CS61A Notes 06 – Data-Directed Programming (with some Scheme-1) (v1.1)

Data-Directed Programming

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, like the corporate divisions problem you saw in the lab. We will do something similar:

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 Professor Harvey 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. Professor Harvey 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 him. 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 Professor Harvey to use.

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. Min uses a list, whose first element is a name item, and the second element the grade item. Justin uses a `cons` pair, whose `car` is the name item, and the `cdr` the grade item. Make calls to `put` and `get`, and write generic `get-name` and `get-grade` procedures that take in a tagged student record and return the name or grade items, respectively.**

3. **Each TA represents names differently. Darren uses a `cons` pair, whose `car` is the last name and whose `cdr` is the first. Jerry 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`). Make calls to `put` and `get` to prepare the table, then 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.**

4. **Each TA represents grades differently. Ahmed is lazy, so his grade item is just the total number of points for the student. Justin is more careful, so his 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. Make calls to `put` and `get` to prepare the table, and write a generic `get-total-points` procedure that take in a tagged student record and return the total number of points the student has.**

5. **Now Professor Harvey wants you to convert all student records to the format he wants. He has supplied you with his 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 Professor Harvey likes. He 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-harvey-format records)`, which takes in a list of student records, and returns a list of student records in Professor Harvey's format, each record tagged with `'Brian`.**

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

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 Scheme0/1, 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.

QUESTIONS

1. **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?

2. **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?