CS61A Notes – Week 6: Scheme1, Data Directed Programming

You Are Scheme – and don't let anyone tell you otherwise

If you're not already crazy about Scheme (and I'm sure you are), then here's something to get excited about: only 6 weeks into the semester, and you can already write a fairly competent interpreter using Scheme! A *certain lesser college* in the South Bay doesn't let you do that till your third year!

Now, now, I know it's a little intimidating to look at this big piece of code with recursion out the wazoo, but hang in there and throw in the time; this code is only going to get bigger and more complicated as the semester rolls on. But the interpreter will also grow, not just in size, but in power and elegance as well.

Great. Now you look so excited to learn.

Our intention is to build a Scheme interpreter, a program that reads in Scheme code and executes it much like STk does. It's pure, pure coincidence that we happen to be using Scheme to build this very Scheme interpreter. **Don't get confused by that!** Make sure you separate what is `Scheme1`, and what is STk. Sometimes, **it might help to pretend you're not interpreting Scheme**, but just some crazy, bizarre language.

Another word of caution: There are many, many ways to implement a Scheme interpreter. You'll often find yourself asking, *why* are we doing things this way? Well, very possibly, we don't have to, but that's a design decision we made. So be more concerned with *how* it works before you ask *why?*

The Meat is in `eval-1` and `apply-1`

The whole operation of `Scheme1` depends on the mutual recursion of `eval-1` and `apply-1`. Here's their job:

`eval-1`

- **INPUT: a valid Scheme expression**
- **OUTPUT: the result of evaluating the Scheme expression; the "value" of the expression**
- If `exp` is "constant" (numbers, strings, `#t`, `#f`, etc.), then return itself.
- If `exp` is a symbol (`+`, `*`, `car`, etc.), then use the underlying STk's `eval` to evaluate it. This is where `Scheme1` cheats; all symbols are assumed to be primitive procedures of STk. We do not expect to see actual variables here, since we don't have `define`, and all variables in a `lambda` body should've already been substituted with values.
- If `exp` is a special form, do special form voodoo (`if`, `lambda`, `quote`).
- If `exp` is a procedure call, evaluate what the procedure is (using `eval-1`), and evaluate the arguments. Then, call `apply-1` with the evaluated procedure and arguments.

`apply-1`

- **INPUT: a procedure and its arguments, both already evaluated**
- **OUTPUT: the result of applying the procedure to its arguments**
- If the procedure is a primitive procedure (+, *, cons, car, etc.), then use STk's apply procedure to perform primitive voodoo. We could tell it's a primitive procedure using procedure? because, remember, eval-1 should've already called STk's eval on it already and turned it into an STk procedure.
- If the procedure is a compound procedure (a lambda expression), call eval-1 on the body of the lambda with the parameters substituted with the argument values.

It is crucial to understand how `eval-1` and `apply-1` interact! It's really not that complicated, and it's exactly what I've been muttering on the board whenever I pretend I'm Scheme and evaluate an expression. We'll step through an example soon with primitive procedures, but first, let's look at how primitives are represented.

A Word On Special Forms

Before we dive too deeply into `Scheme1` let's briefly consider special forms. Where do we actually deal with them? A quick glance at the code reveals that they're very special indeed – we take care of them in `eval-1`. Note that we, in no sense, consider them to be compound procedures, and we do NOT call `apply-1` on a special form!

Remember why special forms are special – we don't always evaluate every argument. If we treat it as a procedure call, then we're going to `map eval-1` on all of its arguments, which is exactly what we don't want to do.

So remember – if you want to add a special form, like we did for "`and`", the place to do it is in `eval-1`. You must catch it before `eval-1` thinks it's a procedure and evaluates all its arguments!

The Art of Being Complicated

Note that, as a quirk specific only to `Scheme1`, a `lambda` expression is *self-evaluating* (you can see this in `eval-1`). That is, a `lambda` expression evaluates to itself, just like `3` evaluates to `3`. Note, however, that a `lambda` expression is *not* a procedure! This is an important distinction. A `lambda` expression is an *expression* – something that you pass into `eval-1` as an argument. A procedure is a *value* – something that `eval-1` returns. It's simply a matter of coincidence that a procedure in `Scheme1` is represented just like a `lambda` expression. This is rather unfortunate, but will be fixed in `mc-eval`, the last iteration of these "normal" Scheme interpreters.

How is a compound procedure different from a primitive procedure? Well, to `eval-1`, a procedure call is a procedure call, and when `eval-1` sees one, it simply evaluates the procedure and its arguments, regardless of whether the procedure is primitive or compound. It then just passes both the procedure and its arguments to `apply-1`.

Thus it is `apply-1` who distinguishes between primitive and compound procedures. Thus, the "`proc`" argument of `apply-1` can be one of two things:

- **a primitive procedure** – the result you get when `eval-1` calls STk's `eval` on a symbol like `+` or `car` which are bound to primitive procedures in STk. These will pass STk's `procedure?` test. To apply a primitive procedure, `apply-1` will use STk's `apply` procedure to apply an STk procedure to the list of arguments. Here we're just passing the work off to the underlying STk.
- **a compound procedure** – this is represented as a `lambda` expression (though it is a *procedure*, not an *expression*). These will pass the `lambda-exp?` test. Recall that, according to our "substitution model", in order to evaluate a compound procedure (a `lambda`) call, we substitute argument values for parameters in the procedure's body, and then evaluate the body. More precisely, we're going to call `eval-1` on the body of the procedure, after we substitute all the formal parameters with the evaluated argument values.

To perform substitution when doing a compound procedure call, we do something that is very similar to the `substitute2` that you wrote, where we take in the body of the procedure, and wherever we see a formal parameter, we replace it with the corresponding argument. It does have this extra bound argument; this is a list of symbols that we *do not* want to substitute for. For example, given `((lambda(x) ((lambda(x) (* x 2)) 5)) 10)`, when we're substituting `x` for `10`, we do not replace the `x` in the inner `lambda` with `10`. Instead, we're supposed to leave that `x` alone since that `x` refers to the inner `x` (the one that will be bound to `5`) rather than the outer `x` (the one that's bound to `10`).

All of this will become clearer once we try to trace through an example. For this example, we're going to gloss over most of substitution and just focus on the interaction between `eval-1`, `apply-1` and `substitute`.

```
(((lambda (x) ((lambda (y) (lambda (x) (+ x y x))) x)) 5) 10)
```

Before doing anything, do it by hand – what should it return?

➔ `eval-1` called with `(((lambda(x) ((lambda(y) (lambda(x) (+ x y x))) x)) 5) 10)`; it sees that it is a procedure call, naturally – a list of two elements. What's the procedure? Well, to find out, call `eval-1` on it!

- ➔ eval-1 called with ((lambda(x) ((lambda(y) (lambda (x)(+ x y x))) x)) 5). Looks like a procedure call to me! What's the procedure?
    - ➔ eval-1 called with (lambda(x) ((lambda(y) (lambda(x) (+ x y x))) x)). This is just a lambda expression; it evaluates to a procedure represented by the same lambda expression, so we will just return itself.
    - ← eval-1 returns (lambda(x) ((lambda(y) (lambda(x) (+ x y x))) x)). Next, we need to map eval-1 over the arguments, which is just the list of a single argument – (5). Mapping eval-1 over that, of course, will return to us (5), since 5 is self-evaluating. We're done evaluating the procedure and the arguments, so it's time to apply:
    - ➔ apply-1 called with proc (lambda(x) ((lambda(y) (lambda(x) (+ x y x))) x)), and argument list (5). apply-1 sees that this is a compound procedure, so first it tries to substitute 5 for x in the body of the expression:
        - ➔ substitute called with exp as ((lambda(y) (lambda(x) (+ x y x))) x), params as (x), args as (5), and bound as (). Let's skip over the details of substitute, but we already know what it is supposed to return:
        - ← substitute returns ((lambda(y) (lambda(x) (+ x y x))) 5). Evaluate the body:
        - ➔ eval-1 called with ((lambda(y) (lambda(x) (+ x y x))) 5). Looks like a procedure call! The procedure will evaluate to (lambda(y) (lambda(x) (+ x y x))), and the argument list will evaluate to (5). Let's apply this puppy!
            - ➔ apply-1 called with proc (lambda (y) (lambda (x) (+ x y x))) and argument list (5). The procedure is compound, so we try to call substitute on its body:
                - ➔ substitute called with (lambda(x) (+ x y x)) with params (y), args (5), and bound (). It does the obvious thing:
                - ← substitute returns (lambda(x) (+ x 5 x)). Now we evaluate the body:
                - ➔ eval-1 called with (lambda(x) (+ x 5 x)). Why, that's a lambda expression! We're going to just return the procedure, represented as itself:
                - ← eval-1 returns (lambda(x) (+ x 5 x))
            - ← apply-1 returns (lambda(x) (+ x 5 x))
        - ← eval-1 returns (lambda(x) (+ x 5 x))
    - ← apply-1 returns (lambda(x) (+ x 5 x))
- ← eval-1 returns (lambda(x) (+ x 5 x)), and we've finally found out what the procedure of our original procedure is! What are the arguments? Well, it's just (10). Time to apply:
- ➔ apply-1 called with (lambda(x) (+ x 5 x)) and argument list (10). Of course, this is a compound procedure, so we, as usual, call substitute on the body:
    - ➔ substitute called with exp as (+ x 5 x), params (x), args (10), and bound the empty list ().
    - ← substitute returns (+ 10 5 10). Now we can evaluate the body.
    - ➔ eval-1 called with (+ 10 5 10). A procedure call! The procedure is evaluated as the STk + procedure, and the argument list evaluated as (10 5 10). We call apply-1:
        - ➔ apply-1 called with STk's + procedure and args (10 5 10). We will use STk's apply to apply the primitive procedure, getting 25.
        - ← apply-1 returns 25
    - ← eval-1 returns 25
- ← apply-1 returns 25

← eval-1 returns 25

Accept No Substitutes (well, except you kind of have to)

So, just what does substitute do, given an expression and lists params, args and bound?

- If the expression is a **symbol**, then it might be a member of the params or bound lists. If the symbol is in the bound list, then ignore it; otherwise, substitute it with the corresponding value in the args list. We use a very simple procedure called lookup to do this. If it's not a member of either lists, then it's probably a symbol referring to a primitive procedure, like + or car. In that case, where lookup won't find it in the params list, it will just give up and not try to replace it with anything.
- If the expression is a **quoted expression** or a **constant**, then don't substitute it with anything.
- If the expression is a **lambda expression**, then return the same lambda with its body substituted correctly. Look at the code; it makes a new lambda, whose arguments stay as they were, but whose body is passed recursively to substitute. Note, however, that in the recursive call, we append the formal parameters of the lambda expression onto the bound list; this is because, in the body of this lambda expression, we don't want to substitute any argument values for its formal parameters, even if we have a

binding in the `param-args` lists for a variable with the same name as one of its formal parameters. We already touched on why not.

- If the expression is anything else, it is a **procedure call**; then, `map substitute` on every sub-expression.

QUESTIONS

1. **Try it yourself: trace each call to `eval-1` and `apply-1` for the expression `(+ 3 (- 8 5))`.**

2. **How about for `(+ 3 ((lambda (x) (- x 1)) 5))`?**

3. **If I type this into STk, I get an unbound variable error:**
   `(eval-1 'x)`
   This surprises me a bit, since I expected `eval-1` to return `x`, unquoted. Why did this happen? What should I have typed in instead?

4. **Hacking `Scheme1`: For some reason, the following expression works:**
   `(`(lambda(x) (* x x)) 3)`
   Note the quote in front of the `lambda` expression. Well, it's not supposed to! Why does it work? What fact about `Scheme1` does this exploit?

Type Tagging, Data Directed Programming, and Message Passing

Up till now, we have been writing "smart programs" – programs that know what to do given the arguments. But with data-directed programming, we're slowly moving toward the paradigm where programs are dumb, but data are smart. If you recall, procedures like apply-generic are short and simplistic, but the data you pass in – arguments with type tags – and the data you store in the global table – containing separate procedures for dealing with different type tags and operations – are what knows what to do with the data. Hence the term “data-directed programming”.

The advantage is simple: maintainability. Remember, programs or procedures are complicated beasts, and you should cringe every time you need to modify a working procedure. Yet data is simple – putting new items into a global table doesn’t require you to alter existing code. Adding new features will rarely break old, working code.

This is also useful when you’re dealing at once with multiple representations of data, which you'll see more of in this week's homework. We will do something similar here.

But before diving into type-tagging and data directed programming, let's first talk about yet another approach: message passing. Imagine a world where data was intelligent and completely self-aware – that's message passing! In data-directed programming, the data “directed” the programming – that is, the data, with its appropriate tagging and established global table, is able to guide the program by providing the correct procedures for the correct situation. In message passing, the data <u>is</u> the program. You hand the data a “message” - an action you want it to perform – and the data will carry it out on it's own. Message passing is the peak of intelligent data, because everything is handled internally by the data. All it needs is the right message.

QUESTION: The TAs have broken out in a cold war; apparently, at the last midterm-grading session, someone ate the last potsticker and refused to admit it. It is near the end of the semester, and Colleen really needs to enter the grades. Unfortunately, the TAs represent the grades of their students differently, and refuse to change their representation to someone else’s. Colleen is far too busy to work with five different sets of procedures and five sets of student data, so for educational purposes, you have been tasked to solve this problem for her. The TAs have agreed to type-tag each student record with their first name, conforming to the following standard:

```
(define type-tag car)
(define content cdr)
```

It’s up to you to combine their representations into a single interface for Colleen to use.

IN CLASS:

1. **Write a procedure, `(make-tagged-record ta-name record)`, that takes in a TA’s student record, and type-tags it so it’s consistent with the `type-tag` and `content` accessor procedures defined above.**

2. **A student record consists of two things: a “name” item and a “grade” item. Each TA represents a student record differently. Hamilton uses a list, whose first element is a name item, and the second element the grade item. Eric uses a `cons` pair, whose `car` is the name item, and the `cdr` the grade item. Write `get-name` and `get-grade` procedures that take in a student record and return the name or grade items, respectively. We will do this in three different ways:**

a. Conventional-style type tagging

b. Data-directed programming

c. Message passing – this one's a little different. Instead of going off an existing TA's implementation, you now have the chance to create your own. Create an implementation of student records using message passing, which can accept the messages `get-name` and `get-grade`.

EXTRA PRACTICE:

3. **Each TA represents names differently. Kevin uses a `cons` pair, whose `car` is the last name and whose `cdr` is the first. Phill is so cool that a "name" is just a word of two letters, representing the initials of the student (so George Bush would be `gb`). Write generic `get-first-name` and `get-last-name` procedures that take in a tagged student record and return the first or last name, respectively. Try this using conventional-style type tagging, then with data-directed programming.**

4. **Each TA represents grades differently. Eric is lazy, so his grade item is just the total number of points for the student. Stephanie is more careful, so her grade item is an association list of pairs; each pair represents a grade entry for an assignment, so the `car` is the name of the assignment, and the `cdr` the number of points the student got. Write a generic `get-total-points` procedure that takes in a tagged student record and return the total number of points the student has. Try this using conventional-style type tagging, then with data-directed programming.**

5. **Now Colleen wants you to convert all student records to the format she wants. She has supplied you with her record-constructor, `(make-student-record name grade)`, which takes in a name item and a grade item, and returns a student record in the format Colleen likes. She also gave you `(make-name first last)`, which creates a name item, and `(make-grade total-points)`, which takes in the total number of points the student has and creates a grade item. Write a procedure, `(convert-to-colleen-format records)`, which takes in a list of student records, and returns a list of student records in Colleen's format, each record tagged with `'Colleen`.**