CS61A Notes 8– The Attack of the Environmentalists (v1.0)

Imperative Perils

For those of you used to programming in C or Java or almost any other languages, the idea of "assignment" – assigning a value to a symbol – should be a dusty old hat. Certainly, there's nothing exotic about it. But why have we been so cautious about introducing this concept to you – 9 weeks into the semester! – when, if you take an introductory CS class in any other language, you would see assignment on the first day? And why do I write such long and convoluted sentences?

Before that, note that we have now stepped out of pure "functional programming", and now has functional programming with *side effects*. Here, nice properties of functional programming disappear; more specifically, **the order in which expressions are evaluated matters**, and **two calls to the same procedure with the same arguments may not produce the same result!**

Beyond that, you should note one more thing. Up until now, whenever we make a procedure call like `(square 3)` or `(+ 2 5)`, we are interested in what the procedure call *returns*. Such is the fundamental idea of functional programming – procedures are interesting only because they take in something and produce something else (deterministically, might I add!). This is not always true now. For example, if we say `(set! x 5)`, what do we want it to return? Well, we really don't care what it returns, as long as it binds the value of `5` to `x` after it's done! For such expressions, **we're more interested in their side effects than their return values**.

Precisely because of that, sometimes we'll want to evaluate multiple statements at once (each having some side effects), whereas before we only evaluated one. Sometimes you can do that directly:

```
(define (foo x) (set! x 3) (set! y 5) (cons x y))
```

Note that the above defined procedure, `foo`, has three statements, only one of which returns something interesting. `STk` will execute the three statements in the order of appearance, and return the value of the last expression. You can do the same with a `cond` statement:

```
(cond ((< x 5) (set! x 10) (set! y 5) '(cs61a rocks and so do toads))
      (else (set! x 5) (set! y 10) '(flan in the face)))
```

However, you cannot do the same for `if` statements for obvious reasons:

```
(if (> x 3) (set! x 3) (set! y 5) (set! z 10))
```

What does the above mean? There's no way for `STk` to tell what you intended (`set! x` on `#t? set! x` and `y` on `#t? set! x`, `y` and `z` on `#t?`) And we can't simply enclose things in parentheses, as you might be tempted to do; `((set! x 3) (set! y 5))` means a procedure call where the procedure is the value of `(set! x 3)` and the argument to the procedure is the value of `(set! y 5)`, resulting in pure gibberish. It seems that we need a new piece of syntax to deal with such situations!

This calls for `begin`, which groups together a sequence of statements, and returns the value of the last expression. So we can instead, do:

```
(if (> x 3) (set! x 3) (begin (set! y 5) (set! z 10)))
```

which means to `set! y` and `z` if `#f`.

Now, side effects are obviously not all bad; if you're careful, it's immensely more efficient and can often make the code more compact. The keyword, of course, is "careful"; let's look at what havocs they bring.

Assigning Things To Things And Stuff

Our way of doing this in Scheme is **`set!`**. Note that `set!` is a special form. Consider if it's not; then, if we do:

```
(define x 5)
(set! x 3)
```

and `set!` is not a special form, it will evaluate what `x` is – `5` – and try to set the value of `5` to be `3`, making no sense at all. What we really meant to say, is to set the value bound to the symbol `x` (not the value of `x`) to `3`. There's really not much to talk about here, so let's try a few. The key is to keep the scope of variables straight!

QUESTIONS

1. **Personally – and don't let this leave the room – I think `set!` is useless. I mean, why do `set!`, when we can always just redefine a variable using a `define` statement? Instead of doing `(set! x 3)`, why don't we just do `(define x 3)` again? I propose the following alternative implementation of `counter`, similar to the one in class:**

The Old Way

```
(define count
  (let ((current 0))
    (lambda()
      (set! current (+ 1 current))
      current)))

(count) ==> 1
(count) ==> 2
```

My Brilliant New Way

```
(define count
  (let ((current 0))
    (lambda ()
      (define current
              (+ current 1))
      current)))
```

How dumb am I? What happens when I use my new brilliant implementation?

2. **Consider these definitions:**
   ```
   (define x 3)
   (define (z) (set! x 5) x)
   ```
   what would `(list (z) x)` return?

3. **(SICP ex. 3.8) Keeping number 2 in mind, define a procedure `f` so that, given the procedure call**
   ```
   (+ (f 0) (f 1))
   ```
   if `STk` evaluates from left to right, it returns `0`, and if `STk` evaluates from right to left, it returns `1`.

   ```
   (define (f x))
   ```

4. **Define a procedure `fib` so that, every time it is called, it returns the next Fibonacci number, starting from 1:**
   `(fib) ==> 1; (fib) ==> 2; (fib) ==> 3; (fib) ==> 5; (fib) ==> 8`, etc.

The Environment Diagrams

The biggest blow having side effects deal us is that we can no longer use the substitution model of evaluation that we've fallen so hopelessly in love with. Consider this:

```
> (define (foo x) (set! x 3) x)     > (foo 10)
```

If we use the substitution model, then when we do `(foo 10)`, we will replace all occurrences of `x` inside `foo` with `10`. Note that this means `(foo 10)` will then return `10`, not `3`! Obviously, we can no longer afford to be so obtuse, as values of the variables can now change at any point in time. Thus, we need to use what is called the "**environment model**" of evaluation. Briefly, we keep an "environment" of bindings from variable to value, and every time we encounter a variable, we look up its value in the current environment. Note that we only lookup the value when we see the variable, and so we'll lookup the value of `x` only *after* we've already set `x` to be `3`. Drawing these environment diagrams is one of the most hated aspects of CS61A, but remember the mantra: this is what the Scheme interpreter does, and since you're much smarter than the interpreter, if the interpreter can do it, you can do it.

An **environment frame** is a box that contains bindings from variables to values. An environment frame can "extend" another frame; that is, this frame can see all bindings of the frame it extends. We represent this by drawing an arrow from an environment frame to the frame it is extending. The global environment is the only environment that extends nothing. Here's how to go about it:

1. Start with a box labeled **global environment.** This box starts out with bindings for all the primitives like `+`, `cons`, `map`, etc.
2. Set the **current frame** to be the global environment. The current frame is just the environment that you're in at this moment, and of course, you start in the global environment.
3. **Evaluate** your expressions, one by one. I'd recommend converting all sugar-coated procedure definitions to their raw forms –
   `(define (square x) (* x x))` ==> `(define square (lambda(x) (* x x)))`

Evaluation rules: If you're evaluating a:

- **constant** (numbers, strings, etc), they are self-evaluating so don't do any work.
- **variable**, try to find a binding for it in the *current frame*. Failing that, follow what environment the current frame points to, and try to find the binding there, and so on, until you reach the global environment. If it's still not in the global environment, then you can't find it; it's an error! (Recall that the global environment contains all the bindings for primitives like `cons`, `+`, etc.)
- **`define` expression**, first evaluate the value to bind to according to these evaluation rules, then add an entry into the *current frame* for the variable pointing to that value.
- **`lambda` expression**, draw two balls next to each other (usually just in the space around the boxes). The first ball should point to the text of the `lambda` – the argument list and the body – and the second ball should point to the *current frame*. The frame this circle points to is called the "procedure environment".
- **`let` expression**, evaluate all the values of the `let`-bindings first. Then, draw a new box that points to the *current frame*, and let the *current frame* be the new box. In the new box, add all the variables of the `let`-bindings, and let them point to the values you obtained. Then, evaluate the body of the `let` expression from within the new current frame. After you're done, go back to the previous frame.
- **`set!` expression**, evaluate the value to `set!` to. Then, for the given variable, find its closest binding from the current frame, and overwrite the old value with the new value.
- **any other special forms**, just evaluate what you're supposed to in the *current frame*.
- **procedure call**, then check if the procedure is a:
  - **primitive procedure** – these work by magic, so just apply the procedure intuitively in your head.
  - **compound procedure** – *evaluate all the arguments first*. Then, create a new box, and *make this box point to the procedure environment box of the procedure*. Note that you ***do not*** point to the current frame! Set this new box as the new *current frame*, and add all the parameters into the box,

and have them point to the argument values you evaluated. Evaluate the body of the procedure in the current frame. Once done, go back to the *frame from which you made the procedure call*.

QUESTIONS: Draw environment diagrams for the following:

1.
```
(define (f + -) (+ ((lambda(-) (- 3 5)) -) 10))
(f - +)
```

2.
```
(define (hmm n) (lambda(x) (+ x y n)))
(define (uhh y)
   (define hmm-y (hmm y))
   (hmm-y 2))
(uhh 42)
```

3.
```
(define answer 0)
(define (square f x)
   (let ((answer 0))
      (f x)
      answer))
(square (lambda(n) (set! answer (* n n))) 3)
```

4.
```
(define a 3)
((lambda(a)
   ((lambda(a) (a))
    (lambda() (set! a 'myxomatosis)))
   a)
 (* a a))
```

5.
```
(define a 'scatterbrain)
((lambda(a b) (b) a)
 a
 (let ((b 'cuttooth))
   (lambda() (set! a b))))
a
```

6.
```
(define (slow-op-maker op)
   (let ((old-result #f))
      (lambda(x)
         (let ((return old-result))
            (set! old-result (op x))
            return))))
(define slow-sqr (slow-op-maker square))
(slow-sqr 3)
(slow-sqr 5)
(define slow-cube (slow-op-maker cube))
(slow-cube 3)
(slow-cube 5)
```