Notes courtesy of Justin Chen

CS61A Notes – Week 5b: Vectors, Metacircular evaluator

Just When You Were Getting Used to Lists…

Finally we are now introducing to you what many of you already know – arrays. Roughly, an array is a contiguous block of memory – and this is why you can have "instantaneous", random access into the array, instead of having to traverse down the many pointers of a list.

Recall the vector operators:

`(vector [element1] [element2] ...)` => works just like `(list [element1] ...)`
`(make-vector [num])` => creates a vector of `num` elements, all unbound
`(make-vector [num] [init-value])` => creates a vector of `num` elements, all set to `init-value`
`(vector-ref v i) => v[i]`; gets the `i`th element of the vector `v`
`(vector-set! v i val) => v[i] = val`; sets the `i`th element of the vector `v` to `val`
`(vector-length v)` => returns the length of the vector `v`

Beyond using different operators, there are a few big differences between vectors and lists:

Vectors of length N
- a contiguous block of memory cells
- O(1) for accessing any item in the vector
- O(N) for adding an item to the middle of the vector, since you have to move the rest of the vector down
- O(N) for growing a vector; you have to *reallocate* a new, larger block of memory!
- add 1 to index to get next element
- you may have "unbound" elements in the vector; that is, length of vector is not the same as length of valid data

Lists of length N
- many units of two cells linked together by pointers
- O(N) for accessing an item
- O(1) for inserting an item anywhere in the list, assuming we have a pointer to the location
- O(1) for growing a list; just add it at the beginning or the end (if you have a pointer to the end)
- `cdr` down a list
- length of list is exactly the number of elements you've put into the list

Note the last bullet. With lists, you allocate a new piece of memory (using `cons`) when you need to add an element, but with vector, you allocate all the memory you need first, even if you don't have enough data to fill it up.

Also, just as you can have deep lists, where elements of a list may be a list as well, you can also have "deep" vectors, often referred to as n-dimensional arrays, where n refers to how "deep" the deep vector is. For example, a table would be a 2-dimensional array – a vector of vectors. Note that, unlike in, say, C, your each vector in your 2D table does NOT have to have the same size! Instead, you can have variable-length rows inside the same table. In this sense, the vectors of Scheme are more like the arrays of Java than C.

QUESTIONS

1. **Write a procedure `(sum-of-vector v)` that adds up the numbers inside the vector. Assume all data fields are valid numbers.**

2. **Write a procedure `(vector-copy! src src-start dst dst-start length)`. After the call, `length` elements in vector `src` starting from index `src-start` should be copied into vector `dst` starting from index `dst-start`.**

```
STk> a => #(1 2 3 4 5 6 7 8 9 10)
STk> b => #(a b c d e f g h i j k)
STk> (vector-copy! a 5 b 2 3) => okay
STk> a => #(1 2 3 4 5 6 7 8 9 10)
STk> b => #(a b 6 7 8 f g h i j k)
```

3. Write a procedure `(insert-at! v i val)`; after a call, vector `v` should have `val` inserted into location `i`. All elements starting from location `i` should be shifted down. The last element of `v` is discarded.

```
STk> a => #(i'm like you #[unbound] #[unbound])
STk> (insert-at! a 1 'bohemian) => okay
STk> a => #(i'm bohemian like you #[unbound])
```

4. Write a procedure `(vector-double! v)`. After a call, vector `v` should be doubled in size, with all the elements in the old vector replicated in the second half. So,

```
STk> a => #(1 2 3 4)
STk> (vector-double! a) => okay
STk> a => #(1 2 3 4 1 2 3 4)
```

5. Write a procedure `reverse-vector!`. Do I have to explain what it does?

6. Write a procedure `(square-table! t)` that takes in a rectangular table and squares every element.

Meta-metaevaluation

Let's examine how the metacircular evaluator represents things in underlying Scheme. A primitive procedure is represented as list whose first element is the word PRIMITIVE and whose second element is the actual procedure:

```
(PRIMITIVE #[subr car]) ;; car in mceval
```

Non-primitive procedures are a bit more interesting. They're actually a list of four elements: the word `PROCEDURE`, a list of its parameters, a list of expressions in the body, and the environment it was *created in*:

```
(define (foo a b) (+ a b)) =>
  (PROCEDURE (a b) ((+ a b)) <the-global-environment>)
```

Does that last part sound familiar? The metacircular evaluator is just about as powerful as real Scheme, and the primary reason for that is because we're using applicative order and the **environment model**.

Let's look at how environments are represented. The pair structure handles the environment model; each environment corresponds to a pair whose its `car` points to the variable/value bindings (which we'll call a **frame** from now on) and whose `cdr` points to the next environment (just like in the environment model). The global environment, then, has a null `cdr`. What does a frame look like? Well, it's a pair whose `car` contains all of the variables, and whose `cdr` contains all of the values. Here's an example environment created through a `let` call:

```
(let ((x 3) (y 5)) ...) =>
  (((x y) 3 5) <the-global-frame>) ;; the printout of environment 1
```

Of course, multiple environments can and will point to the same environment, so the entire environment diagram is not a straight list structure. Notice, however, that it's simple to simulate evaluating a variable with this model! Simply check the current frame for a binding, and if it's not there, `cdr` to the next environment until we either find our variable or go past the global environment – in which case the next environment is null.

QUESTION

Write `lookup-variable-value`, which takes a variable and starting environment and returns the value associated with the variable or an error if it isn't found after the global environment.

Regular Metaevaluation

So all that above was simple, right? Now let's look at some code (with helpful comments written by yours truly):

```
(define (mc-eval exp env)
  (cond
   ((self-evaluating? exp) exp)
   ((variable? exp) (lookup-variable-value exp env))      ;; you just did this above
   ((quoted? exp) (text-of-quotation exp))                ;; (cadr exp)
   ((assignment? exp) (eval-assignment exp env))          ;; question 1 below
   ((definition? exp) (eval-definition exp env))          ;; question 2 below
   ((if? exp) (eval-if exp env))                          ;; essentially uses Scheme if
   ((lambda? exp)
    (make-procedure (lambda-parameters exp)               ;; (cons 'procedure args)
                    (lambda-body exp)
                    env))
   ((begin? exp)
    (eval-sequence (begin-actions exp) env))
   ((cond? exp) (mc-eval (cond->if exp) env))             ;; makes nested ifs
   ((application? exp)
    (mc-apply (mc-eval (operator exp) env)
              (list-of-values (operands exp) env)))       ;; map mc-eval onto exps
   (else
    (error "Unknown expression type -- EVAL" exp))))

(define (mc-apply procedure arguments)
  (cond ((primitive-procedure? procedure)
         (apply-primitive-procedure procedure arguments)) ;; use underlying Scheme apply
        ((compound-procedure? procedure)
         (eval-sequence
           (procedure-body procedure)
           (extend-environment                            ;; question 3 below
             (procedure-parameters procedure)
             arguments
             (procedure-environment procedure))))
        (else
         (error
          "Unknown procedure type -- APPLY" procedure))))
```

A lot of code, but remember `mc-eval` is just an implementation of environment diagrams. One of the mysteries not covered above is `eval-sequence`. This is how `begin` statements and, more importantly, compound procedures are handled:

```
(define (eval-sequence exps env)
  (cond ((last-exp? exps)                         ;; last-exp? => (null? (cdr exps)))
         (mc-eval (first-exp exps) env))          ;; first-exp => car
        (else (mc-eval (first-exp exps) env)
              (eval-sequence (rest-exps exps) env))))
```

Fairly uninteresting for `begin` statements, but notice how it's called in `mc-apply` for compound procedures – the evaluating environment is a new environment created using `extend-environment`, which you'll code in about 2 questions.

QUESTIONS

1.
```
(define (eval-assignment exp env)
  (set-variable-value! (assignment-variable exp)            ;; (cadr exp)
                       (mc-eval (assignment-value exp) env) ;; (caddr exp)
                       env)
  'okay)
```
Modify your `lookup-variable-value` code above to create `set-variable-value!` (which takes an additional value argument).

2.
```
(define (eval-definition exp env)
  (define-variable! (definition-variable exp)            ;; (cadr exp)
                    (mc-eval (definition-value exp) env) ;; (caddr exp)
                    env)
  'okay)
```
Modify your `set-variable-value!` code above to create `define-variable!`. You should write a helper `add-binding-to-frame!` that takes a variable, value, and frame, and adds the binding into the given frame.

3. **Write `(extend-environment vars vals base-env)` that takes in a list of variables, a list of values, and an environment to extend, and creates the new environment (as when you call a procedure in the environment model).**

4. **Write `(mc-map fn ls)` to work with `mc-eval`. It will be installed as the primitive procedure associated with `map`. `fn` is defined in our new representation.**

Dynamic Scope

The major difference between lexical and dynamic scope's `apply`: In lexical scope, we extend the *procedure environment* (right bubble) of the procedure we're invoking, whereas in dynamic scope, we extend the *current environment*. (Review: Which one does Scheme use?)

Note that in dynamic scope, the right bubble is entirely unnecessary. Dynamic scope tends to be much easier to implement and model, but lexical scope gives us a nice way to do **local state**. It is important to understand dynamic scope though, and it may prove to be of some relevance to you in the near future (*cough* `proj4`).

There are various advantages that one has over the other, and I'll let you read about those in the lecture notes.