EXAM

Introduction to Functional Programming

TDA555/INN040

DAY: 24 October 2006 TIME: 8.30 -- 12.30 PLACE: V-salar

Responsible:	Koen Lindström Claessen, Datavetenskap
	Emil Axelsson, Datavetenskap
Result:	Published 2 November, at the latest
Aids:	An English (or English-Swedish, or English-X) dictionary
Grade:	There are 4 assignments (with 16 + 15 + 25 + 3 = 59 points); a total of at least 20 points guarantees a pass

Please read the following guidelines carefully:

- Answers can be given in Swedish or English
- Begin each assignment on a new sheet
- Write your personal number on each sheet
- Write clearly; unreadable = wrong!
- Full marks are given to solutions which are short, elegant, efficient, and correct
- Less marks are given to solutions which are unnecessarily complicated or unstructured
- For each question, if your solution consists of more than 2 lines of Haskell code, include a short description of what your intention is with your solution
- You can use any standard Haskell function in your solution --- a list of some useful functions is attached
- You may use the solution of an earlier part of an assignment to help solve a later part --- even if you did not succeed in solving the earlier part!

Good Luck!

Assignment 1 – Encryption (total 16p)

In this assignment, we will investigate the encryption and decryption of messages. Encryption means to transform a plain text message into a code --- something that cannot be read by other people; decryption is the task of turning the encrypted message back into the original text.

Already in early times, people felt the need to encrypt their messages. For example, the Roman emperor Caesar wanted to hide the content of the messages that were sent between parts of his army, in the case a messenger was captured by the enemy and the message was intercepted.

Caesar is said to have invented one of the first encryption techniques. The idea is that each letter in the message is replaced by one which has shifted a given amount k of steps in the alphabet (Caesar used k=3). So, the letter A in a message would be replaced by the letter D (=A+3), the letter B by E, and so on. The alphabet is seen in a circular way here, so where for example the letter W would be replaced by Z, X would be replaced by A, Y by B, and Z by C.

For example, if Caesar would want to send the message:

```
S E N D M O R E Y O G U R T
```

The result would be:

```
V H Q G P R U H B R J X U W
```

And nobody would understand! To decode the message, the reverse process would have to be performed.

In Caesar's time, the alphabet had only 25 characters (the U and V were the same letter). In this assignment, we will make use of the standard ASCII alphabet used on most computers, which has 256 characters.

(1p) **Question A**

Define two functions:

```
(+.), (-.) :: Char -> Int -> Char
```

The idea is that `c +. k` shifts the character `c` with `k` steps in the ASCII alphabet. (Note that we make use of the full range 0..255 of character codes here, still using it in a circular way.) The function `(-.)` is the inverse of `(+.)` --- the expression `c -. k` shifts `c` with `k` steps in the other direction.

Examples:

```
Main> 'A' +. 17
'R'
Main> 'r' -. 17
'a'
Main> 'Z' +. 3
']'
Main> 'Y' +. 255
'X'
```

(1p) **Question B**

Implement two functions:

```
encrypt, decrypt :: Int -> String -> String
```

The function **`encrypt`**, given a shift distance and a message, encrypts the message, using Caesar's technique. The function **`decrypt`**, given a shift distance and an encrypted message, decrypts the message, assuming that the message has been encrypted using Caesar's technique.

Examples:

```
Main> encrypt 3 "SEND MORE YOGURT"
```

```
"VHQG#PRUH#\\RJXUW"
Main> decrypt 3 "VHQG#PRUH#\\RJXUW"
"SEND MORE YOGURT"
```

(2p) **Question C**

Define a QuickCheck property that states that encrypting and then decrypting a message with the same shift distance results in the original message.

(1p) **Question D**

In order to be able to test your property using QuickCheck, the type of characters **`Char`** needs to be an instance of the class **`Arbitrary`**. Give a suitable instance of **`Arbitrary Char`**.

Although it is possible to use Caesar's encryption method, it is not very good. For an enemy who knows the method of encryption, it is a simple matter of trying out all possible shift distances to get to know the original message. In Caesar's time, there were only 25 possible shift distances (do you see why?). For our implementation, there are 256. So, to *crack* an encrypted message, it would be enough to go through all possible ways it could have been encrypted, and pick the one that makes most sense.

For 25 possible ways, this would have been OK to do by hand, but for 256 this is a bit much. Therefore, we will write a function that does this for us.

We need to come up with a way of deciding, given 256 ways of decrypting a message, which one of the decrypted messages "makes most sense". One way to do this is to check the amount of *common characters* in the decrypted message. A common character is a character that occurs often in a piece of text. It depends on the language used which characters occur frequently. Often, if we use the wrong shift distance, the decrypted message will contain lots of characters that occur quite infrequently in normal text. But if we use the correct shift distance, the amount of common characters will be quite high.

In order to decide what characters are common in a message, we can look at and analyze texts in the same language as the message.

(4p) **Question E**

Implement a function:

```
common :: String -> [Char]
```

that, given a text, computes the 10 characters that occur most frequently in the text (these should of course all be different).

Example:

```
Main> do s <- readFile "text"; print (common s)
" etsoiahrn"
```

Now, we are ready to write a function that can crack an encrypted message. We simply try all possible decryptions, and produce the one that has the largest amount of common characters in it.

(4p) **Question F**

Implement a function:

```
crack :: FilePath -> String -> IO String
```

that, given a filename (containing a suitable text that we use in order to compute common characters), and an encrypted message, decrypts the message using the most likely shift distance, based on the amount of common characters in the given result.

Example:

```
Main> do s <- crack "text" "uzu1Z1t\131rt|1\133yv1t\128uvP"; print s
"did I crack the code?"
```

Finally, let us take a look at a method of encryption that is a little bit harder to crack: The Vigenère method, developed by the Frenchman Blaise de Vigenère in the 16-th century. The Vigenère method

is a bit like Caesar's method, although the encryption works with a finite sequence of different shift distances. Each shift distance from this sequence is applied to its own letter, by lining up the shift distance repeatedly with the message.

For example, to encrypt the message

```
H A S K E L L
```

with the shift distance sequence 3,2,4, we would shift H by 3, A by 2, and S by 4 steps. Then, we would shift K by 3 steps, E by 2 steps, and L by 4 steps, and so on, until we have seen all letters in the message. The final result would be:

```
K C W N G P O
```

(2p) **Question G**

Implement two functions:

```
encryptV, decryptV :: [Int] -> String -> String
```

that encrypt and decrypt a given message using a Vigenère-sequence of shift distances.

(1p) **Question H**

Define a QuickCheck property that states that encrypting and then decrypting a message using the Vigenère method with the same sequence of shift distances results in the original message. Think about what happens when the sequence is empty!

Note: *The Vigenère method of encryption is not very good either. There exist variants of the cracking method for this method as well. The only thing that is needed is to know the length of the shift distance sequence. Often, this sequence is quite short. However, we can increase the power of the method by increasing the length of the shift sequence.*

Assignment 2 – Summarizing Replies to a Questionnaire (total 15p)

In this assignment, we will develop a Haskell program that can summarize replies to a questionnaire, for example a course questionnaire. The questionnaire is web-based, and stores all replies it has gotten so far in a special file. We will not be bothered with reading in and analyzing this file; instead, we may assume that we know how to read this file to get a list of replies.

An example course questionnaire might look as follows:

1. What is your general impression of the course?
2. What did you think of the exercises classes?
3. What did you think of the fact that there was only one lecture in week 2?
4. What did you think of the lab assignments?
5. How difficult was the course for you?

A reply from one person is modelled by a list of pairs. An example of a reply is the following:

```
aReply :: Reply
aReply = [(1,"very good"), (3,"bad"), (4,"okay")]
```

This models the fact that the person who entered the answers, answered "very good" on question 1, "bad" on question 3, and "okay" on question 4. Note that not all questions have to be answered by a person. For example, the above person did not answer question 2 and 5.

Here is an example of a list of replies:

```
someReplies =
  [ [(1,"very good"), (3,"bad"), (4,"okay")]
  , [(2,"good"), (3,"bad"), (4,"good"), (5,"difficult")]
  , [(4,"okay"), (5,"very difficult")]
  ]
```

(1p) **Question A**

Give a suitable type definition of the type **`Reply`**.

What is the type of the function **`someReplies`**?

(1p) **Question B**

A reply should not have two answers for the same question. Define a function:

```
validReply :: Reply -> Bool
```

that checks this.

(2p) **Question C**

Define a function:

```
questions :: [Reply] -> [Int]
```

that, given a list of replies, returns all question numbers that were answered in any of the replies. The list should not contain any duplicates, and should contain the question numbers in the right order.

Example:

```
Main> questions someReplies
[1,2,3,4,5]
```

(2p) **Question D**

Define a function:

```
answers :: Int -> [Reply] -> [String]
```

that, given a question number and a list of replies, gathers all answers to this question given in any of the replies. Note: This list *can* contain duplicates!

Examples:

```
Main> answers 3 someReplies
["bad","bad"]
Main> answers 5 someReplies
["difficult","very difficult"]
```

(3p) **Question E**

Define a function:

```
summary :: [Reply] -> [(Int,[(Int,String)])]
```

that, given a list of replies, produces a table, containing, for each question, all answers that were given to that question, and how many times that answer was given. The answers should be sorted in such a way that the most frequent answer comes first.

Example:

```
Main> summary someReplies
[ (1,[(1,"very good")]), (2,[(1,"good")]),
    (3,[(2,"bad")]), (4,[(2,"okay"),(1,"good")]),
      (5,[(1,"very difficult"),(1,"difficult")]) ]
```

We can see for example that the answers to question 3 were 2 "bad"s, and the answers to question 4 were: 2 "okay"s and 1 "good".

(3p) **Question F**

Define a function:

```
summarize :: [Reply] -> IO ()
```

that, given a list of replies, prints out a summary of the replies, where also percentages are calculated.

Example:

```
Main> summarize someReplies
Q1: 100% very good
Q2: 100% good
Q3: 100% bad
Q4: 66% okay 33% good
Q5: 50% very difficult 50% difficult
```

(2p) **Question G**

To get a better overview of the actual answers that are given, we would like to produce an additional summary of the results, where answers like "good" and "very good" (or "difficult" and "very difficult") are grouped together. In this way, it is easier to see what percentage of answers are on the "right side".

Define a function:

```
mild :: [Reply] -> [Reply]
```

that, transforms all answers in the given replies into *milder* answers, by removing the word "very" from them.

Example:

```
Main> summarize (mild someReplies)
Q1: 100% good
Q2: 100% good
Q3: 100% bad
Q4: 66% okay 33% good
Q5: 100% difficult
```

Assignment 3 -- A Theorem Prover (total 25p)

In this assignment, we will develop a simple *theorem prover* for propositional logic. (Remember that propositional logic simply is a different name for boolean expressions.) In the end, our theorem prover will be able to decide if a given formula is *valid* or not. A formula is valid if it is true for all possible values of the variables.

An example of a valid formula is (x || not x), because it evaluates to true no matter what value we pick for x. An example of a non-valid formula is (x && y), because it is false when for example x is false.

We will work with the following datatype:

```
type Var = String

data Form = Var Var
          | And Form Form
          | Not Form
          | Bool Bool
```

Using the above type, the logical formula (x && y) is modelled by the Haskell expression **`(Var "x" `And` Var "y")`**.

An immediate observation is that there are no constructors for other logical operators, such as for example or and implication! Luckily, we know that (x || y) = not (not x && not y), and we also know that (x => y) = (not x || y).

(2p) **Question A**

Define two functions:

```
orr, impl :: Form -> Form -> Form
```

that calculate the or and implication of two formulas, respectively. You will have to express these in terms of the constructor functions you have. (The function **`orr`** is called that because the function name **`or`** is already taken.)

You may not change the **`Form`** datatype to do this!

(2p) **Question B**

Later on, we will have to know what variables occur in a given formula. Define a function:

```
vars :: Form -> [Var]
```

that calculates which variables occur in a given formula. The list of variables that is computed should not contain the same variable name twice.

Example:

```
Main> vars (Var "x" `And` (Var "x" `orr` Var "y"))
["x","y"]
```

(1p) **Question C**

Define a QuickCheck property that states that no variable name occurs twice in the result of the function **`vars`**.

In order to be able to test your property using QuickCheck, the **`Form`** type needs to be made an instance of the class **`Arbitrary`**. We have started with this already:

```
instance Arbitrary Form where
  arbitrary = sized arbForm
    where
      arbForm n =
        frequency
        [ (1, do s <- elements ["x","y","z"]
                 return (Var s))
```

```
  , (n, do p <- arbForm (n-1)
           return (Not p))
  ]
```

The above generator only generates formulas with variables "x", "y", and "z". There are also a few cases missing.

(2p) **Question D**

Explain in English or Swedish how this generator works: What role does the argument **`n`** to the function **`arbProp`** play? Why does it need to be there? What does **`frequency`** do?

(2p) **Question E**

Add the cases for **`And`** and **`Bool`** to the generator. (You do not have to copy the above code, just say where you make changes and what the changes are.)

The next step is to create a function that can compute the value of a given formula. To do this, we need to know what the values of the variables occurring in the formula are. We do this by specifying an *environment*. We model the environment as a function from variable names to values.

```
type Env = Var -> Bool
```

An example of an environment is the following:

```
example :: Env
example "x" = True
example _   = False
```

This environment assigns the value true to the variable x, and false to all other variables.

(3p) **Question F**

Define a function:

```
eval :: Env -> Form -> Bool
```

that calculates the value of a formula, given the environment that defines all variables.

Example:

```
Main> eval example (Var "x" `And` (Var "x" `orr` Var "y"))
True
```

Formulas can often be simplified enormously. For example, the formula (x || (y && false)) can be simplified to just x. This is because constant values, such as True and False, as arguments to logical operators, can always be simplified away.

(5p) **Question F**

Define a function:

```
simplify :: Form -> Form
```

that simplifies a given formula.

In the resulting formula, there should be no occurrences of **`(Bool _)`** left as arguments to **`And`** or **`Not`**. Also, all occurrences of **`Not (Not x)`** should have been simplified to **`x`**.

Example:

```
Main> simplify (Var "x" `orr` (Var "y" `And` Bool False))
Var "x"
```

(3p) **Question G**

Write two QuickCheck properties about **`simplify`**. (1) A property that states that the value of a simplified formula in a given environment is the same as the original formula. (2) A property that states that the result of simplification does not contain any occurrences of true or false anymore.

(4p) **Question H**

Define a function:

```
valid :: Form -> Bool
```

that checks if a given formula is valid or not. It should do this by first simplifying the formula, and then checking if the formula evaluates to true for all possible environments.

(1p) **Question I**

Define a QuickCheck property that states *soundness* of your theorem prover; if a formula is deemed valid by your function, it really evaluates to true for every possible environment.

Assignment 4 -- Background Knowledge (total 3p)

In this assignment, you have the chance to show us what background knowledge you have picked up during the course.

Please only answer *one* of the following questions. You can choose which one!

(3p) **Question A**

Discuss the difference between functions of type `A -> B` and functions of type `A -> IO B`. Are there things that you can do with one that you cannot do with the other? What is the point of separating these two kinds of functions? How does this difference influence the design of your program?

(3p) **Question B**

What are the main differences between the programming language Haskell and the programming language Erlang? What are the main similarities? You may discuss programming language "features" as well as the difference in purpose behind the two.

(3p) **Question C**

In a *sequential* or *imperative* programming language, a programmer expresses a sequence of instructions that should be carried out by the computer, one step at a time. Discuss what the disadvantage of this principle is in a parallel setting (where many things happen at the same time), for example when using a dual core processor. What can the advantage of functional programming be in this context?

Appendix – Standard Haskell Functions

This is a list of selected functions from the standard Haskell modules: Prelude, Data.List, Data.Maybe, Data.Char. You may use these in your solutions.

```
--------------------------------------------------------------------------
-- standard type classes

class Show a where
  show :: a -> String

class Eq a where
  (==), (/=) :: a -> a -> Bool

class (Eq a) => Ord a where
  (<), (<=), (>=), (>) :: a -> a -> Bool
  max, min             :: a -> a -> a

class (Eq a, Show a) => Num a where
  (+), (-), (*)    :: a -> a -> a
  negate           :: a -> a
  abs, signum      :: a -> a
  fromInteger      :: Integer -> a

class (Num a, Ord a) => Real a where
  toRational       ::  a -> Rational

class (Real a, Enum a) => Integral a where
  quot, rem        :: a -> a -> a
  div, mod         :: a -> a -> a
  toInteger        :: a -> Integer

class (Num a) => Fractional a where
  (/)              :: a -> a -> a
  fromRational     :: Rational -> a

class (Fractional a) => Floating a where
  exp, log, sqrt      :: a -> a
  sin, cos, tan       :: a -> a

class (Real a, Fractional a) => RealFrac a where
  truncate, round  :: (Integral b) => a -> b
  ceiling, floor   :: (Integral b) => a -> b

--------------------------------------------------------------------------
-- numerical functions

even, odd        :: (Integral a) => a -> Bool
even n           = n `rem` 2 == 0
odd              = not . even

--------------------------------------------------------------------------
-- monadic functions

sequence     :: Monad m => [m a] -> m [a]
sequence     = foldr mcons (return [])
                   where mcons p q = do x <- p; xs <- q; return (x:xs)

sequence_    :: Monad m => [m a] -> m ()
sequence_ xs = do sequence xs; return ()

--------------------------------------------------------------------------
-- functions on functions

id               :: a -> a
id x             = x

const            :: a -> b -> a
const x _        = x

(.)              :: (b -> c) -> (a -> b) -> a -> c
f . g            = \ x -> f (g x)
```

```
flip             :: (a -> b -> c) -> b -> a -> c
flip f x y       = f y x

($) :: (a -> b) -> a -> b
f $  x           = f x

--------------------------------------------------------------------------
-- functions on Bools

data Bool = False | True

(&&), (||)       :: Bool -> Bool -> Bool
True  && x       = x
False && _       = False
True  || _       = True
False || x       = x

not              :: Bool -> Bool
not True         = False
not False        = True

--------------------------------------------------------------------------
-- functions on Maybe

data Maybe a = Nothing | Just a

isJust                  :: Maybe a -> Bool
isJust (Just a)         =  True
isJust Nothing          =  False

isNothing               :: Maybe a -> Bool
isNothing               =  not . isJust

fromJust                :: Maybe a -> a
fromJust (Just a)       =  a

maybeToList             :: Maybe a -> [a]
maybeToList Nothing     =  []
maybeToList (Just a)    =  [a]

listToMaybe             :: [a] -> Maybe a
listToMaybe []          =  Nothing
listToMaybe (a:_)       =  Just a

--------------------------------------------------------------------------
-- functions on pairs

fst              :: (a,b) -> a
fst (x,y)        =  x

snd              :: (a,b) -> b
snd (x,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)

--------------------------------------------------------------------------
-- functions on lists

map :: (a -> b) -> [a] -> [b]
map f xs = [ f x | x <- xs ]

(++) :: [a] -> [a] -> [a]
xs ++ ys = foldr (:) ys xs

filter :: (a -> Bool) -> [a] -> [a]
filter p xs = [ x | x <- xs, p x ]

concat :: [[a]] -> [a]
concat xss = foldr (++) [] xss

concatMap :: (a -> [b]) -> [a] -> [b]
concatMap f = concat . map f
```

```
head, last       :: [a] -> a
head (x:_)       = x

last [x]         = x
last (_:xs)      = last xs

tail, init       :: [a] -> [a]
tail (_:xs)      = xs

init [x]         = []
init (x:xs)      = x : init xs

null             :: [a] -> Bool
null []          = True
null (_:_)       = False

length           :: [a] -> Int
length []        = 0
length (_:l)     = 1 + length l

(!!)             :: [a] -> Int -> a
(x:_)  !! 0      = x
(_:xs) !! n      = xs !! (n-1)

foldr            :: (a -> b -> b) -> b -> [a] -> b
foldr f z []     =  z
foldr f z (x:xs) =  f x (foldr f z xs)

foldl            :: (a -> b -> a) -> a -> [b] -> a
foldl f z []     =  z
foldl f z (x:xs) =  foldl f (f z x) xs

iterate          :: (a -> a) -> a -> [a]
iterate f x      =  x : iterate f (f x)

repeat           :: a -> [a]
repeat x         =  xs where xs = x:xs

replicate        :: Int -> a -> [a]
replicate n x    =  take n (repeat x)

cycle            :: [a] -> [a]
cycle []         =  error "Prelude.cycle: empty list"
cycle xs         =  xs' where xs' = xs ++ xs'

take, drop             :: Int -> [a] -> [a]
take n _      | n <= 0 =  []
take _ []              =  []
take n (x:xs)          =  x : take (n-1) xs

drop n xs     | n <= 0 =  xs
drop _ []              =  []
drop n (_:xs)          =  drop (n-1) xs

splitAt                :: Int -> [a] -> ([a],[a])
splitAt n xs           =  (take n xs, drop n xs)

takeWhile, dropWhile    :: (a -> Bool) -> [a] -> [a]
takeWhile p []          =  []
takeWhile p (x:xs)
            | p x       =  x : takeWhile p xs
            | otherwise =  []

dropWhile p []          =  []
dropWhile p xs@(x:xs')
            | p x       =  dropWhile p xs'
            | otherwise =  xs

lines, words     :: String -> [String]
-- lines "apa\nbepa\ncepa\n" == ["apa","bepa","cepa"]
-- words "apa  bepa\n cepa"  == ["apa","bepa","cepa"]

unlines, unwords :: [String] -> String
-- unlines ["apa","bepa","cepa"] == "apa\nbepa\ncepa"
-- unwords ["apa","bepa","cepa"] == "apa bepa cepa”

reverse          :: [a] -> [a]
```

```
reverse          =  foldl (flip (:)) []

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

elem, notElem    :: (Eq a) => a -> [a] -> Bool
elem x           =  any (== x)
notElem x        =  all (/= x)

lookup           :: (Eq a) => a -> [(a,b)] -> Maybe b
lookup key []    =  Nothing
lookup key ((x,y):xys)
    | key == x   =  Just y
    | otherwise  =  lookup key xys

sum, product     :: (Num a) => [a] -> a
sum              =  foldl (+) 0
product          =  foldl (*) 1

maximum, minimum :: (Ord a) => [a] -> a
maximum []       =  error "Prelude.maximum: empty list"
maximum xs       =  foldl1 max xs

minimum []       =  error "Prelude.minimum: empty list"
minimum xs       =  foldl1 min xs

zip              :: [a] -> [b] -> [(a,b)]
zip              =  zipWith (,)

zipWith          :: (a->b->c) -> [a]->[b]->[c]
zipWith z (a:as) (b:bs)
                 =  z a b : zipWith z as bs
zipWith _ _ _    =  []

unzip            :: [(a,b)] -> ([a],[b])
unzip            =  foldr (\(a,b) ~(as,bs) -> (a:as,b:bs)) ([],[])

nub              :: Eq a => [a] -> [a]
nub []           = []
nub (x:xs)       = x : nub [ y | y <- xs, y /= x ]

delete           :: Eq a => a -> [a] -> [a]
delete y []      = []
delete y (x:xs)  = if x == y then xs else x : delete y xs

(\\)             :: Eq a => [a] -> [a] -> [a]
(\\)             =  foldl (flip delete)

union            :: Eq a => [a] -> [a] -> [a]
union xs ys      = xs ++ (ys \\ xs)

intersect        :: Eq a => [a] -> [a] -> [a]
intersect xs ys  = [ x | x <- xs, x `elem` ys ]

intersperse      :: a -> [a] -> [a]
-- intersperse 0 [1,2,3,4] == [1,0,2,0,3,0,4]

transpose        :: [[a]] -> [[a]]
-- transpose [[1,2,3],[4,5,6]] == [[1,4],[2,5],[3,6]]

partition        :: (a -> Bool) -> [a] -> ([a],[a])
partition p xs   = (filter p xs, filter (not . p) xs)

group            :: Eq a => [a] -> [[a]]
-- group "aapaabbbeee" == ["aa","p","aa","bbb","eee"]

isPrefixOf, isSuffixOf   :: Eq a => [a] -> [a] -> Bool
isPrefixOf []     _      =  True
isPrefixOf _      []     =  False
isPrefixOf (x:xs) (y:ys) =  x == y && isPrefixOf xs ys

isSuffixOf x y           =  reverse x `isPrefixOf` reverse y
```

```
sort              :: (Ord a) => [a] -> [a]
sort              = foldr insert []

insert            :: (Ord a) => a -> [a] -> [a]
insert x []       = [x]
insert x (y:xs)   = if x <= y then x:y:xs else y:insert x xs

--------------------------------------------------------------------------
-- functions on Char

type String = [Char]

toUpper, toLower :: Char -> Char
-- toUpper 'a' == 'A'
-- toLower 'Z' == 'z'

digitToInt :: Char -> Int
-- digitToInt '8' == 8

intToDigit :: Int -> Char
-- intToDigit 3 == '3'

ord :: Char -> Int
chr :: Int  -> Char

--------------------------------------------------------------------------
```