Homework 5
SNU 4190.210 Fall 2011

Kwangkeun Yi

due: 11/02(Wed) 24:00

The purpose of this homework are

- practicing what we've learned about principles of programming
- making components for term-project

Exercise 1 "Prefix-free Variable-length Code"

Let's design a code for expressing sentence. Sentence is a sequence of finite words. For example, we can use only 4 words for a sentence; "ba", "na", "lize" and "tion". We can make various sentense such as "nabananalizenation", "lizebationalizebabanalize" or etc.

Let's make a rule for expressing these sentence by using 0 and 1. We can assign a code which consists of sequence of 0/1 for each words, then sentence is a list of that code. There are two ways to assign codes to the words.

1. Fixed-length encoding: Because we have only four words, we can assign two bits for each words: 00(ba), 01(na), 10(lize), and 11(tion). Then "nabananalizenation" is encoded into 01000101100111. Decoding is quite easy. We can split the code by 2bits and translate it to mapped word. This way needs 14 bits.

2. Variable-length encoding: When we use shorter bits for more frequently used words, we can reduce the code size.

   In the case of the sentence, "na" is the most frequent word. "ba", "lize" and "nation" have same frequency. We can assign variable length code: 0("na"), 11("ba"), 100("lize") and 101("tion"). Then "nabananalizenation" is encoded into 011001000101, 14bits. By varaible-length ecoding rather that fixed-length encoding, we can reduce 2bits.

   Decoding the variable-length encoding is also easy. we can decode the code to read the bit sequentially: when we read 0, the word is "na". Nothing ambigous. Second bit is 1 but there's no mapped word for that bit, so we read the bit once again. Third bit is 1, and composited bit is 11 that stands for "ba".

What is the reason that decoding the variable-length code is easy? That is because every encoded codes have "prefix-free" property. It is a limitation that every code used in the encoding should not have same prefix bit. We can distinguish each code by looking only a prefix in the code. This encoding method is what JPEG, MPEG and ZIP used.

The assignment is to make `vlencode` that creates variable-length codes which have "prefix-free" property. This function takes list of pair of word and frequency as inputs and returns list of pair of word and code. A word has string type, a frequency is integer, a code is 0/1 list.

This way is suggested by David Huffman, a student of MIT, in 1951. it was a project report of "Information Theory". Robert Fano was the professor of the class. He thought over optimized way of encoding with his colleague Claude Shannon. Fano made it as a challenge of his class project and his student Huffman solved this problem. As this result, Huffman stand high above his professor. He got A in the class of course and became famous. You can do as well as Huffman. The hint below is enough for doing assigment. Try to solve without googling.

- `vlencode` can be implemented by using binary tree. Every leafs of tree structure has the word value and every node has $\cdots$ *omission*$\cdots$. As a result, "prefix-free" codes are assigned each word. This is because only leafs can have the word.

  For tree structure, define functions below:

  $$\begin{aligned}
  &\texttt{leaf} : \mathit{string} \times \mathit{value} \to \mathit{tree}\\
  &\texttt{node} : \mathit{tree} \times \mathit{value} \times \mathit{tree} \to \mathit{tree}\\
  &\texttt{isleaf?} : \mathit{tree} \to \mathit{bool}\\
  &\texttt{leftsub} : \mathit{tree} \to \mathit{tree}\\
  &\texttt{righsub} : \mathit{tree} \to \mathit{tree}\\
  &\texttt{leafval} : \mathit{tree} \to \mathit{value}\\
  &\texttt{leafstr} : \mathit{tree} \to \mathit{string}\\
  &\texttt{rootval} : \mathit{tree} \to \mathit{value}
  \end{aligned}$$

- To make tree structure against frequency of words is a key for assigning optimized prefix-free code to word. Start making the tree structure with a rare word $\cdots$ *omission*$\cdots$ you can compose the tree like that way.□

Exercise 2 "Turing Machine: Value & Object"

Let's read the paragraph below and implement Turing Machine.

"Designing Turing Machine"
`ropas.snu.ac.kr/~kwang/paper/cs-ch1.pdf`

Submit the functions described below. You must submit two versions of the function: value-oriented (applicative) and object-oriented (imperative) way. The type of functions are not changed between two versions except `empty-ruletable`.

$$\texttt{empty-ruletable} : \textit{ruletable}$$

is the value-oriented case, and in the case of object-oriented version

$$\texttt{empty-ruletable} : \textit{void} \rightarrow \textit{ruletable}.$$

- Functions for making and using tapes:

  $$\begin{aligned}
  &\texttt{init-tape} : \textit{symbol list} \rightarrow \textit{tape}\\
  &\texttt{read-tape} : \textit{tape} \rightarrow \textit{symbol}\\
  &\texttt{write-tape} : \textit{tape} \times \textit{symbol} \rightarrow \textit{tape}\\
  &\texttt{move-tape-left} : \textit{tape} \rightarrow \textit{tape}\\
  &\texttt{move-tape-right} : \textit{tape} \rightarrow \textit{tape}\\
  &\texttt{print-tape} : \textit{tape} \rightarrow \textit{void}
  \end{aligned}$$

  symbol written on tape is a string. To move head for read and write to right/left, we use the function `move-tape-left`/`move-tape-right` respectively.

- Functions for making and using ruletables:

  $$\begin{aligned}
  &\texttt{empty-ruletable} : \textit{ruletable}\\
  &\texttt{add-rule} : \textit{rule} \times \textit{ruletable} \rightarrow \textit{ruletable}\\
  &\texttt{make-rule} : \textit{state} \times \textit{symbol} \times \textit{todo} \times \textit{move} \times \textit{state} \rightarrow \textit{rule}\\
  &\texttt{match-rule} : \textit{state} \times \textit{symbol} \times \textit{ruletable} \rightarrow \textit{todo} \times \textit{move} \times \textit{state}
  \end{aligned}$$

  state which represent Turing Machine's state is a string. *todo* is one of these value: `'erase`, pair of `'write` *symbol* made by cons. *move* is `'left`, `'right`, or `'stay`.

- Functions for Turing Machine:

  $$\begin{aligned}
  &\texttt{make-tm} : \textit{symbol list} \times \textit{state list} \times \textit{state} \times \textit{ruletable} \rightarrow \textit{tm}\\
  &\texttt{run-tm} : \textit{tm} \rightarrow \textit{tm}\\
  &\texttt{print-tm} : \textit{tm} \rightarrow \textit{void}
  \end{aligned}$$

  Function `make-tm` initialize Turing Machine using given symbols as input, locate head to first symbol on the tape, set up the machine state with given initial state and let Turing Machine have given ruletable.

  Function `run-tm` execute the given Turing Machine as input. It follows Turing Machine's ruletable. When the execution ends, print the contents of the tape and exit. □

Exercise 3 "SKI Solution Reactor"

Let's imagine "SKI" solution. SKI solution react itself in the normal temperature and as a result its components are changed. A solution doesn't change anymore after some time or react itself and change their component eternally. It depends on the initial combination of the solution.

A way to make SKI solution E is one of these five.

$$\begin{array}{rcll} E & \rightarrow & \texttt{S} \mid \texttt{K} \mid \texttt{I} & \\ & \mid & x & \text{variables} \\ & \mid & \texttt{(}E\ E\texttt{)} & \end{array}$$

An example of SKI solution is

$$\texttt{K}, \texttt{(I}\ x\texttt{)}, \texttt{(S ((K}\ x\texttt{)}\ y\texttt{))}, \texttt{(((S K) K)}\ x\texttt{)}$$

.

A rule of SKI solution is described below. When there are some matches in the solution, left side of the rules, it replace right side of the rules.

$$\begin{array}{rcl} \texttt{(I}\ E\texttt{)} & \rightarrow & E \\ \texttt{((K}\ E\texttt{)}\ E'\texttt{)} & \rightarrow & E \\ \texttt{(((S}\ E\texttt{)}\ E'\texttt{)}\ E''\texttt{)} & \rightarrow & \texttt{((}E\ E''\texttt{) (}E'\ E''\texttt{))} \end{array}$$

For example, SKI solution $\texttt{(((S K) I)}\ x\texttt{)}$ is react like following :

$$\texttt{(((S K) I)}\ x\texttt{)} \rightarrow \texttt{((K}\ x\texttt{) (I}\ x\texttt{))} \rightarrow x$$

Let's make a functions `react` that take SKI solution as an input and print final shape of the solution.

$$\texttt{react} : \textit{solution} \rightarrow \textit{void}$$

FYI, There can be several matches in the solutions. For example, $\texttt{((K}\ x\texttt{) (I}\ y\texttt{))}$ is possible to be changed into one of solution:

$$\texttt{((K}\ x\texttt{) (I}\ x\texttt{))} \rightarrow x$$

or

$$\texttt{((K}\ x\texttt{) (I}\ x\texttt{))} \rightarrow \texttt{((K}\ x\texttt{)}\ x\texttt{)} \rightarrow x.$$

If there are multiple ways to change, your `react` just choose one of them.

Implement these functions to make and use SKI solution:

`S` : *solution*	
`K` : *solution*	
`I` : *solution*	
`v` : *string* $\rightarrow$ *solution*	(* Make solution using variable name *)
`a` : *solution* $\times$ *solution* $\rightarrow$ *solution*	(* Compose two solution with parenthesis *)
`isS?` : *solution* $\rightarrow$ *bool*	
`isK?` : *solution* $\rightarrow$ *bool*	
`isI?` : *solution* $\rightarrow$ *bool*	
`isv?` : *solution* $\rightarrow$ *bool*	
`isa?` : *solution* $\rightarrow$ *bool*	
`var` : *solution* $\rightarrow$ *string*	(* get variable name of the solution made by v*)
`al` : *solution* $\rightarrow$ *solution*	(* left solution of the composed solution*)
`ar` : *solution* $\rightarrow$ *solution*	(* right solution of the composed solution*)
`pprint` : *solution* $\rightarrow$ *void*	(* pretty printer *)

The action of `pprint` is noticed by TA. □