CSE 1729	Introduction to Principles of Programming Problem Set 8	October 27, 2017

This problem set builds upon this week's lab, in that we start by getting the frequencies of symbols (words) in a list (text). From there, we use a heap to support finding the least-weighted values. Where this differs from the lab is that we will use the symbol frequencies and heap to build a Huffman tree to encode the symbols. You will then use this Huffman tree to be able to encode and decode texts. This looks fairly complex, but can be broken down into a number of manageable tasks.

For testing, you can use any string, but you should demonstrate all of the tasks using the text of Surfin' Bird, a combination of two Rivingtons' song recorded by the Trashmen in 1963. The Surfin' Bird text (encoded in a Scheme string) is available with this assignment in file "huffman-support.rkt".

The goal of this assignment is to be able to read in a text, turn it into a list of symbols, and use the frequencies of those symbols to develop a Huffman coding scheme. Once you have built the Huffman tree, you should be able to use it to encode inputs into strings of zeroes and ones, and decode strings of zeroes and ones into texts. Some code will be provided, in particular some of the code to deal with converting strings to lists of symbols, and vice-versa. The coding scheme that you will build is a prefix code that associates a string of zeroes and ones to each symbol in its language.

There is code supplied in file "huffman-support.rkt" that you can use. The main things you need are `(string->symbol-list str)` which turns a string into a list of symbols (including newlines), and `(symbol-list->string lst)` which turns a list of symbols into a string. These two functions are nearly inverses of each other. There is also a defined function of no arguments `(surfin-bird)`, which evaluates to a relatively long string that can be used for final testing. You will need to use some shorter strings when you are trying to debug your code.

Huffman trees are discussed in your textbook, but there are differences in how they represent the trees and how they represent coded texts (as lists rather than strings, as we do in this assignment). By all means read the Huffman tree section in the book, but be sure that your code matches the tasks given in this assignment rather than the code in the book.

1. Write a Scheme procedure `(word-frequencies str)` that will take a string and return a list of the words and their frequencies in the string. Look familiar? This is the same as the `freq-list` procedure from Lab 9 except its input will be a string. Hint: use the supplied `string->symbol-list` function to get something `freq-list` can act upon. This function will evaluate to a list of pairs; the first element of each pair will be a symbol corresponding to a word, the second element will be the number of times the word occurs in the string (which is the number of times the symbol occurs in the output of `string->symbol-list`.

   Here is an example:

```
> (word-frequencies  "da doo ron ron ron da doo ron ron")
((ron . 5) (doo . 2) (da . 2))
```

2. This question has you construct a Huffman encoding tree as specified below in parts (a) and (b). You might want to first look at **Background: Constructing Huffman trees** near the end of this document. You may assume that the symbols and their frequencies are given in a list of pairs: for example, the list `((ron . 57) (doo . 21) (da . 12))` represents the 3 symbols 'ron, 'doo, and 'da, with frequencies 57, 21, and 12, respectively. This is the form you should expect as

output from `word-frequencies` in the previous question. Given such a list, you wish to compute the tree that results from the algorithm described at the end of the document. Your Huffman trees should be binary trees, where each node has a value, a left subtree, and a right subtree.

In Huffman trees, the symbols are in leaf nodes that have the form

```
(s () ())
```

where `s` is the symbol held by the leaf. Internal nodes do not have meaningful values: I suggest that you use the form

```
('internal 0-tree 1-tree)
```

where `internal` is a symbol that indicates that this is an internal node, and `0-tree` and `1-tree` are the two subtrees. You could use the following code to construct an internal node:

```
(define (make-internal-node 0-tree 1-tree)
  (make-tree 'internal 0-tree 1-tree))
```

Note that when you traverse a Huffman coding tree, you can determine if a given node is an internal node by deciding if the `value` of the tree associated with that node is the token `internal`.

So how do I build this tree? Use a heap with pairs like in the lab, where the first element of the pair is a Huffman tree and the second element is its weight. Start out by making a tree (a leaf node) out of every character, make a pair of it and its frequency – these will be the initial pairs in the heap. You combine pairs by making a pair where the first element is a tree made of the trees from the two pairs; the weight of this new tree is the sum of the weights. So the process works like this: remove the two minimum-weight pairs from the heap; combine them into a new pair; insert that new pair into the heap. Continue until there is only one pair left in the heap – *the first element of that pair is your Huffman tree.* IMPORTANT: To get credit for this function you need to construct the Huffman tree using a heap as described above, not by using an ordered list as in the textbook.

(a) Write the Scheme function `(combine-htree-pairs hp1 hp2)` that takes two tree-weight pairs (the first element of each pair is a Huffman tree, the second element is its weight) and combines them into a new pair, where the first element of the pair is a Huffman tree whose root is 'internal, left subtree is the tree from `hp1`, and right subtree is the tree from `hp2`; the second element of this pair is the sum of the weights from pairs `hp1` and `hp2`. This will be a useful function in the process described above.

Example:

```
> (define hpair1 (cons (make-tree 'doo '() '()) 21))
> hpair1
((doo ()()) . 21)
> (define hpair2 (cons (make-tree 'da '() '()) 12))
> hpair2
((da ()()) . 12)
> (combine-htree-pairs hpair1 hpair2)
((internal (doo ()()) (da ()())) . 33)
```

(b) Write a Scheme function `(build-huffman sf-list)` which, given a list of symbols and frequencies, constructs a Huffman encoding tree.

Example:

```
> (build-huffman '((ron . 57) (doo . 21) (da . 12)))
(internal (internal (da () ()) (doo () ())) (ron () ()))
```

3. Define a Scheme function `get-encoding-list` that takes, as input, a Huffman coding tree and outputs a list containing the elements at the leaves of the tree along with their associated encodings as a string over the characters `#\0` and `#\1`. For example, given the tree of Figure 1, your function should return the list

   ```
   ((ron . "1") (da . "00") (doo. "01"))
   ```

   in some order.

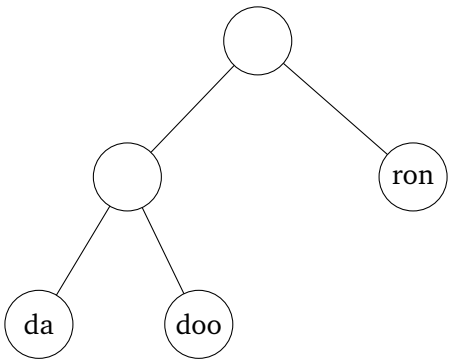

Figure 1: A Huffman tree yielding an encoding of the three symbols `da`, `doo`, and `ron`.; the codeword associated with `doo` is `"01"`.

   Hint: the easiest way to build the strings of zeroes and ones is to build them as lists of characters (using the characters `#\0` and `#\1`), then convert them to strings when you are ready to build each pair. See the "Strings and characters in Scheme" section at the end of the assignment for guidance.

4. Define a Scheme function `encode` which takes, as input, a text string and a list of character-encoding pairs (as in problem 3) and encodes the text string into a string of zeroes and ones using Huffman coding. (You may assume that every character in the string is indeed in the list of character-encoding pairs.)

   ```
   > (encode "doo doo doo da da da da"
             '((da . "00") (doo . "01") (ron . "1")))
   "01010100000000"
   ```

   This will involve building up a string by concatenating the code strings. Scheme provides a function for appending strings called `string-append`, which takes a number of strings and appends them together, for example:

```
>(string-append "abc" "def")
"abcdef"
```

5. Define a Scheme function `(decode str huff-tree)` which takes, as input, a string over the characters `#\0` and `#\1` and a Huffman coding tree and "decodes" the string according to the tree. (It should return a text string.) As with some of the functions above, it is probably easier to have an auxillary function that takes a list of zero and one characters and a Huffman tree and outputs a list of symbols, which is then converted to a string using the supplied `symbol-list->string` function.

6. Finally, demonstrate that your code works by running it on the Surfin' bird text:
   (a) Run `word-frequencies` on the Surfin' bird string to produce a list of word-frequency pairs.
   (b) Use `build-huffman` to produce a Huffman tree that encodes the words in the Surfin' bird string
   (c) Use `get-encoding-list` on the Surfin' bird Huffman tree to get a list of word-code pairs.
   (d) Use `encode` on the Surfin' bird text to get a 0/1 string that encodes it.
   (e) Use `decode` on the encoded Surfin' bird text to get from the 0/1 string back to the original string. Note: if you `(display decoded-text)` you should get something that matches the original including line breaks.

7. This question is for no points, only glory.

   Strings of zeroes and ones are not as compact as binary numbers; in binary numbers each 0 or 1 is a single bit, while the characters in a string need to have enough bits to represent at all of the alphabetic characters (upper and lower case), the digits 0-9, and others things like newline.

   It is possible to convert your 0-1 strings to binary numbers with a bit of effort: integers are represented as binary numbers. As a final embellishment of your code, write two functions:

   1. `(bit-encode text htree)` that takes a text and an encoding list and produces the binary number that encodes the text – this is an extension of `encode` in problem 4.
   2. `(bit-decode number henc)` that takes a number and a Huffman tree and produces the text that the number and the Huffman tree encodes. – this is an extension of `decode` in problem 5.

   How to do this bit-encoding is explained at the end of this writeup under **Lists and bit-vectors**; code to support these encodings and decodings is in huffman-support.rkt.

   If you do this, and want to know the number of bits that your number represents, take the floor of its base-2 logarithm.

Background: Constructing Huffman trees Recall the discussion from class on Huffman trees. In particular, to construct an optimal encoding tree for a family of symbols $\sigma_1, \ldots, \sigma_k$ with frequencies $f_1, \ldots, f_k$, carry out the following algorithm:

1. Place each symbol σ_i into its own tree; define the weight of this tree to be f_i.

2. If all symbols are in a single tree, return this tree (which yields the optimal code).

3. Otherwise, find the two current trees of minimal weight (breaking ties arbitrarily) and combine them into a new tree by introducing a new root node, and assigning the two light trees to be its subtrees. The weight of this new tree is defined to be the sum of the weights of the subtrees. Return to step 2.

As an example, consider Huffman encoding a long English text document:

- You would begin by computing the frequencies of each symbol in the document. This would produce a table, something like the one shown below.

Symbol	Frequency
the	2013
be	507
to	711
⋮	⋮

  Here the "frequency" is the number of times the symbol appeared in the document. (If you prefer, you could divide each of these numbers by the total length of the document; in that case, you could think of the frequencies as probabilities. This won't change the results of the Huffman code algorithm.)

- Following this, you can apply the Huffman code algorithm above: this will produce a "Huffman code tree." The purpose of this tree is to associate a "codeword" with each symbol. Specifically, the path from the root to a given symbol can be turned into a codeword by treating every step to a left child as a zero and every step to a right child as a one. In Figure 3 below, the symbol `da` would be associated with the codeword "00".

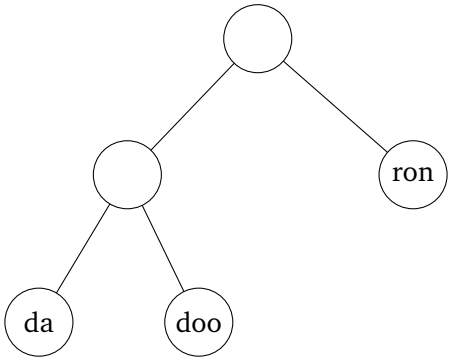

Figure 2: A Huffman tree yielding an encoding of the three symbols `da`, `doo`, and `ron`.; the codeword associated with `doo` is `"01"`.

- Finally, you can "encode" the document by associating each symbol in the document with the corresponding codeword. Note that this will turn the document into a long string of 0s and 1s.

- Likewise, you can "decode" the encoded version of a document, by reading the encoded version of the document from left to right, and following the path it describes in the Huffman tree. Every time a symbol is reached, the process starts again at the root of the tree.

Strings and characters in Scheme Scheme has the facility to work with strings and characters (a *string* is just a sequence of characters). In particular, Scheme treats *characters* as atomic objects that evaluate to themselves. They are denoted: `#\a`, `#\b`, ... Thus, for example,

```
> #\a
#\a
> #\A
#\A
> (eq? #\A #\a)
#f
> (eq? #\a #\a)
#t
```

The "space" character is denoted `#\space`. A "newline" (or carriage return) is denoted `#\newline`.

A *string* in Scheme is a sequence of characters, but the exact relationship between strings and characters requires an explanation. A string is denoted, for example, `"Hello!"`. You can build a string from characters by using the `string` command as shown below. An alternate method is to use the `list->string` command, which constructs a string from a list of characters, also modeled below. Likewise, you can "explode" a string into a list of characters by the command `string->list`:

```
> (string #\S #\c #\h #\e #\m #\e)
"Scheme"
> (list->string '(#\S #\c #\h #\e #\m #\e))
"Scheme"
> (string->list "Scheme")
(#\S #\c #\h #\e #\m #\e)
> "Scheme"
"Scheme"
```

You can also combine strings using `string-append`:

```
> (string-append "abc" "123")
"abc123"
> (string-append "do re mi" "a b c" "1 2 3")
"do re mia b c1 2 3"
```

Note that strings, like characters, numbers, and Boolean values, evaluate to themselves.

Lists and bit-vectors If we really wanted to save space, rather than coding into a string of zeroes and ones we would code into a bit-vector, which is a sequence of bits in contiguous memory, something that is supported in many languages. Scheme does not have a data type for bit-vectors, nor bitwise operators, but it is possible to use positive integers (which are represented as binary numbers) to implement bit sequences.

There are functions in huffman-support.rkt, available with the assignment, that allow us to represent strings of zeroes and ones as positive numbers whose sequence of bits matches the sequence in the list. There is a small hack required, as numbers ignore leading zeroes and we need to know how many there are.

In this file, the functions are `(string->bit-sequence str01)`, which takes a string made of zeroes and ones and turns it into a number corresponding to its binary encoding (less the first 1), and `(bit-sequence->string num)`, which takes a number and turns it into an string of zeroes and ones that correspond to its binary encoding (less the first 1).

These allow us convert back and forth between zero-one strings and numbers, so if you want a compact representation of an encoding you can apply `string->bit-sequence` to it and see how short the encoded version is, then apply `bit-sequence->string` to that to get back the encoding that your `decode` function above can handle.

You could try these out between steps (d) and (e) of problem 6. The length of the number (in bits) that you get from bit-length-encode is one more than the number of elements in the zero-one list. Or you may just want to play around with this code and see what it can (or cannot) do.