Lists in Computer Memory

We explain how lists can be implemented in computer memory, showing that car (first), cdr (rest), and cons are constant time operations, that is, take time $O(1)$. We also describe the box-and-pointer representation of lists, including the ideas of cons cell, pointer, and shared substructure.

Implementing lists

Suppose we execute the command:

```
(define x (list 17 14))
```

Then the value of `x` is the Racket list `'(17 14)`. How could this list be represented in computer memory, for example, in the TC-201 computer? We would like to use the 16-bit integers of the TC-201 to represent different things: an integer, or a pointer, or a null list. To do this, we add type information, and represent each possible value as two consecutive memory locations, the first one giving the type of the value (integer, pointer, null list) and the second one giving the contents of the value (the value of an integer, the 12-bit address for a pointer, or 0 for a null list.) We arbitrarily choose type = 0 for an integer, type = 1 for a pointer, and type = 2 for a null list.

A cons cell will be represented by four consecutive memory locations, representing two typed values: the car part of the cell in the first two memory locations, and the cdr part of the cell in the second two memory locations. For example, we might have the following representation of the list `'(17 14)`:

```
address:   contents    interpretation

92:        0           type is integer
93:        17          value of integer is 17
94:        1           type is pointer
95:        100         pointer is to address 100


100:       0           type is integer
101:       14          value of integer is 14
102:       2           type is null list
103:       0           (this value doesn't matter)

204:       1           type is pointer
205:       92          pointer is to address 92
```

Here the value of `x` is contained in the two memory locations 204, 205, which give a pointer to the list `'(17 14)`.

The cons cell at 92 indicates that the first element of the list is the integer 17, and the next cons cell is at address 100. The cons cell at 100 indicates that the second element of the list is the integer 14, and this is the last cons cell in the list (the rest of the list is the empty list, indicated by the 2 in 102.)

Constant time operations on lists

With this representation of lists, the basic operations of `cons`, `car` (or `first`), and `cdr` (or `rest`) can each be implemented in time $O(1)$, that is, constant time independent of the size of the lists involved. To see this, note that the value of `x` is the pointer at address 204. To take the `car` of `x`, we get the contents of addresses 92 and 93, which are the integer value 17. To take the `cdr` of `x`, we get the contents of addresses 94 and 95, which are a pointer to address 100, that is, a pointer to the list `'(14)`. To `cons` the number 13 onto the list `x`, we allocate four consecutive cells from free memory for a new cons cell, say memory locations 220 through 223. We then place the integer 13 in the car of the cons cell (by setting 220 to 0 and 221 to 13) and the pointer to `x` in the cdr of the cons cell (by setting 222 to 1 and 223 to 92.) The value returned would be a pointer to the address 220, that is, a pointer to the list `'(13 17 14)`. The result would be the following memory contents.

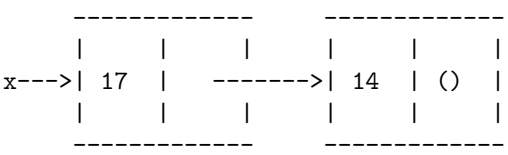
```
address:   contents    interpretation

92:        0           type is integer
93:        17          value of integer is 17
94:        1           type is pointer
95:        100         pointer is to address 100

100:       0           type is integer
101:       14          value of integer is 14
102:       2           type is null list
103:       0           (this value doesn't matter)

204:       1           type is pointer
205:       92          pointer is to address 92

220:       0           type is integer
221:       13          value of integer is 13
222:       1           type is pointer
223:       92          pointer is to address 92
```

Note that each of these operations involved allocating, copying, or changing a bounded number of memory locations, independent of the length of the list. Thus, each operation (cons, car, cdr) can be implemented in constant time. Two more constant time operations are null?, which tests whether a value is the null list (type 2 in our implementation) and pair?, which tests whether a value is a pointer to a cons cell (type 1 in our implementation).

Box and pointer diagrams

Abstracting away from the specific addresses, and representing each cons cell by two side-by-side boxes and each pointer by an arrow, we can give a box-and-pointer representation of the fact that the value of `x` is the list `'(17 14)`.

A box and pointer diagram showing that the value of `x` is `'(17 14)`.

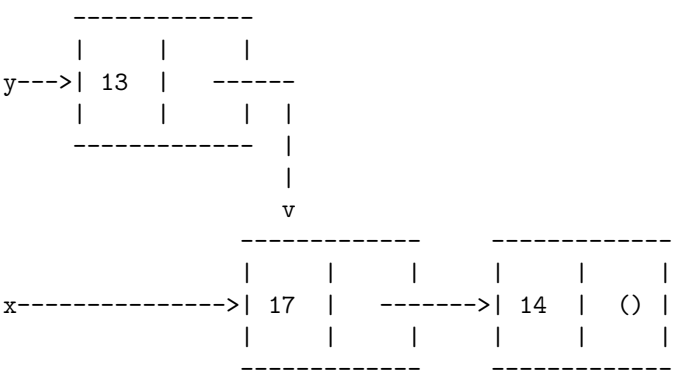


Here the value of `x` is a pointer to the first of two cons cells, each consisting of a car (the left side) and a cdr (the right side.) To find the car of the list `x`, we take the car part of the first cons cell to get the value 17. To find the cdr of the list `x`, we take the cdr part of the first cons cell, which points to the second cons cell. To find the next element of the list `x`, we take the car part of the second cons cell to get the value 14. The cdr part of the second cons cell is the null list, so there are no more elements in the list `x`.

If we now execute

```
(define y (cons 13 x))
```

then the `cons` creates a new cons cell, and sets its car to 13 and its cdr to the value of `x`, resulting in the structure:



Note that the list that is the value of `x` is a substructure of the list that is the value of `y`. This really only becomes relevant if you can mutate the values of cons cells, which `set-car!` and `set-cdr!` would allow you to do.