SICP section 2.3.4

September 12, 2007 at 06:47 **Tags** SICP

The solutions for this section are coded in Common Lisp.

Here is the code of the section before the first exercise, translated into CL:

```
(load "common")

(defun make-leaf (symbol weight)
  (list 'leaf sym weight))

(defun leaf? (obj)
  (eq (car obq) 'leaf))

(defun symbol-leaf (x)
  (cadr x))

(defun weight-leaf (x)
  (caddr x))

(defun make-code-tree (left right)
  (list
    left
    right
    (append (symbols left) (symbols right))
    (+ (weight left) (weight right))))

(defun left-branch (tree)
  (car tree))

(defun right-branch (tree)
  (cadr tree))

(defun symbols (tree)
  (if (leaf? tree)
    (list (symbol-leaf tree))
    (caddr tree)))

(defun weight (tree)
  (if (leaf? tree)
    (weight-leaf tree)
    (cadddr tree)))
```

```
(defun adjoin-set (x set)
  "Add a new element _x_ into a set of elements,
  sorted by weight"
  (cond ((null set) (list x))
        ((< (weight x) (weight (car set)))
          (cons x set))
        (t (cons  (car set)
                  (adjoin-set x (cdr set))))))

(defun make-leaf-set (pairs)
  (if (null pairs)
    '()
    (let ((pair (car pairs)))
      (adjoin-set (make-leaf
                    (car pair)      ; symbol
                    (cadr pair))    ; frequency
                  (make-leaf-set (cdr pairs))))))

(defun decode (bits tree)
  (labels (
    (decode-1 (bits branch)
      (if (null bits)
        '()
        (let ((next-branch
                (choose-branch (car bits) branch)))
            (if (leaf? next-branch)
              (cons (symbol-leaf next-branch)
                    (decode-1 (cdr bits) tree))
              (decode-1 (cdr bits) next-branch))))))

    (decode-1 bits tree)))

(defun choose-branch (bit branch)
  (cond ((= bit 0) (left-branch branch))
        ((= bit 1) (right-branch branch))
        (t (error "bad bit -- CHOOSE-BRANCH ~A" bit))))
```

I have a serious concern regarding the `decode` function. Its helper function `decode-1` recurses over the input list (*bits*), and not as a tail recursion (look at the positive branch of the inner `if` – consing a new decoded symbol to the decoding result of the rest of the input). Therefore, the stack depth of this function is proportional to the input size. Since input sizes may become quite large when decoding large files, it smells of a stack overflow.

I also wrote a convenience function for printing out Huffman trees. This makes debugging easier:

```
(defun print-tree (tree)
  (labels (
    (print-node (node offset)
      (dotimes (i offset) (format t " "))
      (if (leaf? node)
        (format t "~A ~A~%"
          (symbol-leaf node) (weight-leaf node))
        (progn
          (format t "~A ~A~%"
            (symbols node) (weight node))
          (print-node
            (left-branch node) (+ 2 offset))
          (print-node
            (right-branch node) (+ 2 offset))))))

    (print-node tree 0)))
```

Note that I'm using `format`. CL has a few basic printing functions with disgustingly unhelpful names: `prin1, princ, print, pprint`. I much prefer just using `format` even for the simple things instead of trying to remember what each of those does[1].

Exercise 2.67

```
(defvar sample-tree
  (make-code-tree
    (make-leaf 'A 4)
    (make-code-tree
      (make-leaf 'B 2)
      (make-code-tree
        (make-leaf 'D 1)
        (make-leaf 'C 2)))))

(defvar sample-message
  '(0 1 1 0 0 1 0 1 0 1 1 1 0))

(print (decode sample-message sample-tree))
=>
(A D A B B C A)
```

You can easily follow the decoding process manually – `decode` returns a correct result.

And re the stack overflow, sure enough, for inputs about 1000 bits long, `decode` begins overflowing the stack. This certainly is one of those places where iteration (or tail recursion) is much more appropriate. After all, there's no real state to keep between decoded characters. Therefore, decoding them in a loop is appropriate and readable.

Exercise 2.68

Here's `encode-symbol`:

```
(defun encode-symbol (sym tree)
  (labels (
    (tree-walk (sym node encoding)
      (if (leaf? node)
        encoding
        (cond
          ((element-of-set? sym (symbols (left-branch node)))
            (tree-walk sym (left-branch node) (cons 0 encoding)))
          ((element-of-set? sym (symbols (right-branch node)))
            (tree-walk sym (right-branch node) (cons 1 encoding)))
          (t (error "Symbol not in tree -- ~A" sym))))))

    (reverse (tree-walk sym tree '()))))
```

The internal function `tree-walk` recursively walks the encoding tree, looking for the full binary sequence that encodes the given symbol. Note that it builds the encoding by consing each new bit in front of the existing list, so its result must be reversed[2].

Note also that it uses the `element-of-set?` function (from the previous section). It is the unordered list version, since this is how sets of symbols are represented here.

And the ultimate test:

```
(decode
  (encode '(A D A B B C A) sample-tree)
  sample-tree))

=> '(A D A B B C A)
```

Exercise 2.69

I don't know why the authors say `successive-merge` is tricky. It appears pretty straightforward to me, just following the algorithm outlined in the subsection named "Generating Huffman trees"

```
(defun generate-huffman-tree (pairs)
  (successive-merge (make-leaf-set pairs)))

(defun successive-merge (node-set)
  (if (null (cadr node-set))
    (car node-set)
    (successive-merge
      (adjoin-set
        (make-code-tree (car node-set)
                        (cadr node-set))
        (cddr node-set)))))
```

At each step, `successive-merge` examines the set of nodes. If there's only one node in it `cadr` is empty, meaning "no second element") then this is the root of the tree and can be returned. Otherwise, it takes the first two nodes (that happen to be the nodes with the lowest weight, because `adjoin-set` keeps the list ordered) out of the set, makes a node that has them as its left and right branches, and puts the new node back, keeping the order. Eventually, only the tree root is left.

Here’s an example call with the same characters as the sample in the “Generating Huffman trees” section

```
(print-tree
  (generate-huffman-tree
    '((a 8 )
      (b 3)
      (c 1)
      (d 1)
      (e 1)
      (f 1)
      (g 1)
      (h 1))))

=>

(A H G F E D C B) 17
  A 8
  (H G F E D C B) 9
    (H G F E) 4
      (H G) 2
        H 1
        G 1
      (F E) 2
        F 1
        E 1
    (D C B) 5
      (D C) 2
        D 1
        C 1
      B 3
```

Indeed, the result is exactly as expected.

Exercise 2.70

```
(defvar rock-50s-tree
  (generate-huffman-tree
    '((a 2) (boom 1) (get 2) (job 2)
      (na 16) (sha 3) (yip 9) (wah 1))))

(print
  (encode
    '(get a job sha na na na na na na na na
      get a job sha na na na na na na na na
      wah yip yip yip yip yip yip yip yip yip
      sha boom)
    rock-50s-tree))

=>
(1 1 1 1 1 1 1 0 0 1 1 1 1 0 1 1
1 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1
0 0 1 1 1 1 0 1 1 1 0 0 0 0 0 0 0
0 0 1 1 0 1 0 1 0 1 0 1 0 1 0 1 0
1 0 1 0 1 0 1 0 1 1 1 0 1 1 0 1 1)
```

Altogether 84 bits. Had we used a fixed encoding we'd need 3 bits per symbol to encode the 8 different symbols. The length of the song is 36 symbols – so the fixed encoding would be `36 * 3 = 108` bits long.

Exercise 2.71

For `n=5`:

```
(print-tree
  (generate-huffman-tree
    '((a 1) (b 2) (c 4) (d 8 ) (e 16))))

=>
(A B C D E) 31
  (A B C D) 15
    (A B C) 7
      (A B) 3
        A 1
        B 2
      C 4
    D 8
  E 16
```

For `n=10`:

```
(print-tree
  (generate-huffman-tree
    '((a 1) (b 2) (c 4) (d 8 ) (e 16)
      (f 32) (g 64) (h 128) (i 256) (j 512))))

=>
(A B C D E F G H I J) 1023
  (A B C D E F G H I) 511
    (A B C D E F G H) 255
      (A B C D E F G) 127
        (A B C D E F) 63
          (A B C D E) 31
            (A B C D) 15
              (A B C) 7
                (A B) 3
                  A 1
                  B 2
                C 4
              D 8
            E 16
          F 32
        G 64
      H 128
    I 256
  J 512
```

As we can see, the result is a “linear” tree – each successive leaf is exactly one level deeper than the previous one. It is easy to see why – when the relative frequencies of symbols are successive powers of two, the sum of the weights of the two least frequent nodes is almost lower than the weight of the next node, so in the construction function, the tree is built “sequentially” deeper and deeper.

In such a tree, the most frequent symbol will need 1 bit of encoding, and the least frequent symbol will need `n-1` bits of encoding.

Exercise 2.72

In general, in the worst case we’d need to descend`n` levels (as Exercise 2.71 shows), and at each step we have to find the symbol in a set of symbols at that node. The implementation of `encode` used an unordered set to keep the symbols, so the search takes`O(n)` [3]. Therefore, the whole encoding procedure takes `O(n^2)`. Had we used a binary tree for the sets of symbols, we could bring this time down to `O(n^2)`. Of course, building the tree would then take longer.

---

[1] Except, perhaps the most trivial `(print obj)` when debugging, since it is shorter to write than the corresponding `format` call.

[2] This is a common pattern in list processing – consing in front takes only a constant time, and in the end you `reverse` once, whereas appending each new element to the end of the list will result in quadratic runtime since the appending operation takes time proportional to the length of the list.

[3] You may be tempted to say that the set to search grows smaller at each step, and that’s true.

However, the average set size for the worst tree is `n/2` which is still `O(n)`.

For comments, please send me an email.

Back to top