MC28 Lab 4: Printing list structure

Due: October 30, 1995

In this lab, you will fight fire with fire: you will use the power of mutable data structures to build a tool to help you cope with the complexity of mutable data structures.

As you have seen, one of the difficulties of working with mutable data structures is that Scheme's printed representation of list structure (i.e., pairs) is not very helpful with such issues as sharing and circularity. What would be nice is a tool which would print out a value essentially as its box-and-pointer diagram. You will write exactly such a tool. For example, you could use it to help with the problem I (Karl) did in class on Wednesday as follows:

```
(define x (list '(a b) '(c d)))

(define y (cons x (cdr x)))

(set-car! (car y) (cdr y))

y
;Value: ((((c d)) (c d)) (c d))

x
;Value: (((c d)) (c d))

(print-list-structure y)
Value: (1)
1: (2) (3)
2: (3) (3)
3: (4) ()
4: c (5)
5: d ()
;No value

(print-list-structure x)
Value: (2)
;No value
```

Here the parenthesized numbers correspond to pointers to pairs, and the pairs themselves are listed as an identifying number followed by the `car` and `cdr` (which are either non-pairs such as `c` or `()` or are themselves pointers, i.e. parenthesized numbers). For example, the above printout corresponds with the box-and-pointer diagram shown in Figure 1, where I have indicated the numbers above the boxes to make the correspondence more clear. The dotted arrow indicates the value of `(car x)` before the procedure call `(set-car! (car y) (cdr y))`.

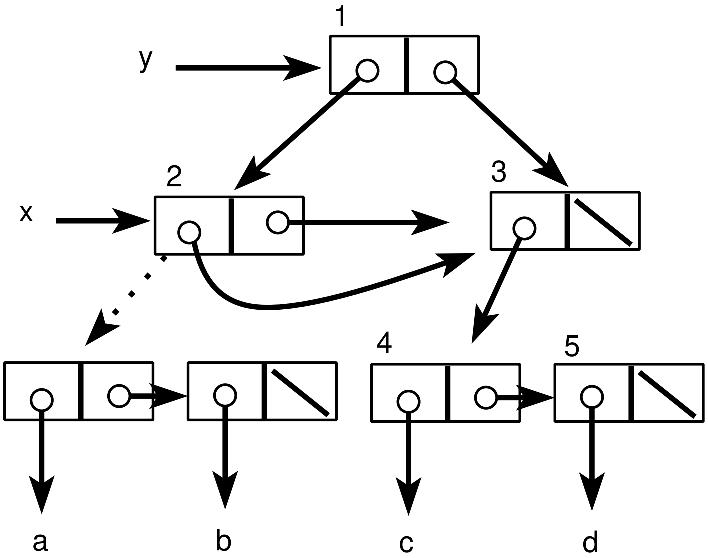

Figure 1: Box-and-pointer diagram for x and y

In order to write this `print-list-structure` procedure, you'll need two mutable data structures:

1. You will need to keep track of which pairs you have already encountered (and the identifying numbers you have assigned them), so that if you encounter the same pair again you can refer to it by the same number.

2. You will need to keep track of which pairs you have referred to but not yet printed, so that you can print each one out in turn. That is, you will need to keep an "agenda" of pairs to print. Each pair will get added onto this agenda when first referenced, and will get removed from it when it is printed out (referring to its `car` and `cdr` in turn).

Luckily, the two mutable abstract data structures we have recently encountered, the table and the queue, are ideally suited for these applications. Therefore, you can design `print-list-structure` as a procedure with local state, where the local state variables include a table (mapping pairs to their identifying numbers) and a queue (of those pairs waiting to be printed). You will also want to keep a counter, so that you can assign a newly encountered pair the next available identifying number.

There are many ways you could structure this procedure. However, to get you started, here are some useful subprocedures you may want to include (note that these subprocedures must be within the scope of the local state):

- `(pair-id` *p*`)` returns the integer identifier used for the given pair, *p*. If none has yet been assigned, it assigns one, records it in the table, and enqueues the pair for eventual printing.

- `(print-value` *v*`)` prints out the value *v* (either for the opening line of the printout, or also for the `car` or `cdr` of an individual pair). No newline is provided. If v is an non-pair, it is printed itself, while if it is a pair, a one-element list containing just its pair-id is printed instead. (The actual printing can be done with `write`.) To tell whether something is a pair, you can use the pre-defined predicate `pair?`.

- `(print-pair` p`)` prints out the pair p for the table constituting the main body of the printout. It does a `(newline)`, then prints out p's identifier, a colon and space, the car of p, a space, and the cdr. The colon and spaces can be printed out using `display`.

- `(process-queue)` prints out each pair in the queue in turn.

For many of you, this is the first time you have had to actually write output code; if you have any problems with it, be sure to ask for help (it isn't the main issue).

You can get the book's implementation of queues and tables from the file `/Net/mcs-server/usr/local/scheme/examples/chapter3.scm`. (From now on, you'll be expected to know that this directory contains all the code from the book; I won't be excerpting stuff for you.) However, that code makes use of the name `nil`, on the false presumption that it is predefined as a name for the empty list. So, you'll need to do `(define nil '())`.

One final feature you should probably include is that if the user does `(print-list-structure 'reset!)`, the procedure's state is reset back to the way it started.

In testing your procedure, you should not only try the example shown above, but also several other examples from the book or homework. For example, you might replicate figure 3.19 from page 210, display your answers to exercise 3.16 (page 206), or do exercise 3.15 on page 206.

If you have some extra time, you might want to change the order in which the pairs are printed. The current order, using a queue, is what is called a *breadth-first* directed graph traversal. That is, the original pair's children are both printed out before any of the grandchildren, and all grandchildren are printed out before any great-grandchildren, etc. Many times this is harder to understand than if the pairs were printed in *depth-first* order. That is, all the descendants of the car are printed before starting on the cdr. One way to achieve this is to simply replace the queue with a stack. So, if you have time, implement a stack abstract data type and then modify `print-list-structure` to use it.

Another shortcoming of `print-list-structure` you might want to address is that, although it handles nicely the sharing and circularity that come with mutation, it doesn't actually handle the mutation itself, in the sense of redisplaying a pair if it has changed. You might want to consider what it would take to fix this.