
2. List Comprehension (7)
3. Algebraic Data 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

```
type Word = String
type CGuess = String
type FGuess = String
type Hang = (Word,CGuess,FGuess)

data Result = Win | Lose | Alive
              deriving (Show,Eq)

a) member x []     = False
   member x (y:ys) = if x == y then True else member x ys

   diff :: Eq a => [a] -> [a] -> [a]
   diff [] gs = []
   diff (x:xs) gs = if member x gs then rs else x:rs
                    where rs = diff xs gs

b) status :: Hang -> Result
   status (ws,gs,fs)
      | length wrong < 11 $\&\&$ diff ws gs == [] = Win
      | length wrong > 10                         = Lose
      | otherwise                                 = Alive
      where wrong = diff gs ws

c) repeatUntil :: (a -> a) -> (a -> Bool) -> a -> a
   repeatUntil f p x = if p x then x else repeatUntil f p (f x)

d) noMoreGuess :: Hang -> Bool
   noMoreGuess (ws,gs,fs) = (fs == []) || status (ws,gs,fs) /= Alive

e) makeGuess :: Hang -> Hang
   makeGuess (ws,gs,f:fs) = (ws,f:gs,fs)


f) hangman  :: (Word,Word) -> Hang
   hangman (ws,fs) = repeatUntil makeGuess noMoreGuess (ws,[],fs)


g) hangman2 :: (Word,Word) -> Hang
   hangman2 (ws,fs) = repeatUntil makeGuess2 noMoreGuess (ws,[],fs)
                      where makeGuess2 (ws,gs,f:fs)
```

```
                              | member f gs = makeGuess (ws,gs,fs)
                              | otherwise  = (ws,f:gs,fs)
```

Question 2:

```
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 as) (msortBy f bs)
                where (as,bs) = (take l xs, drop l xs)
                      l       = length xs `div` 2


type Party = String
type Result = [(Party,Int)]
type ElectionRes     = [Result]

decl :: Result -> Result
decl cs = msortBy weight cs  where weight x = 0 - (snd x)

winVotes,secVotes,totVotes :: Result -> Int
winVotes = snd . head . decl
secVotes = snd . head . tail . decl
totVotes = sum . (map snd)

sortWinVotes :: ElectionRes -> ElectionRes
sortWinVotes es = msortBy winVotes es

sortMaj :: ElectionRes -> ElectionRes
sortMaj es = msortBy fun1 es
             where fun1 xs = winVotes xs - secVotes xs

sortPerCent :: ElectionRes -> ElectionRes
sortPerCent es = msortBy fun2 es
                 where fun2 xs = (winVotes xs)*100 `div` (totVotes xs)
```

```
sortMarginal :: ElectionRes -> Party -> Party -> ElectionRes
sortMarginal es p1 p2 = sortMaj (filter fun3 es)
                        where fun3 xs = (fst . head . decl) xs == p1 &&
                                        (fst . head . tail . decl) xs == p2
```

Question 3:

```
data Move  = Paper | Rock | Scissors
             deriving Show
type Round = (Move, Move)

beats :: Move -> Move -> Bool
beats Rock     Scissors = True
beats Scissors Paper    = True
beats Paper    Rock     = True
beats _        _        = False

score :: Round -> (Int,Int)
score (x,y)
  | x `beats` y = (1,0)
  | y `beats` x = (0,1)
  | otherwise   = (0,0)

type Strategy = [Move] -> Move

follow :: Strategy
follow [] = Rock
follow xs = last xs

smart :: Strategy
smart ms = choose (count ms)
           where
           count = foldl check (0,0,0)
           check (p,r,s) Paper    = (p+1,r,s)
           check (p,r,s) Rock     = (p,r+1,s)
           check (p,r,s) Scissors = (p,r,s+1)
           choose (p,q,r) = Rock

-- Note choose is defined this way only to ensure compilation
```

```
-- without recourse to random numbers in Haskell. Students need
-- only write count

nextround :: (Strategy,Strategy) -> [Round] -> Round
nextround (s1,s2) xs = (s1 (map snd xs), s2 (map fst xs))

rounds :: (Strategy,Strategy) -> Int -> [Round]
rounds (s1,s2) 0     = []
rounds (s1,s2) (n+1) = prevrounds ++ [nextround (s1,s2) prevrounds]
                       where  prevrounds = rounds (s1,s2) n

match :: (Strategy,Strategy) -> Int -> (Int,Int)
match (s1,s2) = total . map score . rounds (s1,s2)

total :: [(Int,Int)] -> (Int,Int)
total xs = (sum (map fst xs), sum (map snd xs))
```

Question 4:

```
eval :: Exp -> Int
eval (Val n) = n
eval (Add e1 e2)  = eval e1 + eval e2
eval (Div e1 e2)  = eval e1 `div` eval e2
eval (Mul e1 e2)  = eval e1 * eval e2
eval (Sub e1 e2)  = eval e1 - eval e2


maxInt :: Int
maxInt = 1000

data Exception = NegNumberException
               | MaxNumberException
               | DivByZeroException
               | RemainderException
               deriving Show

data Excep a = Throw Exception
             | Ok a
```

```
            deriving Show

applyIfOk :: (a -> b -> c) -> Excep a -> Excep b -> Excep c
applyIfOk f (Ok x) (  Ok y   )  = Ok (f x y)
applyIfOk f (Throw e) _         = Throw e
applyIfOk f _         (Throw e) = Throw e

checkIntOk :: Int -> Excep Int
checkIntOk n
    | n < 0      = Throw NegNumberException
    | n > maxInt = Throw MaxNumberException
    | otherwise  = Ok n



eval2 :: Exp -> Excep Int
eval2 (Val n)      = checkIntOk n
eval2 (Add e1 e2)  = case applyIfOk (+) (eval2 e1) (eval2 e2) of
                         Ok y    -> checkIntOk y
                         Throw e -> Throw e
eval2 (Mul e1 e2)  = case applyIfOk (*) (eval2 e1) (eval2 e2) of
                         Ok y    -> checkIntOk y
                         Throw e -> Throw e
eval2 (Sub e1 e2)  = case applyIfOk (-) (eval2 e1) (eval2 e2) of
                         Ok y    -> checkIntOk y
                         Throw e -> Throw e
eval2 (Div e1 e2)  = case (eval2 e2) of
                         Throw e -> Throw e
                         Ok 0    -> Throw DivByZeroException
                         Ok n    -> case eval2 e1 of
                                      Throw e -> Throw e
                                      Ok m    -> if m `mod` n == 0
                                                 then Ok (m `div` n)
                                                 else Throw RemainderException
```

Question 5:

```
a) Recursion is the basic building block for defining functions in
Haskell and ensures the language is Turing complete. A recursive function
```

is one which calls itself ie of the form

```
f x = exp
```

where exp contains f. For example

```
fac 0   = 1
fac (n+1) = (n+1) * fac n
```

More sophisticarted examples are the sorting algorithms

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
```