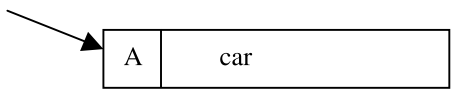
Quick Guide to Lisp Implementation

Representation of basic data structures

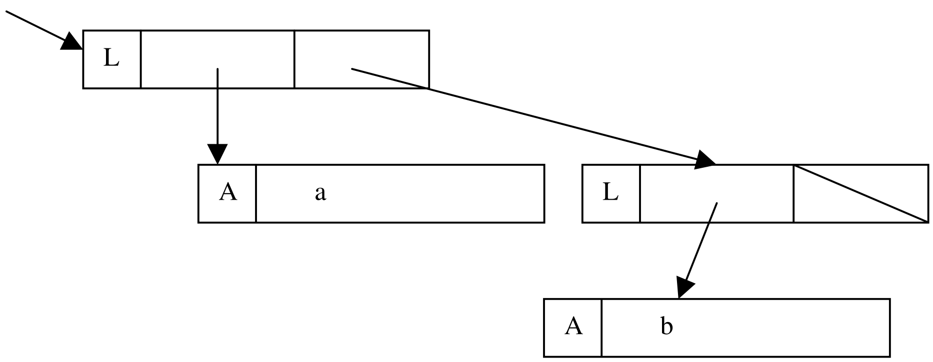
Lisp data structures are called **S-expressions**. The representation of an S-expression can be broken into two pieces, the tag (which stores type information) and the actual data. There are two kinds of S-expression, atoms and lists. The s-expression car (the atom car) would be stored as:

With the 'A' tag signifying that this is an atom (as opposed to a list), and the symbol car being the actual atom (in most lisp systems, the string "car" would be stored somewhere else, and the atom would just be a pointer to that location, but to keep things simple we will assume that the string can be stored in the data section of the lisp data structure)

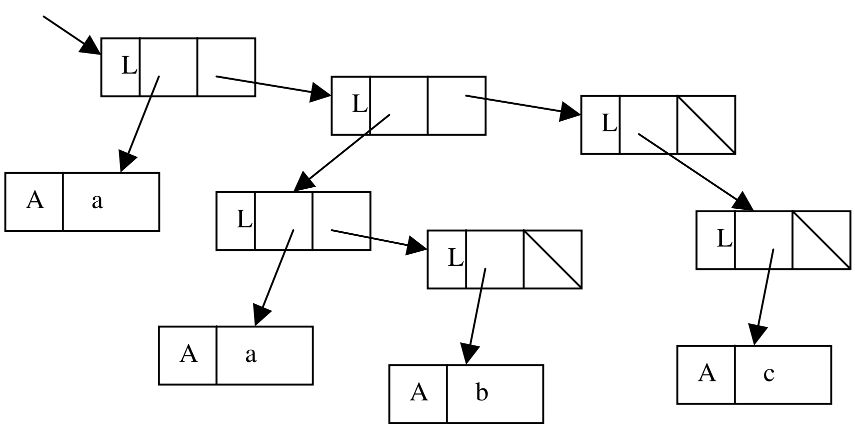
Since lisp stores everything as pointers, the atom car would really be a pointer to the above data structure.



Lists are built up using a list structure that has two pointers – a pointer to the first element of the list, and a pointer to the rest of the list. So, the list (a b) would be represented by the following:



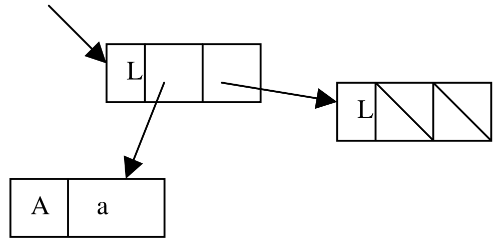
Let's get a little more complicated, and look at lists inside of lists. For example, the representation of the s-expression (a (a b) (c)) is



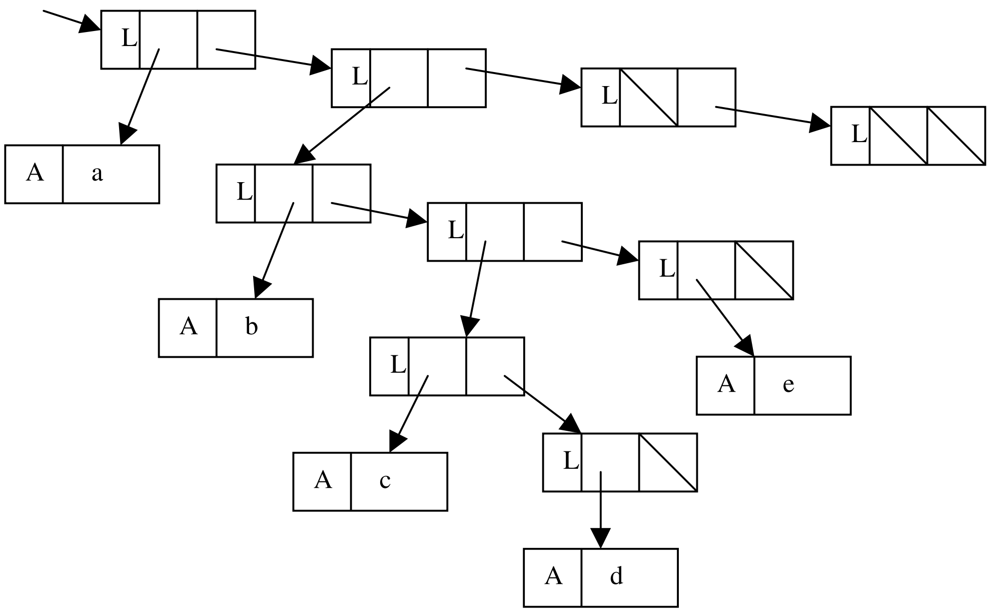
List within Lists within Lists

Here are even more examples of lists that contain lists, including the empty list:

(a ())



(a (b (c d) e) ( ) ( ))

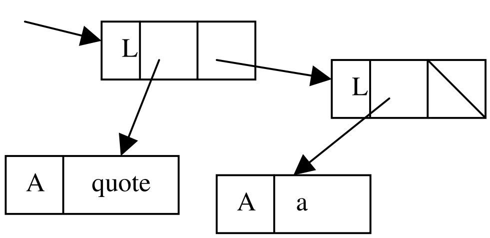


Our Friend quote

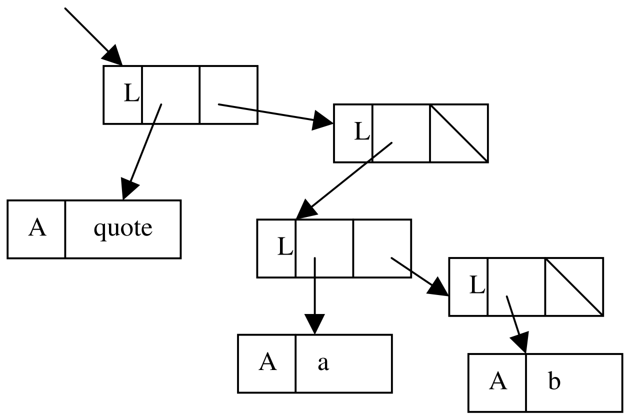
Recall that '<sexpr> is just a shorthand for (quote <sexpr>), so that

'a is shorthand for (quote a) and '(a b) is shorthand for (quote (a b))

Hence, the internal representation of 'a is the internal representation of (quote a), which is:



Likewise, the internal representation of '(a b) is the same as the internal representation of (quote (a b)), which is:

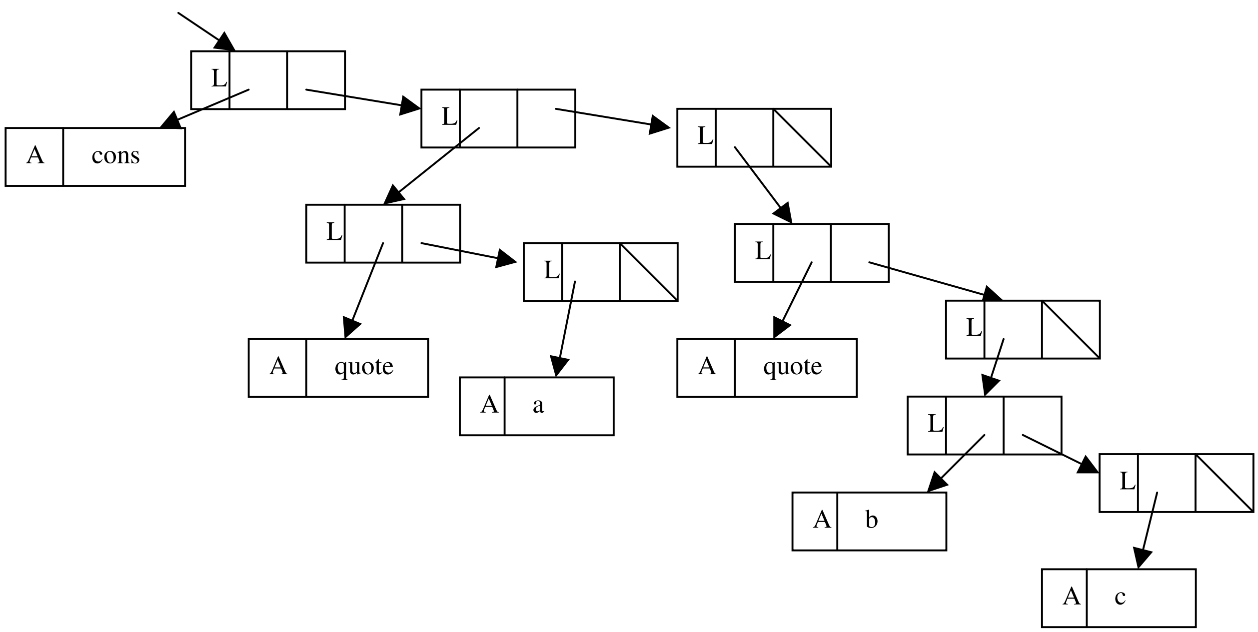


Read-Eval-Print loop

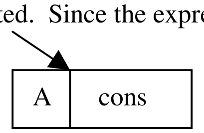
The lisp interpreter is elegant in its simplicity. An s-expression is read in (and converted to the internal representation), evaluated, and the result is printed out. So, it the user typed in:

(cons 'a '(a b))

First, the interpreter would convert this s-expression to the appropriate internal representation [remember that (cons 'a '(a b)) is shorthand for (cons (quote a) (quote (b c)))] :

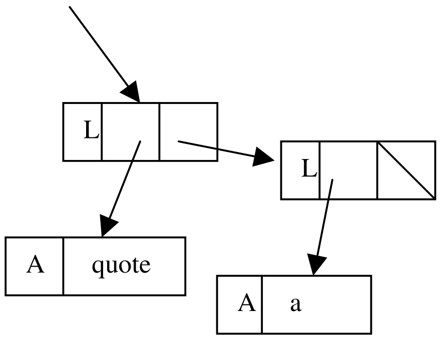


Next, the expression is evaluated. Since the expression is a list, the interpreter first looks at the first element of the list (which is

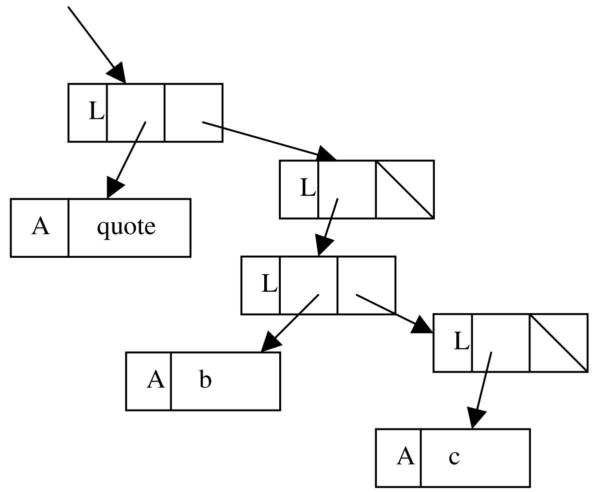


) to see if it is a valid function name.

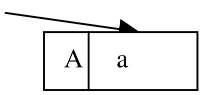
Since cons is a valid function name, each of the arguments to cons are evaluated. The 2 arguments to cons are the 2nd and 3rd elements of the list, which are:



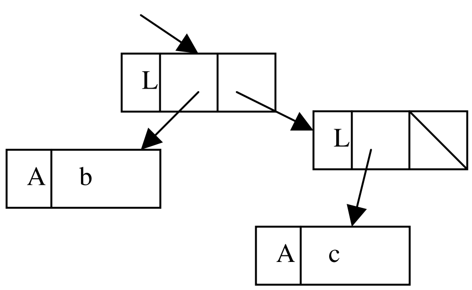
and



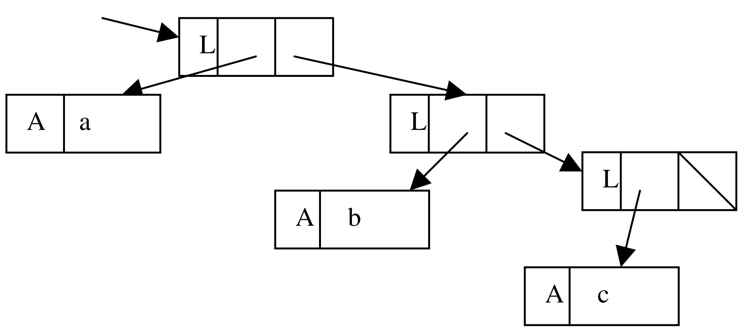
Taking the first argument first, this is a list. which means it is a function call. Check the first element of the list to make sure that it is a valid function name – and quote is a valid single argument function name. We are calling quote with a single argument, so we are OK so far. Quote is a special function that returns its argument unevaluated. So, this call to quote will return its one argument (the second element of the list) which is:



OK, so now we evaluate the second argument to cons, which is a list, which means a function call. First, check the first element of the list (the atom quote), which is a valid function name. The quote function returns its single argument (the second element in the list) unevaluated, to get:



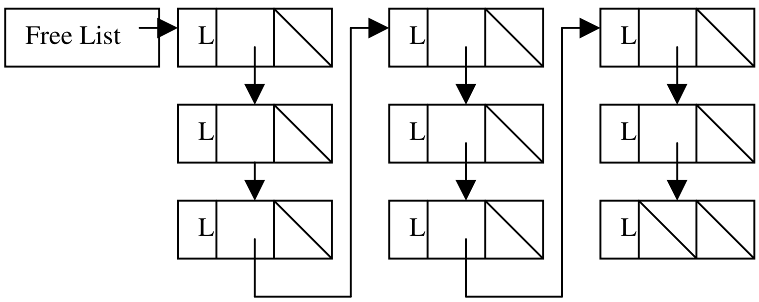
So, we can call cons, with the above two arguments. Cons creates a new list element, sets the car equal to the first element and the cdr equal to the second element, to get:



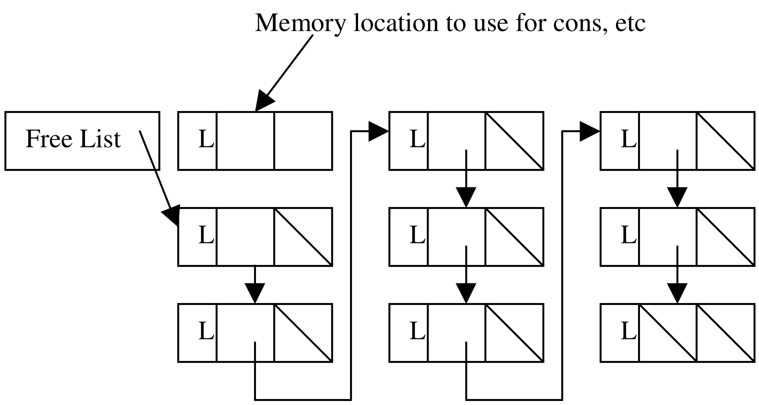
Finally, the interpreter prints out the result, which is (a b c).

Getting new memory – the free list

How do we get a new memory location when we need one? When cons is called, where does that memory location come from? All available memory is stored in a free list. The free list is a list of memory locations that are available. This list is a little different than a standard lisp list, like (a b c). To store the list (a b c), we need space for the three atoms a, b, and c, as well as 3 list constructs to glue everything together. It seems a waste to use 6 memory locations to store 3 memory locations for a free list, so we will use a slightly different method. The free list pointer will point to the first element of the free list, and the car of that element will point to the next element in the free list, and so on. So, the free list might look something like:



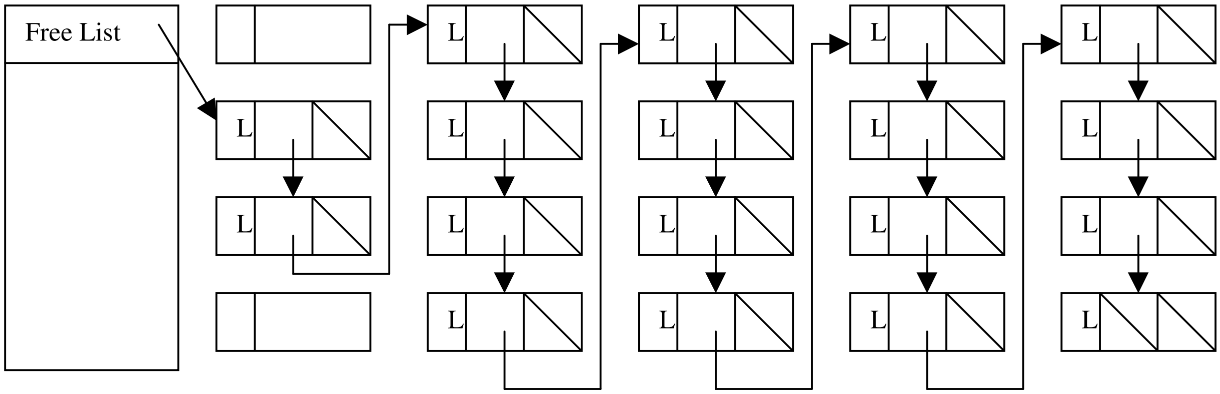
Any time a new piece of memory is needed, the first element of the free list is used, and the free list pointer is advanced.



Of course, eventually we are going to run out of memory. What happens then? The interpreter will do mark and sweep garbage collection. Every piece of memory that is currently being used is marked. Then the interpreter does a sweep through memory, adding every unmarked element to the free list. Then the free list should no longer be empty, and the interpreter will continue. Let's examine a complete example. Consider the function :

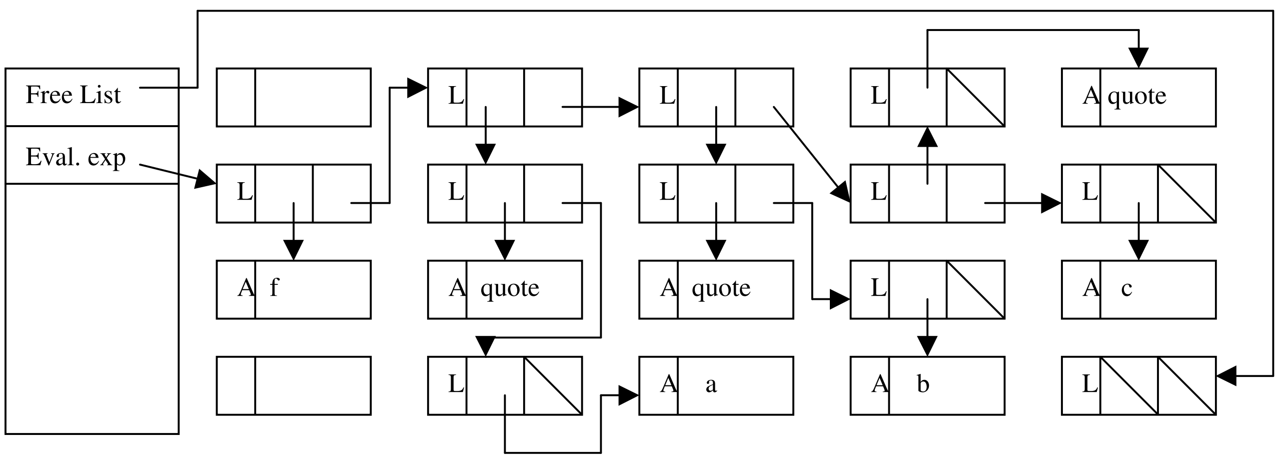
```
(defun f (x y z)
      (cons x (cons y (cons z ( )))))
```

Let's say the free list looks like the following:

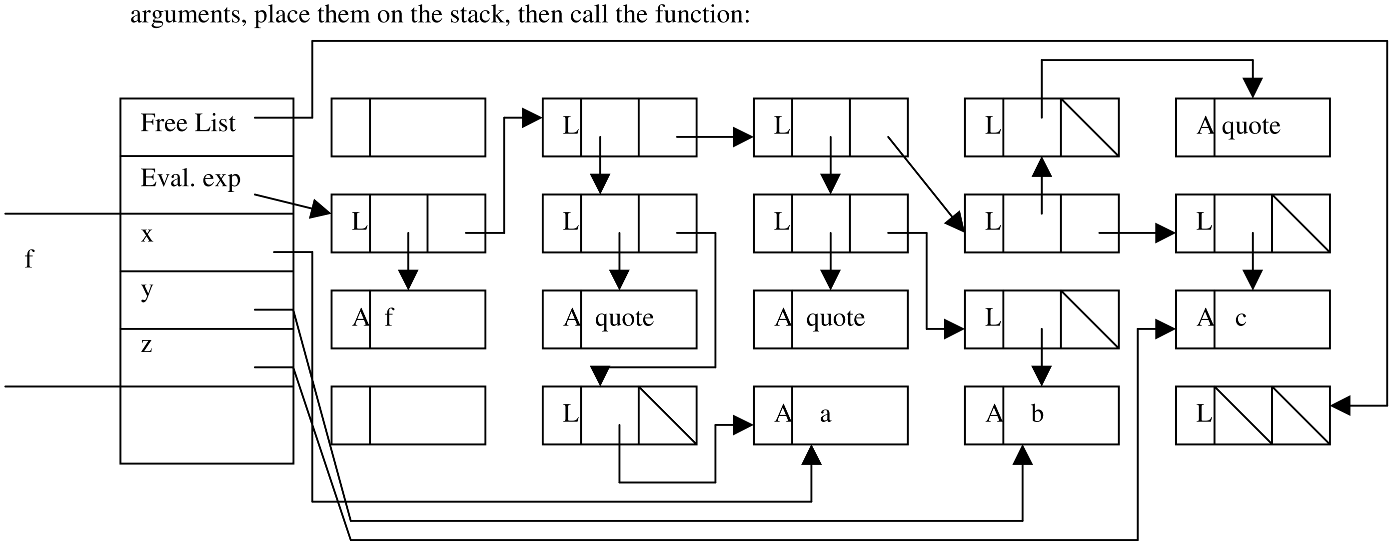


and we make the function call (f 'a 'b 'c) [which is shorthand for (f (quote a) (quote b) (quote c))]

The call (f 'a 'b 'c) is first read into memory:

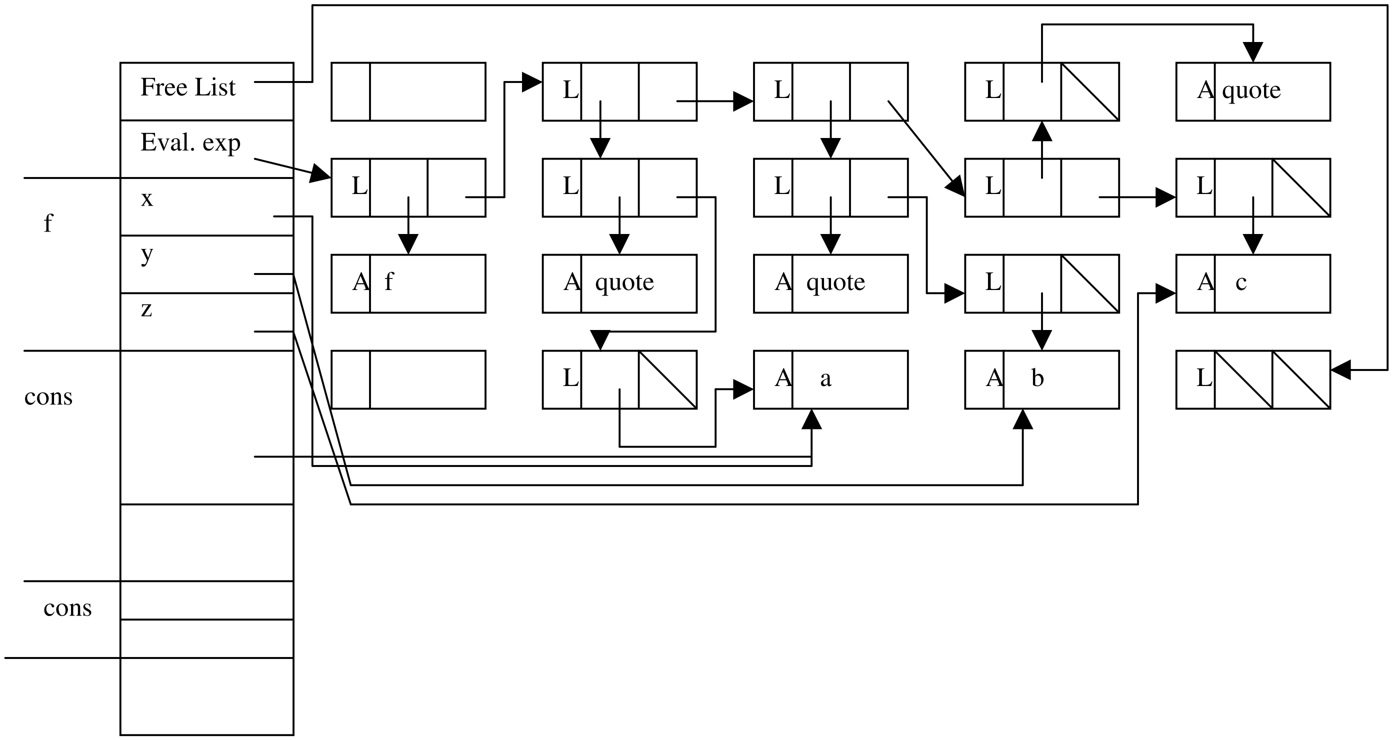


Now we have to evaluate the expression. "f" is a function that has been defined, so we evaluate each of the arguments, place them on the stack, then call the function:



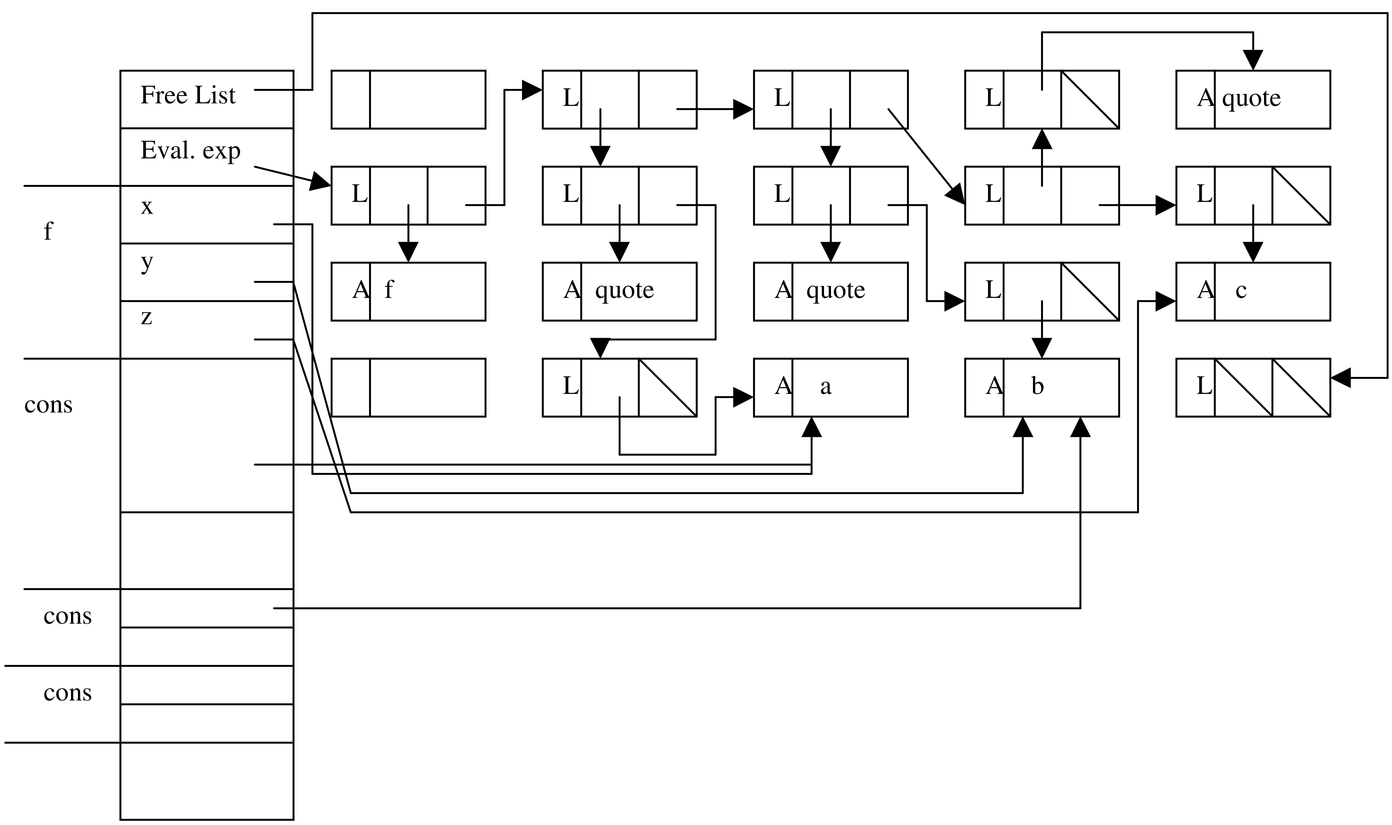
The first argument is (quote a), which evaluates to a, the second argument is (quote b), which evaluates to b, and the third argument is (quote c), which evaluates to c

Now we call the function f. That function returns a cons of two arguments:

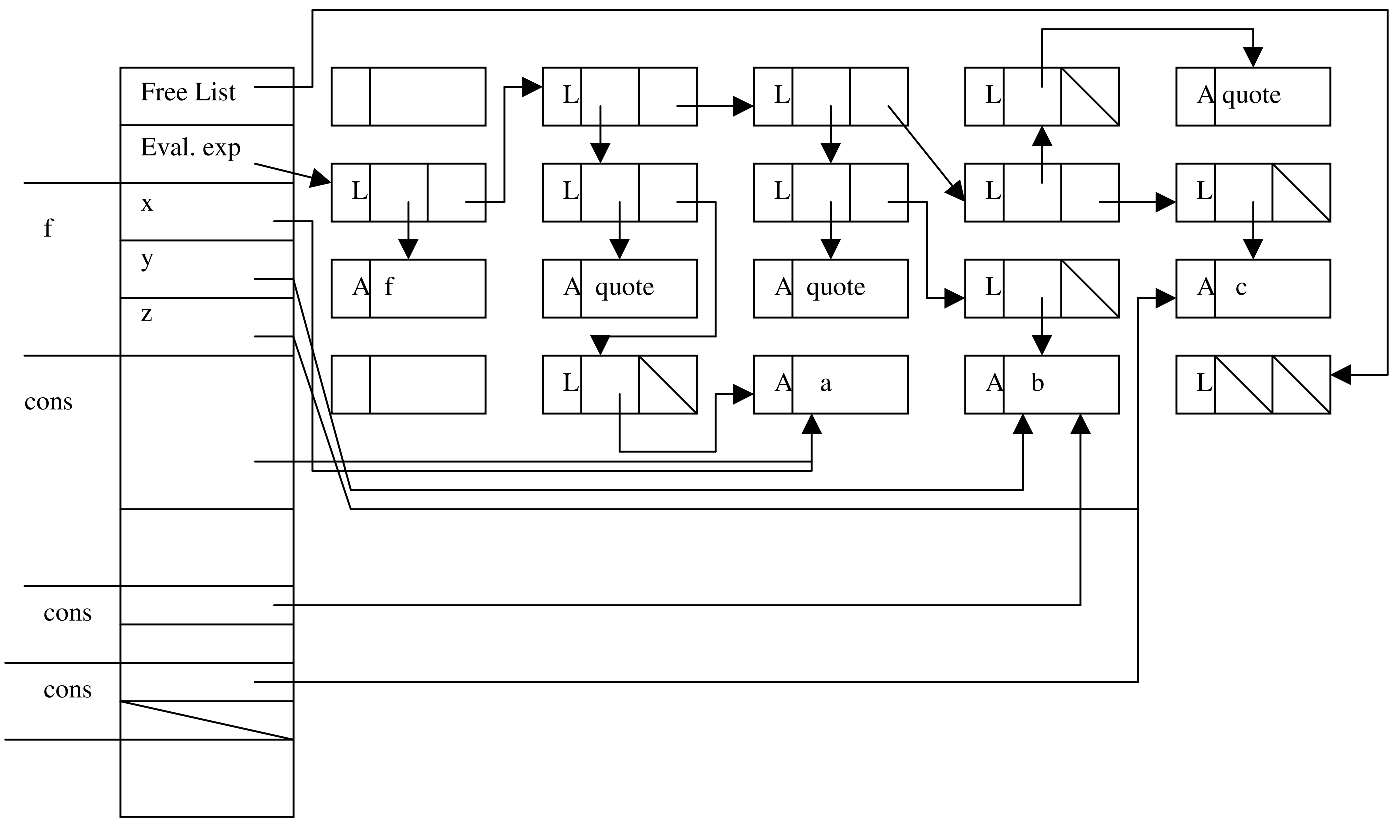


The first argument is x, but the second argument is another cons, which needs to be evaluated (the rather large graphical space allocated for the first cons on the stack is merely to make the pointers easier to see)

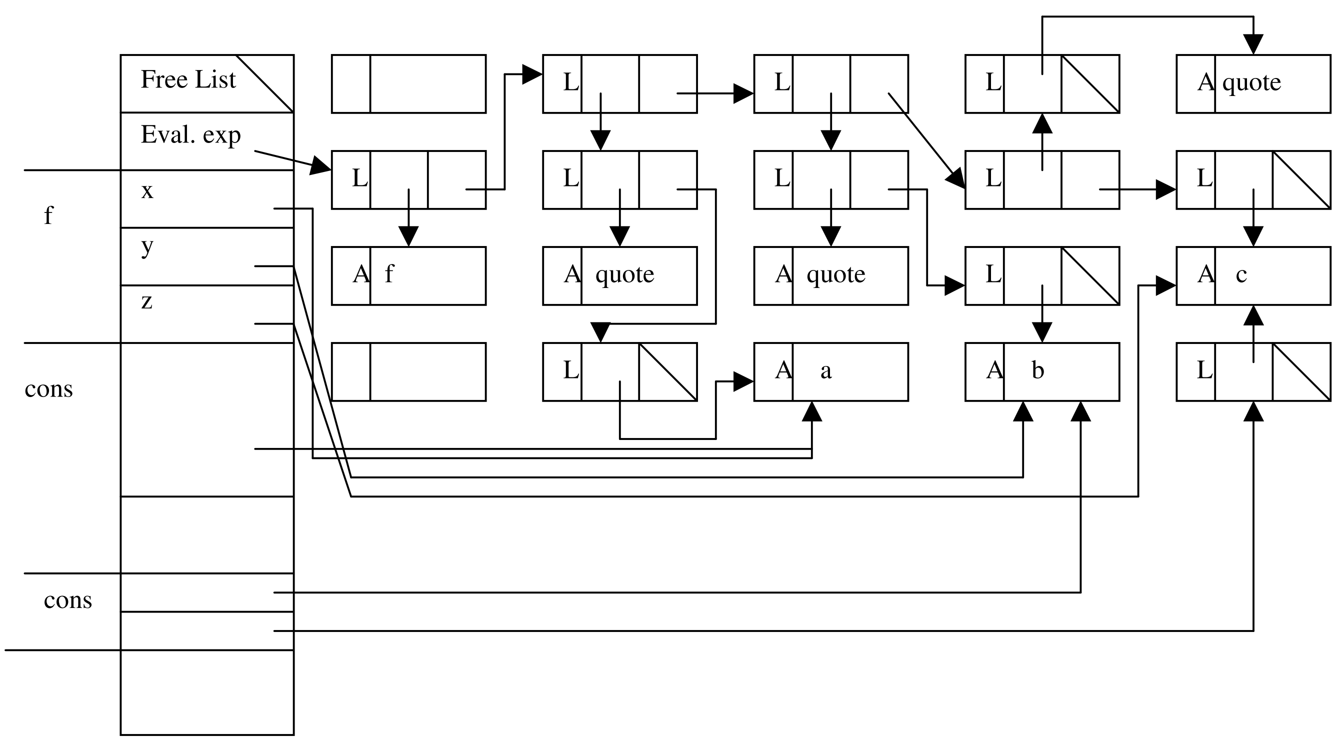
The first argument to *this* cons is y, but the second argument is another cons



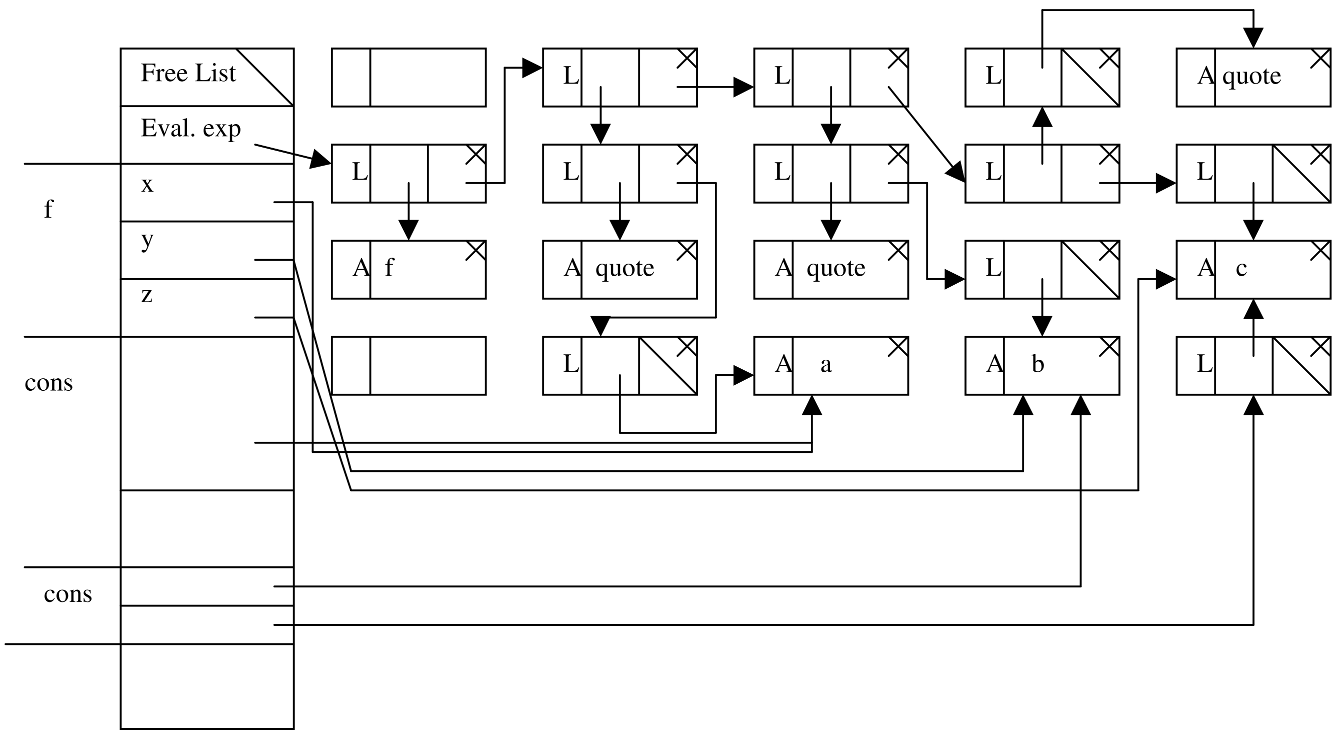
The arguments to this cons are z and () :



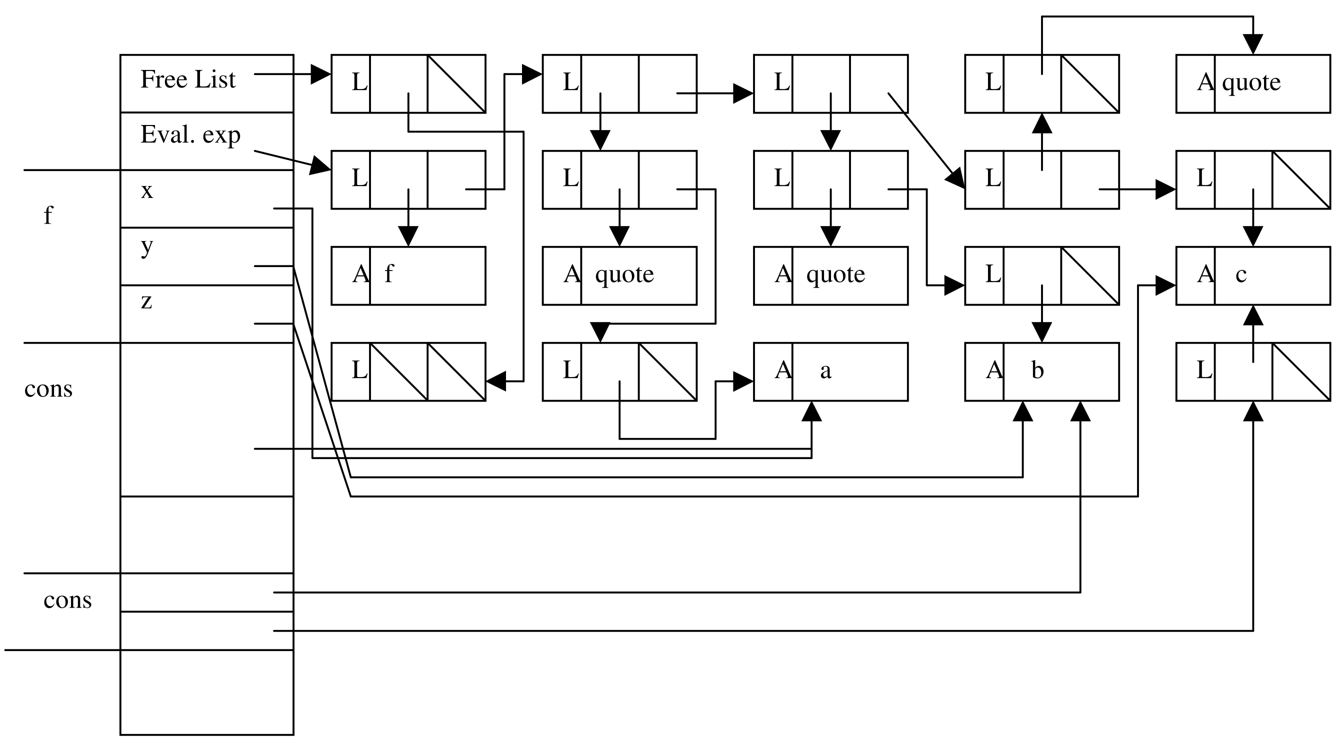
Now that we have evaluated the arguments for the final cons, we can execute the function body. Cons takes a new piece of memory from the free list, copies the first argument into the car and the second argument into the cdr, and returns this block of memory, which is used as the second argument to the second cons:



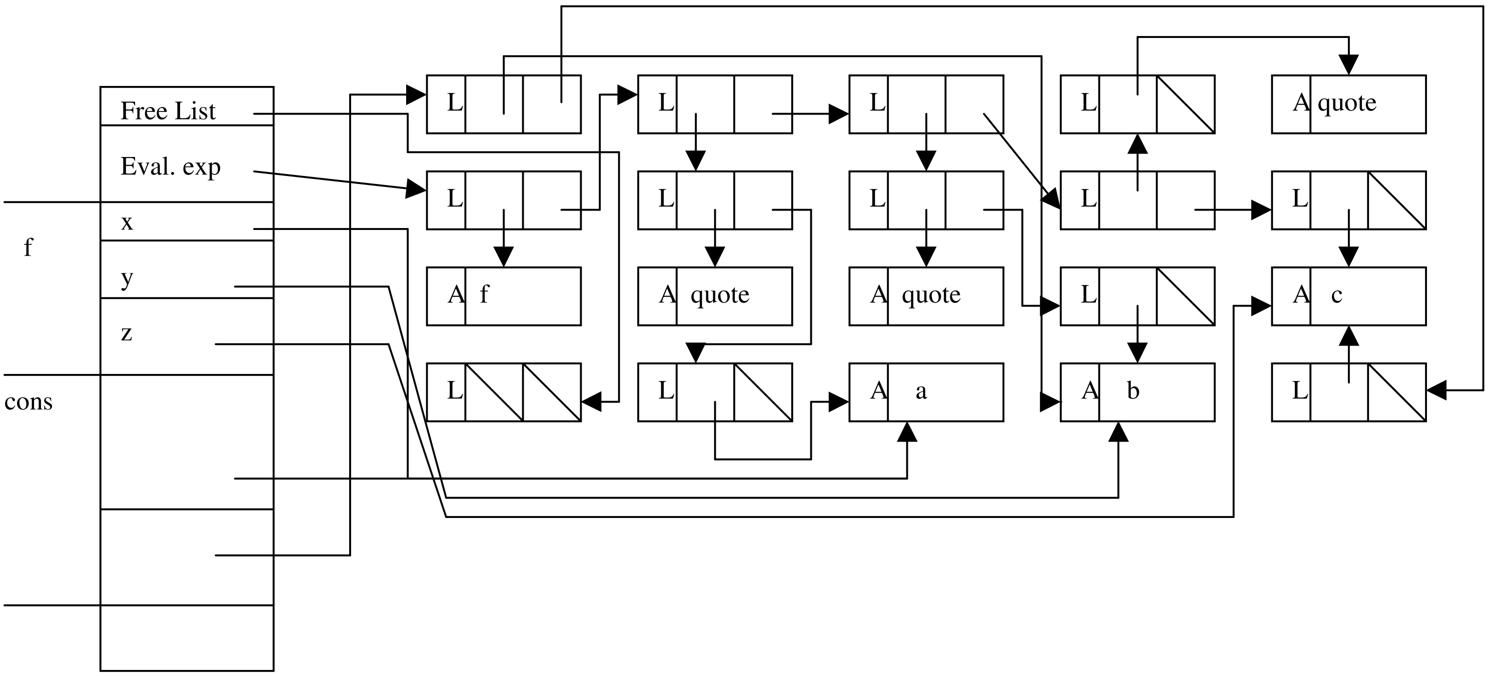
Now that we have evaluated both arguments for the second cons, we can execute the cons function body. Cons takes a new memory location off the free list – but the free list is empty! So, we need to do a garbage collection step. First we go through the stack and mark every piece of memory that is reachable from some pointer on the stack, as follows:



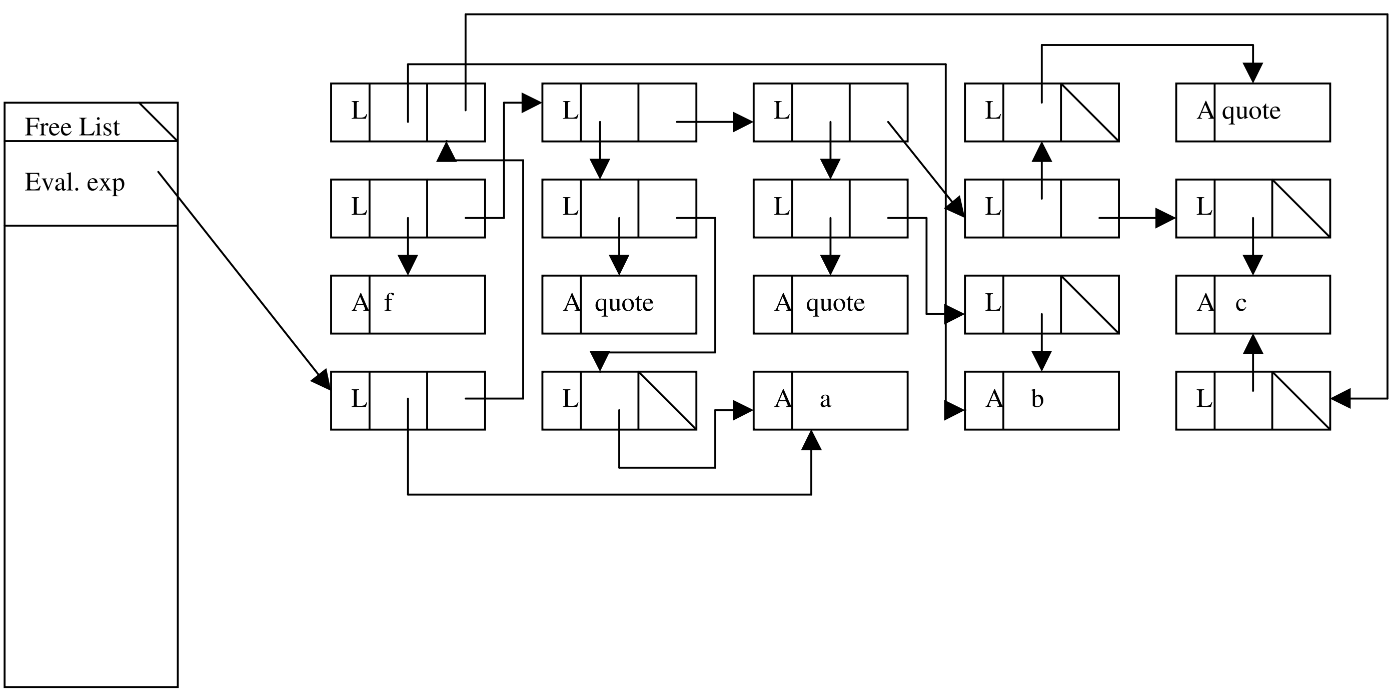
Of course, each memory location will need to have an extra bit reserved to store these marks. Now, we sweep through memory, adding everything that is not marked to the free list (we might as well also clear the marks on this pass, to be ready for the next time we want to do garbage collection).



Now the free list is no longer empty, so we can finish the call to the second cons:



Finally, we can complete the call to the first cons, and return that value (whew!)



Now we can print out the value of the expression, which is (a b c), as expected. Note that now the free list is again empty, and there is a whole lot of garbage. The next time we want to evaluate an expression, we will need to run mark-and-sweep garbage collection again, to free up some more memory.