CS61A Notes – Week 14: Lazy evaluator, Logic programming

The Lazy Way Out

The lazy evaluator implements normal order of evaluation that you learned about way back when. We do this by delaying *execution as much as possible by* thunking *it; when we can't delay it further, we* force *the thunk to obtain the actual value.*

A **thunk** in the lazy evaluator is a list whose first element is the word `thunk`, second element the expression we're delaying, and third element the environment in which to evaluate the expression when we need to force the thunk.

There's only one place where we delay evaluation: **we thunk all arguments to a compound procedure**.

There are four places in which we want to force:

1. We always force the operator of a procedure call
2. We always force arguments to a primitive procedure
3. We always force what we're printing out to the screen on the top level
4. In special cases of special forms, like the predicate of `if`, we also want to force thunks

The procedure that actually does the delaying is called `delay-it`:

```
(define (delay-it exp env)
    (list 'thunk exp env))
```

Obviously, this just creates a thunk that looks just as we described above. The procedure that we use to force an expression is called `actual-value`:

```
(define (actual-value exp env)
    (force-it (mc-eval exp env)))
```

Note that `actual-value` *always* takes in a valid Scheme expression, and never a thunk! Thus, it can first call `mc-eval` to evaluate the expression. Then, since `mc-eval` might return a thunk, it needs to call `force-it` to actually force the thunk to get the real answer.

Note that the lazy evaluator also memoizes forced thunks. Take a look at the memoizing version of `force-it`:

```
(define (force-it obj)
    (cond ((thunk? obj)
           (let ((result (actual-value (thunk-exp obj) (thunk-env obj))))
               (set-car! obj 'evaluated-thunk)
               (set-car! (car obj) result)
               (set-cdr! (cdr obj) '())
               result))
          ((evaluated-thunk? obj)
           (thunk-value obj))
          (else obj)))
```

First, note the mutual recursion between `force-it` and `actual-value`; one calls the other. This is because forcing an object might yield another thunk, so we need to keep forcing it until we get some actual value. And note how we perform the memoization: when we find a thunk, we first call `actual-value` on the delayed expression to find the value. Then, we *transform* the thunk into an *evaluated-thunk*, and store the result of the forcing into the evaluated-thunk. From then on, if we try to force this evaluated-thunk again, we simply return the value we've already calculated.

QUESTIONS: What is printed at each line?

1.
```
> (define x (+ 2 3))
> x => ?
```

```
> (define y ((lambda (a) a) (* 3 4)))
> y => ?
```

```
> (define z ((lambda (b) (+ b 10)) y))
> z => ?
```

2.
```
> (define count 0)
> (define (foo x y) (x y))
> (define z (foo (lambda (a) (set! count a) (* a a))
                 (begin (set! count (+ 1 count)) count)))
> count => ?
```

```
> z => ?
```

```
> count => ?
```

3.
```
> (define count 0)
> (define (incr!) (set! count (+ count 1)))
> (define (foo x)
      (let ((y (begin (incr!) count)))
          (if (<= count 1)
              (foo y)
              x)))
> (foo 10) => ?
```

Paradigm Shift Again (Why Not?)

With this many paradigm shifts in a single semester, we expect you to at least be able to pronounce the word "paradigm" correctly after this class.

To reiterate, we are now in the realm of logic or declarative programming, at least for a week. Here in the magical world of the non-imperative, we can say exactly what we want – and have the computer figure out how to get it for us. Instead of saying how to get the solution, we describe – declare – what the solution looks like.

The mind-boggling part of all of this is that it all just works through pattern matching. That is, there are no "procedures" in the way you're used to; when you write out a parenthesized statement, it's not really a procedure call, and you don't really get a return value. Instead, either you get entries from some database, or nothing at all.

Be Assertive And Ask Questions (Fitter, Happier, More Productive)

There are two things you can type into our query system: an **assertion** and a **query**.

A **query** asks whether a given expression matches some fact that's already in the database. If the query matches, then the system prints out all matches in the database. If the query doesn't match to anything, you get no results.

An **assertion** states something true; it adds an entry to the database. You can either assert a simple fact, or a class of facts (specified by a "rule").

So here's an assertion that you've seen: `(assert! (justin likes chocolate))`

You can also assert rules. In general, a rule looks like: `(rule <conclusion> <subconditions>)`

And it's read: "conclusion is true if and only if all the subconditions are true". Note that you don't have to have subconditions! Here's a very simple rule:

```
(rule (same ?x ?x))
```

The above says two things satisfy the "same" relation if they can be bound to the same variable. It's deceptively simple – no subconditions are provided to check the "sameness" of the two arguments. Instead, either the query system can bind the two arguments to the same variable `?x` – and it can only do so if the two are equivalent – or, the query system can't.

And, of course, the rule of love:

```
(assert! (rule (?person1 loves ?person2)
               (and (?person1 likes ?something)
                    (?person2 likes ?something)
                    (not (same ?person1 ?person2))))))
```

The above rule means that `?person1` loves `?person2` if the three conditions following can all be satisfied – that is, `?person1` likes `?something` that `?person2` also likes, and that `?person1` is not the same person as `?person2`.

Note the "`and`" special form enclosing the three conditions – an entry in the database must satisfy ALL three conditions to be a match for the query. If you would like a rule to be satisfied by this *or* that condition, you can either use the `or` special form in the obvious way, or you can make two separate rules. For example, if one loves another if they like the same things *or* if `?person1` is a parent of `?person2`, we would add the following to the database:

```
(assert! (rule (?person1 loves ?person2) (parent ?person1 ?person2)))
```

Note the new rule does NOT "overwrite" the previous rule; this is not the same thing as redefining a procedure in Scheme. Instead, the new rule complements the previous rule.

To add to confusion, you can also use the `and` special form for queries. For example,

```
(and (justin loves ?someone) (?someone likes chocolate))
```

is a query that finds a person Justin loves because that person likes chocolate.

There's another special form called `lisp-value`: `(lisp-value <pred?> <arg1, arg2, ...>)`

The lisp-value condition is satisfied if the given `pred?` applied to the list of `args` return `#t`. For example, `(lisp-value even? 4)` is satisfied, but `(lisp-value < 3 4 1)` is not. `lisp-value` is useful mostly for numerical comparisons (things that the logic system isn't so great at).

A note on writing rules: it's often tempting to think in terms of procedures – "this rule takes in so and so, and returns such and such". This is not the right way to approach these problems – remember, nothing is returned; an expression or query either has a match, or it doesn't. So often, you need to have both the "arguments" and the "return value" in the expression of the rule, and the rule is satisfied if "return value" is what would be returned if the rule were a normal Scheme procedure given the "arguments". Always keep in mind that everything is a Yes or No question, and your rule can only say if something is a correct answer or not. So when you write a rule, instead of trying to "build" to a solution like you've been doing in Scheme, think of it as trying to check if a given solution is correct or not.

In fact, **this is so important I'll say it again:** when you define rules, don't think of it as defining procedures in the traditional sense. Instead, think of it as, *given arguments and a proposed answer, check if the answer is correct.* The proposed answer can either be derived from the arguments, or it can't.

A different approach for writing declarative rules is to try to convert a Scheme program to a rule. For example, let's take a crack at the popular `append`:

```
(define (append ls1 ls2)
    (cond ((null? ls1) ls2)
          (else (cons (car ls1) (append (cdr ls1) ls2)))))
```

The `cond` specifies an "either-or" relationship; either `ls1` is null or it is not. This implies that, for logic programming, we'd need two separate rules, each corresponding to each `cond` clause. The first one is straightforward:

```
(rule (append () ?ls2 ?ls2))
```

The second `cond` clause breaks `ls1` into two parts – its `car` and its `cdr` – and basically says that appending `ls1` and `ls2` is the same as `cons`ing the first element of `ls1` to the list obtained by appending the `cdr` of `ls1` to `ls2`. Translated to logic programming, it means the `cdr` of the result is equivalent to `append`ing the `cdr` of `ls1` to `ls2`, and that the `car` of the result is just the `car` of `ls1`. This implies:

```
(rule (append (?car . ?cdr) ?ls2 (?car . ?r-cdr))
      (append ?cdr ?ls2 ?r-cdr))
```

We will try the above techniques on some of the harder problems below.

The Search For Truth and Honor (defined as such in our database)

Don't think declarative programming is useless – you might see it more than you think. The most common use of it is in database queries, and if you're worked with databases before, you know that SQL is a declarative language:

```
SELECT * FROM people WHERE age=13 ORDER BY first_name;
```

The above is a valid query into the table (or "relation") called "`people`" to select all columns of entries with column "`age`" equal to 13, and we want the result to be ordered by the column "`first_name`".

Note how we did *not* instruct to the database on how to give us the result; we simply trust the database management system to figure out the most efficient way to satisfy our desires. This is one place where declarative programming is absolutely natural (and think of the nightmare you'd have if you have to specify your queries imperatively!)

Of course, database queries is about the easiest use of logic programming, and you've had plenty of practice with `microshaft` (a rather obscene name if you ask me) in lab and homework. So let's move on to the interesting stuff.

Lists Again (and again, and again, and again, and again...)

Since lists are just patterns of symbols, logic programming is especially good at dealing with them.

```
(my-list (1 2 3 4))
```

Let's look at the following queries. Note the explanations – it's easy to see intuitively, sometimes, why something is bound to something, but you should get used to the way the query processor thinks!

`(my-list ?x)` => `?x` is bound to `(1 2 3 4)` because there's an entry in the database starting with "`my-list`" and followed by one more thing.

`(my-list (1 ?x 3 4))` => `?x` is bound to `2` because there's an entry in the database starting with "`my-list`" and followed by a list of four elements, the first, third and forth of which are `1`, `3` and `4`.

`(my-list (1 ?x))` => nothing. There is no entry in the database that starts with "`my-list`" followed by a two-element list – and, note carefully, `(1 ?x)` is a list of two elements!

`(my-list (1 . ?x))` => `?x` is bound to `(2 3 4)` because there's an entry in the database starting with "`my-list`" followed by a list whose first element is `1`. `?x` is simply bound to the rest. Note that this makes sense, since `(1 . (2 3 4))` is equivalent to `(1 2 3 4)`.

Now, let's play a few.

QUESTIONS

1. **Write a rule for `car` of list. For example, `(car (1 2 3 4) ?x)` would have `?x` bound to `1`.**

2. **Write a rule for `cdr` of list. For example, `(cdr (1 2 3) ?y)` would have `?y` bound to `(2 3)`.**

3. **Define our old friend, `member`, so that `(member 4 (1 2 3 4 5))` would be satisfied, and `(member 3 (4 5 6))` would not, and `(member 3 (1 2 (3 4) 5))` would not.**

4. **Define its cousin, `deep-member`, so that `(deep-member 3 (1 2 (3 4) 5))` would be satisfied as well.**