Exam in Functional Programming TDA450/DIT140

Friday 19 January 2007, 8.30 – 12.30.
Examiner: Bengt Nordström, phone 1033 or 0730-79 42 89
Permitted aids: English-Swedish or English-other language dictionary.

You are free to use any Haskell standard functions, including the attached ones, unless the question specifically forbids you. You may use the solution of an earlier part of a question to solve a later part, even if you did not solve the earlier part. You may lose marks for complicated solutions. Your code should be written in such a way that it is quickly obvious to the reader what the code does and that it is correct. This means that you should strive for the right balance between conciseness and clarity. If any part of your code is not easy to understand, it should be commented in an appropriate way. Note that superfluous comments make the code more difficult to read.

This written exam is worth up to 180 points. It is possible to count up to 20 points for assignments handed in during the fall term of 2006. We have the following limits for different grades: Chalmers students: 3 = 80 pts, 4 = 100 pts, 5 = 120 pts. University students: G = 100 pts, VG = 150 pts.

The exam will be shown on Wednesday 24 January at 11.00 – 11.15 in Bengt's office. Solutions to the exam will be available from the homepage of the course.

1. Consider the definition of the data type of binary trees:

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

In a tree of the form `Tree a t1 t2`, we will call `a` the *root* of the tree and `t1` and `t2` the *left and right subtree*, respectively. We will say that the tree is a *leaf* when it has the form `Leaf`.

Define the function (10)

```
depth ::  Tree a -> Integer
```

which computes the maximum depth of a tree. The depth of a leaf is defined to be 0.

2. A binary tree is ordered if it is a leaf or if its subtrees are ordered and the values in its left subtree holds only values which are less than the root and the right subtree holds only values which are greater or equal than the root. Write the functions (15)

```
minimal :: (Tree a) -> a
maximal :: (Tree a) -> a
```

which computes the minimal and maximal value of its argument, which is an ordered tree. You can assume that the argument is not a leaf. Notice that the types of the functions are *not*:

```
minimal :: Ord a => (Tree a) -> a
maximal :: Ord a => (Tree a) -> a
```

3. Write a function (20)

```
isordered :: Ord a => (Tree a) -> Bool
```

which checks if its argument is an ordered tree.

4. An inorder traversal of a tree is a traversal which first traverses the left subtree, then the root and finally the right subtree.

   Write a function (10)

```
inorder :: (Tree a) -> [a]
```

which computes the inorder traversal of its input.

5. Write a function (20)

```
insert :: Ord a => a -> (Tree a) -> Tree a
```

which inserts the element `a` into the tree `t`, i.e. `insert a t` is an ordered tree containing all values in `t` and `a`.

6. Write a function (10)

```
list2tree :: Ord a => [a] -> Tree a
```

which converts a list to an ordered tree by inserting (using the function `insert` above) all elements of the list into a tree.

7. Notice that the function above is almost a sorting algorithm, it takes a list and produces an ordered tree. Use this function to define (15)

   ```
   sort :: Ord a => [a] -> [a]
   ```

   which sorts its input.

8. Define the function (20)

   ```
   merge :: Ord a => [a] -> [a] -> [a]
   ```

   which merges two ordered lists, i.e. `merge as bs` is an ordered permutation of `as ++ bs`.

9. Define now a function (15)

   ```
   merges :: Ord a => [[a]] -> [a]
   ```

   which merges (not a pair of lists but) a list of lists such that the output is ordered and contains all elements of its input lists. You can for instance define it such that `merges [as, bs, ..., us, vs]` is equal to `(merge as (merge bs ( ... (merge us vs)...)))`

10. Define now a function (10)

    ```
    splitlist :: [a] -> [[a]]
    ```

    which splits a list of values into a list of elements, each element being a singleton list of one of those values. For instance, the value of `splitlist [1, 3, 4]` should be `[[1], [3], [4]]`.

11. Using the two functions `merges` and `splitlist` define now another sorting function (15)

    ```
    mergesort :: Ord a => [a] - [a]
    ```

12. Explain the difference between an overloaded function and a polymorphic function! Give examples (you can refer to earlier examples in this exam). (20)

Good Luck!
Bengt
PS The next pages contains a list of function definitions.

```
-- Numerical functions: --------------------------------------
(^) :: (Num a, Integral b) => a -> b -> a
x ^ 0        =  1
x ^ n|n > 0  =  x * x^(n-1)
     |True   =  error "Prelude.^: negative exponent"
gcd :: Integral a => a -> a -> a
gcd 0 0        =  error "Prelude.gcd: gcd 0 0 is undefined"
gcd x y        =  gcd' (abs x) (abs y)
                    where gcd' x 0 = x
                          gcd' x y = gcd' y (x `mod` y)
sum, product   :: Num a => [a] -> a
sum            =  foldr (+) 0
product        =  foldr (*) 1
-- Char functions:--------------------------------------------
isAscii c             =  fromEnum c < 128
isControl c           =  c < ' ' ||  c == '\DEL'
isPrint c             =  c >= ' ' &&  c <= '~'
isSpace c             =  c == ' ' || c == '\t' || c == '\n' ||
                         c == '\r' || c == '\f' || c == '\v'
isUpper c             =  c >= 'A'   &&  c <= 'Z'
isLower c             =  c >= 'a'   &&  c <= 'z'
isAlpha c             =  isUpper c  ||  isLower c
isDigit c             =  c >= '0'   &&  c <= '9'
isAlphanum c          =  isAlpha c  ||  isDigit c
toUpper, toLower      :: Char -> Char
toUpper c | isLower c
        = toEnum (fromEnum c - fromEnum 'a' + fromEnum 'A')
          | otherwise  = c
toLower c | isUpper c
        = toEnum (fromEnum c - fromEnum 'A' + fromEnum 'a')
          | otherwise  = c
ord                   :: Char -> Int
ord                    = fromEnum
chr                   :: Int -> Char
chr                    = toEnum
-- List functions: -------------------------------------------------
null           :: [a] -> Bool
null []        =  True
null (_:_)     =  False
head           :: [a] -> a
head (x:_)     =  x
tail           :: [a] -> [a]
tail (_:xs)    =  xs

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

init             :: [a] -> [a]
```

```
init [x]          = []
init (x:xs)       = x : init xs
(++)           :: [a] -> [a] -> [a]
[]     ++ ys   =  ys
(x:xs) ++ ys   =  x : (xs ++ ys)
concat         :: [[a]] -> [a]
concat         =  foldr (++) []
length         :: [a] -> Int
length   []    =  0
length (_:xs)  =  1 + length xs
reverse        :: [a] -> [a]
reverse []     =  []
reverse (x:xs) =  reverse xs ++ [x]
elem           :: Eq a => a -> [a] -> Bool
elem x []      =  False
elem x (y:ys)  =  x == y || elem x ys
take, drop     :: Int -> [a] -> [a]
take 0 _            = []
take _ []           = []
take n (x:xs) | n>0  = x : take (n-1) xs
take _ _            = error "PreludeList.take: negative argument"
drop 0 xs           = xs
drop _ []           = []
drop n (_:xs) | n>0  = drop (n-1) xs
drop _ _            = error "PreludeList.drop: negative argument"

replicate           :: Int -> a -> [a]
replicate 0 x       =  []
replicate n x | n>0 =  x : replicate (n-1) x

foldr             :: (a -> b -> b) -> b -> [a] -> b
foldr f e []      =  e
foldr f e (x:xs)  =  f x (foldr f e xs)
map            :: (a -> b) -> [a] -> [b]
map f []       =  []
map f (x:xs)   =  f x : map f xs
zip                :: [a] -> [b] -> [(a,b)]
zip (a:as) (b:bs)  =  (a,b):zip as bs
zip _      _       =  []
zipWith                  :: (a -> b -> c) -> [a] -> [b] -> [c]
zipWith z (a:as) (b:bs)   =  z a b : zipWith z as bs
zipWith _ _      _        =  []
filter          :: (a -> Bool) -> [a] -> [a]
filter p []     =  []
filter p (x:xs) |  p x        =  x:filter p xs
                |  otherwise  =    filter p xs
takeWhile      :: (a -> Bool) -> [a] -> [a]
takeWhile p []       = []
takeWhile p (x:xs)
```

```
         | p x       = x : takeWhile p xs
         | otherwise = []
dropWhile      :: (a -> Bool) -> [a] -> [a]
dropWhile p []       = []
dropWhile p (x:xs)
         | p x       = dropWhile p xs
         | otherwise = x:xs
span, break            :: (a -> Bool) -> [a] -> ([a],[a])
span p []            = ([],[])
span p xs@(x:xs')
            | p x       =  (x:ys,zs)
            | otherwise =  ([],xs)
                           where (ys,zs) = span p xs'
break p                =  span (not . p)
-- lines breaks a string up into a list of strings at newline characters.
-- The resulting strings do not contain newlines.  Similary, words
-- breaks a string up into a list of words, which were delimited by
-- white space.  unlines and unwords are the inverse operations.
-- unlines joins lines with terminating newlines, and unwords joins
-- words with separating spaces.
lines            :: String -> [String]
lines ""         =  []
lines s          =  let (l, s') = break (== '\n') s
                      in  l : case s' of
                                []      -> []
                                (_:s'') -> lines s''
words            :: String -> [String]
words s          =  case dropWhile isSpace s of
                      "" -> []
                      s' -> w : words s''
                            where (w, s'') = break isSpace s'
unlines          :: [String] -> String
unlines          =  concatMap (++ "\n")
unwords          :: [String] -> String
unwords []       =  ""
unwords ws       =  foldr1 (\w s -> w ++ ' ':s) ws
until          :: (a -> Bool) -> (a -> a) -> a -> a
until p f x    =  if p x then x else until p f (f x)
unzip            :: [(a,b)] -> ([a],[b])
unzip []         =  ([],[])
unzip ((a,b):xs)  =  let (as,bs) = unzip xs
                     in (a:as,b:bs)
nub :: (Eq a) => [a] -> [a]
nub []     = []
nub (x:xs) = x : [ y | y <- xs, x /= y ]
sort :: (Ord a) => [a] -> [a]
sort []     = []
sort (x:xs) = sort smaller ++ [x] ++ sort bigger
 where
```

```
  (smaller, bigger) = partition (< x) xs
and, or         :: [Bool] -> Bool
and             =  foldr (&&) True
or              =  foldr (||) False
any, all        :: (a -> Bool) -> [a] -> Bool
any p           =  or . map p
all p           =  and . map p
intersect              :: Eq a => [a] -> [a] -> [a]
intersect              =  intersectBy (==)
intersectBy            :: (a -> a -> Bool) -> [a] -> [a] -> [a]
intersectBy eq xs ys   =  [x | x <- xs, any (eq x) ys]
-- Standard combinators: ----------------------------------------------------
flip           :: (a -> b -> c) -> b -> a -> c
flip f x y     =  f y x
(.)            :: (b -> c) -> (a -> b) -> a -> c
(f . g) x      =  f (g x)
fst            :: (a,b) -> a
fst (x,_)       = x
snd            :: (a,b) -> b
snd (_,y)       = y
curry          :: ((a,b) -> c) -> (a -> b -> c)
curry f x y     = f (x,y)
uncurry        :: (a -> b -> c) -> ((a,b) -> c)
uncurry f p     = f (fst p) (snd p)
id             :: a -> a
id    x         = x
const          :: a -> b -> a
const k _       = k
error  :: String -> a   -- primitive function, no definiton here
```