List processing in Scheme

Scheme vs. LISP

LISP suffered from a number of defects in its early versions, and then a number of incompatible versions arose which fixed the defects in different ways. Moreover, the language was extended to make it more practical as a general-purpose programming language. It became large and unwieldy. Gerald Sussman, an others at MIT, decided to make LISP revert to its origins as a simple implementation of the lambda calculus. This language was called Scheme. It is an exceptionally pure version of LISP, with a lot of the defects fixed. One early defect of LISP was its dynamic scoping. This was changed in Scheme to static scoping, making its referential transparency complete.

List Processing

Without data structures, LISP is a just an arithmetic engine. The idea of using the list as a general purpose data structure came about because a) it can serve to represent a wide variety of purposes: trees, arrays, stacks etc., and b) it is simple to implement. As an abstract data type, the list has four main operations:

- head – return the first item in the list
- tail – return the rest of the list except the first item
- cons – make a new list element and attach to an existing list
- null? – test whether a list if empty

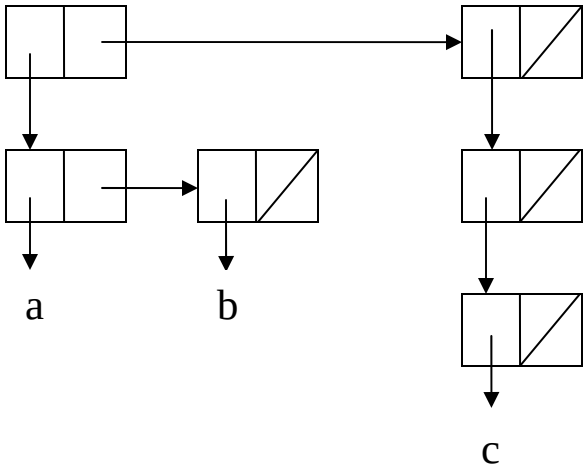
The remaining question is how to represent lists themselves in the syntax of the language. The answer that the LISP people came up with was to represent lists using the *same* syntax as expressions. Thus (a b c) could either be the application of a function a to its arguments b and c, or it could be a list of three *symbols*, a, b and c. Thus the syntax of a list is:

list = symbol | ‘(‘ (list)* ‘)’

This means that lists can be nested:

((a b) ((c)))

The internal representation of a list can be drawn in a box-and-arrow diagram. Each element of a list is a cell of two parts. Each part is a pointer to a list. The leaves of the tree are symbols, and the end of a list is a null pointer. The list above is drawn:



This can be seen to be a binary tree as well.

List quoting

Since the list syntax is the same as the expression syntax, we need a way to distinguish them. To stop evaluation of an expression as an application of a function, we *quote* the list:

‘(a b c)

Is thus a list – eval will simply return it unchanged. Symbols can also be quoted:

‘a

Notice the single quote mark – since a list is always parenthesized, like an expression, we only need one.

The quote mark is actually shorthand for the pseudo-function ‘quote’

‘(a b c) and (quote (a b c))

mean the same thing.

List Processing operations

The head of a list is just the first element of the list:

(head ‘(a b c)) = a
(head ‘((a b) ((c)))) = (a b)
(head ‘a) = error
(head ()) = error

The tail of a list is everything except the first element:

(tail ‘(a b c)) = (b c)
(tail ‘((a b) ((c)))) = (((c)))
(tail ‘a) = error
(tail ())= error

Notice the error conditions. A list is defined recursively; the base case (a symbol) will be an invalid input for head and tail, as will the empty list, written ().

The cons operation creates a new cell:

(cons ‘a ‘(b c)) = (a b c)
(cons ‘(a b) ‘(((c)))) = ((a b) ((c)))
(cons ‘a ()) = (a)

(cons () ‘(a b c)) = (() a b c)
(cons ‘a ‘ b) = (a . b)

Notice the last one. If the second argument is a symbol, a ‘dotted pair’ is created which is a cell with two pointers both pointing to symbols.

The null? operation returns true if its argument is an empty list.

(null? ()) = #t
(null? ‘(a b c)) = #f
(null? ‘a) = #f

Notice the empty list syntax, (), which does not need quoting, and the two special symbols #t and #f which are the representations of true and false.

A simple list processing function: length

Using these ideas we can write a function that returns the length of a list:

```
(define length
   (lambda (L)
      (if (null? L)
         0
         (+ 1 (length (tail L))))))
```

Notice the defining form in Scheme: define which binds a name (here length) to a value. The value of length is the function (lambda (L) …). A derivation of this function is:

```
(length ‘(a b c))
= (if (null? ‘(a b c)) 0 (+ 1 (length (tail ‘(a b c)))))
= (+ 1 (length ‘(b c)))
== (+ 1 (if (null? ‘(b c)) 0 (+ 1 (length (tail ‘(b c))))))
= (+ 1 (+ 1 (length ‘(c))))
= (+ 1 (+ 1 (if (null? ‘(c)) 0 (+ 1 (length (tail ‘(c)))))))
= (+ 1 (+ 1 (+1 (length ()))))
= (+ 1 (+ 1 (+ 1 (if (null? ()) 0 (+ 1 (length (tail ())))))))
= (+ 1 (+ 1 (+1 0)))
= 3
```

The expansion in each step comes directly from the function definition is obtained by substituting the value of the argument in the body.