CSE1729 – Introduction to Programming April 1, 2022

Problem Set 9

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.

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"
```

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

1. Write a SCHEME function `(get-count text)` which, given a string `text` computes a list of character counts appearing in the text. Namely, the call `(get-count "hello")` returns the list
`((#\o . 1) (#\l . 2) (#\e . 1) (#\h . 1))`.
2. Write a SCHEME function which, given a list of character counts, returns a list of character frequencies. Namely, a call `(get-freq (get-count "hello"))` returns the list
`((#\o . 1/5) (#\l . 2/5) (#\e . 1/5) (#\h . 1/5))`.

3. Write a SCHEME function (`huffman freqs`) which, given a list of characters and frequencies, constructs and returns a Huffman encoding tree. You may assume that the characters and their frequencies are given in a list of pairs: for example, the list

```
((#\a . 1/5) (#\b . 1/7) (#\c . 1/9) (#\d . 172/315))
```

represents the 4 characters a, b, c, and d, with frequencies 1/5, 1/7, 1/9, and 172/315, respectively. Given such a list, you wish to compute the tree that results from the above algorithm. I suggest that you maintain nodes of the tree as lists: internal nodes can have the form

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

where `internal` is a token that indicates that this is an internal node, and `0-tree` and `1-tree` are the two subtrees; leaf nodes can have the form

```
('leaf weight symbol)
```

where `symbol` is the character held by the leaf. Note that you will have to use the SCHEME `quote` command to construct both types of nodes: for example, to construct an internal node with the two subtrees `0-tree` and `1-tree`, you could use the procedures below

```
(define (htree-leaf letter weight) (list 'leaf weight letter))
(define (htree-node t0 t1) (list 'internal (+ (htree-weight t0)
                                              (htree-weight t1)) t0 t1))
(define (htree-weight t) (cadr t))
```

Observe how the weight of an internal node is simply the sum of the weights of the sub-trees. For convenience, you will also see above a helper function (`htree-weight t`) meant to return the weight of a tree. Note how the weight is always the second value in the list describing a tree node (whether internal or a leave). When you traverse a Huffman coding tree, you can determine if a given node is an internal node by deciding if the `car` of the list associated with that node is the token `internal` (Similarly, you can check if a node is a leaf). The process for building a Huffman tree simply processes a work queue of weighted trees selecting the two *lightest* trees (smallest weights), combining them and placing the composite back into the work queue. The process ends when the work queue has a single tree (and that tree is the result of your function!). (**Hint:** since we are interested in light trees, a heap might come in handy!).

4. Define a SCHEME function (`codeWords t`) that takes, as input, a Huffman coding tree t and outputs a list of pairs $\langle symbol, code\$rangle$ containing the elements at the leaves of the tree ($symbol$) along with their associated encodings as a string over the characters `#\0` and `#\1` (the $code$). For example, given the tree of Figure 1, your function should return the list

```
((#\a . "0") (#\b . "10") (#\c . "10")) .
```

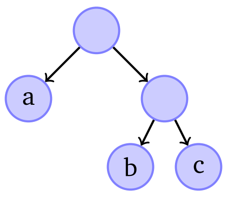

Figure 1: A Huffman tree yielding an encoding of the three symbols a, b, and c.