Homework 4
SNU 4190.210 Fall 2012

Chung-Kil Hur

due: 10/26(Sun) 24:00

Exercise 1 "Huffman Code"

Let's try to devise a way to encode sentences, where a sentence is a finite sequence of words. For example, suppose we have four words: "I", "me", "you and "know". One can make many sentences using these words, such as "I know you" and "you know I know you know me".

We will represent such sentences using 0 and 1. First, we assign to each word binary code consisting of 0 and 1. Then, a sentence is simply encoded as the sequential composition of the code for each word in the sentence. There are two ways to do this.

1. Fixed-length encoding: Since there are four words, each word can be encoded using two bits: 00 for "I", 01 for "me", 10 for "you" and 11 for "know". Then, "you know I know you know me" is encoded as 10110011101101, which is 16 bit long. Decoding such code is easy: just decode each two bits one by one.

2. Variable-length encoding: If we know the frequency of each word, we can encode a sentence more efficiently by assigning short code for more frequent words.

   Suppose that "know" is the most frequent and "you" is the second. Then we can encode "know" as 0, "you" as 10, "I" as 110 and "me" as 111. This way, we can encode "you know I know you know me" as 1001100100111, which is 13 bit long.

   Decoding such code is also easy: just decode from left to right. For example, in 1001100100111, the only first possible code is 10, which is for

"you", and the ony second possible code is 0, which is for "know", and so on. We can easily see that 1001100100111 is uniquely decoded into "you know I know you know me".

Why is the decoding easy? The reason is because there is no word code that is a prefix of another word code. The word codes 0, 10, 1110 and 111 satisfy this property, which is called *prefix-free*. Such a variable-length encoding is used in data compression software such as JPEG, MPEG and ZIP.

This assignment is to write a function `vlencode` that finds the optimal prefix-free variable-length code. Given a list of pairs of a word and its frequency, `vlencode` should return a list of pairs of a word and its code. A word is a string, a frequency is an integer and code is a list of 0 and 1.

In 1951, David Huffman, as a graduate student at MIT, found an optimal solution for the variable-length encoding problem as a term project in the course "Information Theory". With this result, he outperformed Prof. Robert Fano, who taught the course and had worked on the same probelm with Claude Shannon, the inventor of information theory.

You can also do it as Huffman did. The following hints might be enough. Try to solve it by yourself before googling the solution.

- Use a binary tree to generate prefix-free code. Words are assigned to leaves of a binary tree, and each branch in the tree determines 0 or 1.

  Define and use the following functions to implement binary trees.

  `leaf` : $int \times string \to tree$
  `node` : $int \times tree \times tree \to tree$
  `isleaf?` : $tree \to bool$
  `leafval` : $tree \to int$
  `leafstr` : $tree \to string$
  `nodeval` : $tree \to int$
  `leftsub` : $tree \to tree$
  `rightsub` : $tree \to tree$

- Build a binary tree, taking into account the frequency of each word. For this, store the total frequency of words in a sub-tree in the root of the sub-tree.

□

Exercise 2 "SKI combinator calculus"

SKI expressions E are inductively defined as follows:

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

Examples are

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

The evaluation rules of SKI calculus is as follows.

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

For example, $\texttt{(((S K) I)}\ x\texttt{)}$ evaluates as follows:

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

Define a function `execute` that takes an SKI expression and prints out a sequence of SKI expressions that occur during the evaluation.

$$\texttt{execute} : E \to \mathit{void}$$

Note that there are several possible evaluations of the same expression. For example, $\texttt{((K}\ x\texttt{)}\ \texttt{(I}\ y\texttt{))}$ can evaluate as follows:

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

Or,

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

In such a case, `react` is allowed to follow just one of the possible executions.

Implement functions with the following interface as a library for SKI cal-

culus, and use the library to define `react`.

`S` : E	
`K` : E	
`I` : E	
`v` : *string* $\rightarrow E$	(* construct an SKI expression consisting of a variable *)
`a` : $E \times E \rightarrow E$	(* construct an SKI expression of the form (E_1 E_2) *)
`isS?` : $E \rightarrow$ *bool*	
`isK?` : $E \rightarrow$ *bool*	
`isI?` : $E \rightarrow$ *bool*	
`isv?` : $E \rightarrow$ *bool*	
`isa?` : $E \rightarrow$ *bool*	
`var` : $E \rightarrow$ *string*	(* When the argument is a variable, return its name *)
`al` : $E \rightarrow E$	(* When the argument is (E_1 E_2), return E_1 *)
`ar` : $E \rightarrow E$	(* When the argument is (E_1 E_2), return E_2 *)
`pprint` : $E \rightarrow$ *void*	(* pretty-print the given SKI expression *)

where `pprint` should print SKI expressions in the way that TA will tell you. □