

Problem Set 10
Variations on a Scheme

Issued: Section 201 (Dimock): Monday, 06-Dec-2004, Section 202 (Yanco): Tuesday, 07-Dec-2004

Due: Section 201 (Dimock): Monday, 13-Dec-2004, Section 202 (Yanco): Tuesday, 14-Dec-2004

Reading: Covers text (SICP 2nd Edition by Abelson & Sussman): Through 4.2 with a nod to 4.3

What are interpreters good for?

In this problem set we look at several different interpreters for variations of Scheme. There are three reasons for studying interpreters in this course:

1. An interpreter for one language can be embedded in another language. (For example, Gambit, Chicken, and several other packages can be used as embedded Scheme interpreters.) By using an interpreter appropriate to your problem domain you take advantage of “meta-linguistic abstraction” to make your coding easier.
2. The Interpreters in Section 4.1 are for a subset of Scheme. The idea behind these interpreters is to show that the *process* of evaluating using the environment model can be coded as a *procedure*. By doing such coding, we should end up with a more precise understanding of the environment model. Of course, the code appears in the book so, in this case, studying the code rather than writing it is what helps us understand the environment model.

Begin rant mode. The fact that the concrete syntax of the language is easily changed should give you the intuition that a “language” can be independent of the details of syntax: most American school-children in the last generation learned (as play) a “language” called “Pig Latin” based on permuting and adding suffixes to English words. They could understand each other since Pig Latin was a minor change to the syntax of English words and the children involved all know English. So they were communicating in English with a minor syntactic variation. Similarly, we could rewrite Pascal as S-expressions and still understand it, if we know Pascal. There is a language called Standard ML that is basically Scheme + pattern matching + static type checking + Pascal-like syntax. If we wanted to learn that language, we would have to know something about pattern matching, and we would have to have some feel for how to get programs to type-check, but a knowledge of Scheme would carry us through most of the rest (up to that weasel word “basically”). *End rant mode.*

3. Interpreters are a way of performing rapid prototyping of language features: Take an interpreter for a language and add a few features that you are interested in studying. See if the combination makes sense.

In this problem set we are going to play around with the latter two reasons for using interpreters.

Optionally Lazy Scheme

Optionally lazy Scheme is a version of the Scheme subset evaluated by `mc-eval` but with an added feature.

Formal parameters for an abstraction can be either

- variables such as `x`. In this case any call to the procedure will evaluate the argument corresponding to `x` just as in the `mc-eval` subset of Scheme.
- The form `(y lazy)`. In this case the argument to be bound to `y` is evaluated lazily – “thunkified”, and the value is calculated whenever needed.
- The form `(z lazy-memo)`. In this case the argument to be bound to `z` is evaluated lazily, but its value is calculated the first time it is needed and that value is reused from then on: The value is never calculated another time.

Run the optionally lazy evaluator by loading `(load "olz-eval.ss")` and typing `(olz-eval-loop)` to run the evaluator. You can exit the evaluator by typing `quit` at the command prompt.

We will use the optionally lazy evaluator to explore some aspects of lazy evaluation and memoization.

Problem 1: Warmup

Look at the “Streams a lazy lists” code (online at http://mitpress.mit.edu/sicp/full-text/book/book-Z-H-27.html#%_sec_4.2.3)

Show that the definition of `cons`, `car`, and `cdr` really satisfy the contract for `cons`, `car`, and `cdr`, that is:

- use the substitution model to show that for Scheme, `(car (cons 1 2)) = 1`.
- use the environment model to show that for Scheme, `(cdr (cons 1 2)) = 2`. You need to show an environment diagram (which you need to submit with your written homework, but need not submit electronically). Show the term that you are evaluating in each environment.

Problem 2: Using the evaluator.

In Exercise 1.20, (problem set 2), you calculated the number of times that the greatest common divisor algorithm calculated a remainder in applicative-order evaluation, and normal-order evaluation.

```
(uml:define (gcd a b)
  (uml:if (uml:= b 0)
    a
    (gcd b
      (uml:rem a b))))
```

```
(gcd 206 40)
```

Take the code above and modify it using `uml:begin` and `uml:display` to display “rem ” before each use of `uml:rem`.

How many “rem ”s are displayed?

Now modify this, renaming it to `gcd2` and make it `lazy` in both parameters. How many “rem ”s are displayed?

Now modify this, renaming it to `gcd3` and make it `lazy-memo` in both parameters. How many “rem ”s are displayed?

Problem 3: Laziness and streams.

Copy the stream code from the beginning of Section 4.2.3. up through `add-lists` into your definitions buffer. You will find the code in a block-comment in the end of `olz-eval.ss`.

```
(uml:define chatty-integers
  (cons 0
    (uml:begin (uml:display "recurr ")
      (add-lists ones chatty-integers))))
```

Redefine `cons` so that its second parameter is `lazy`. Now try `(list-ref chatty-integers 4)`. How many times is “recurr ” printed?

Redefine `cons` so that both parameters are `lazy`. Re-execute your definitions. Try `(list-ref chatty-integers 4)`. How many times is “recurr ” printed?

Try with the second parameter of `cons` being `lazy-memo`.

Try with the both parameters of `cons` being `lazy-memo`.

What are the differences?

More importantly: why are there differences? What is being calculated redundantly in each of the 4 cases above. (If you need to hand-crank anything, you might use 3 rather than 4 in the `list-ref`).

Problem 4: (Optional)

In the preceding problem, do the results change (and if so how) if you make parameters to some of the other routines `car`, `cdr`, or `add-lists` `lazy` or `lazy-memo`.

Modifying the Analyzing Evaluator

In `mc-eval` we can implement the Scheme `let` keyword by the translation

```
(let ((<x1> <exp1>)          ((lambda (<x1> <x2> ... <xn>)
  (<x2> <exp2>)              <body>)
  ...                        => <exp1>
  (<xn> <expn>))            <exp2> ...
  <body>))                  <expn>)
```

Similar to the `cond->if` translation in `mc-eval`.

The problem with this translation is that `mc-eval` is supposed to be a procedure for the environment model, and our particular description of `let` in the environment model had it creating a frame, linking the frame to the environment, and creating some bindings in the frame. The translation above does all that, but with one difference: It creates a “double-bubble” linked back to the current environment for the anonymous `lambda` and then immediately throws away the double-bubble after use.

We can include `let` directly into `mc-eval`, in a way that does not create a “double bubble” by adding

```
...
((let? exp)
  (eval-sequence
    (let-body exp)
    (extend-environment (map let-var (let-bindings exp))
      (map let-val (let-bindings exp))
      env)))
...

```

into the `eval` code.

The translation method will fit into the analysis phase of the analyzing evaluator almost without modification.

Problem 5:

Instead of using the translation of `let` in the analyzing evaluator, try directly including `let` in the analyzing evaluator.

This will require writing a bit more code. Note that it is possible to write a `let` with no definitions before the body! (`let () <body>`)

You can get the analyzing evaluator code off the web from the page at <http://mitpress.mit.edu/sicp/code/index.tml>

Problem 6:

It is possible to define `let*` by rewriting.

```
(let* ((<x1> <exp1>)           (let ((<x1> <exp1>))
      (<x2> <exp2>)           (let ((<x2> <exp2>))
      ...                     ...
      (<xn> <expn>))          (let ((<xn> <expn>))
<body>)                      <body> ... ))
```

Define `let*` directly (without translating the syntax) in the analyzing evaluator.

The Unambiguous Evaluator

Section 4.3, which you may or may not have read, defines a new form for Scheme and makes significant changes to the structure of the analyzing evaluator to support the form. The part that interests us is that an analyzed expression is a procedure that never returns. Instead, each procedure produced by the analyze phase of the computation calls another procedure to continue the computation. In the Nondeterministic Evaluator of section 4.3, each function calls either `succeed` or `fail`, allowing back-tracking search to be implemented in the evaluator. We will just have each procedure produced by the analysis phase in the Unambiguous Evaluator call a routine `continue` to continue the evaluation. Why the name “Unambiguous Evaluator”? The answer is that by specifying how to continue the computation, we can override the preference of the underlying Scheme implementation for left-to-right or right-to-left computation.

(We can actually do a lot more: we can change the order of evaluation, from applicative order to normal order just by modifying the way continuations are passed, but the transformation to do so is not at all obvious. If you are interested in how this is done, and want to try to read a classic paper in programming language theory at the grad-school level, you can look at “Call-by-value, Call-by-name, and the Lambda Calculus” by Gordon Plotkin.)

You have actually seen a use of continuations in Problem Set 8 where `check-regularity` did not necessarily return, but instead called the routine passed to it as `receiver`.

The unambiguous evaluator is in file `unamb-eval.ss`

Problem 7: Change the analysis phase of the Unambiguous Evaluator so that

- evaluation of arguments to combinations of from right-to-left, and
- evaluate the operator after all of the arguments have been evaluated.