6.001 Tutorial 10 Notes

TA: Gerald Dalley
25–26 Apr 2005

- Problem set due Tuesday.
- Project 5 is due 6 May.
- The handout for this tutorial has changed. The preliminary version used examples that can be seen elsewhere, so they were changed at the last minute.
- Website additions:
  - *env-dia.zip:* Automatic environment diagram generator.
  - *assert.scm:* Assertion procedures that I find handy in finding and avoiding bugs.
  - *scm2html.pl:* A PERL script for doing syntax highlighting on Scheme source code.
  - Links to several free source code editors

  `http://people.csail.mit.edu/~dalleyg/6.001/`

1 Dynamic Scoping

Scheme uses what is called static or lexical scoping of variables: we look up variable bindings based on the *environment that existed when a procedure was created*. Static scoping is the most common form of scoping in modern programming languages. It is the only method directly available in languages such as Scheme, C++, Java, and Pascal.

An alternative form of scoping is called dynamic scoping. Dynamic scoping is much easier to implement. Variable lookup is done dynamically by using the current stack of procedure calls. Some languages only support dynamic scoping, such as JavaScript, TeX, and Postscript. Others such as Lisp and PERL support both dynamic and static scoping.

Consider the following Scheme code:

```
(define x 1)
(define (foo) (define x 2) (bar))
(define (bar) (display x))
(foo)
```

What value gets displayed?

What if we used dynamic scoping?

2 Meta-Circular Evaluator

The meta-circulator evaluator is an implementation of a Scheme evaluator, *in Scheme*. It's an important distinction — we've written Scheme code (the evaluator) that manipulates data (lists that look like Scheme expressions) to produce values. We could have written the evaluator in another language (like C). We could also have written an evaluator for another language (like Perl). The language the evaluator is written in and the language the evaluator evaluates are separate.

In our evaluator, we have expressions (lists, symbols, numbers, etc.) and environments. An expression is a piece of data, the same sort of data that the reader produces in the read, eval, print loop. An environment is a list of name-value bindings, along with a pointer to a parent environment.

The evaluator works by determining the type of the expression — is it a self-evaluating expression, a special form, etc.? Once it determines the type, it dispatches to an appropriate evaluation function for that type of expression. For most expression types, we have an evaluation rule that tells us how to evaluate the expression — now you can see the environment model rules in code!

3 Counting Procedure Calls

This fragment of evaluator code is used for creating and calling procedures:

```
(define (m-eval exp env)
  (cond ((self-evaluating? exp) exp) ...
        ((lambda? exp)
         (make-procedure (lambda-parameters exp) (lambda-body exp) env)) ...))

; double bubbles
(define (make-procedure parameters body env) (list 'procedure parameters body env))
(define (compound-procedure? exp)            (tagged-list? exp 'procedure))
(define (procedure-parameters p)             (second p))
(define (procedure-body p)                   (third p))
(define (procedure-environment p)            (fourth p))

(define (m-apply procedure arguments)
  (cond ((primitive-procedure? procedure)
         (apply-primitive-procedure procedure arguments))
        ((compound-procedure? procedure)
         (eval-sequence
          (procedure-body procedure)
          (extend-environment (procedure-parameters procedure)
                              arguments
                              (procedure-environment procedure))))
        (else (error "Unknown procedure type -- APPLY" procedure))))
```

For the purposes of debugging, we might want to add a way of counting how many times a particular procedure has been called. We would also like to add a special form `(times-called proc)` that returns the number of times which that procedure has been called. For example

```
(define (sqr x) (* x x)
(times-called sqr)   => 0
(sqr (sqr 4))        => 256
(times-called sqr)   => 2
(map sqr '(1 2 3 4)) => (1 4 6 9)
(times-called sqr)   => 6
```

How would we change the existing code to do this?
This is related to the implementation of `named-lambda` in Scheme.

4 Evaluating New Expressions

When we create new types of expressions (or change the meaning of existing ones), we need to add code to the evaluator to evaluate those expressions. There are two methods of doing this – write a new evaluation

function, and desugar the expression into something simpler.

4.1 Writing New Selectors

Let's work on handling `until` expressions. The first thing we need whenever we add a new expression type is procedures that handle the *syntax* of the new expression.

In the case of a `until`, we have the *test* and the *expressions*. An `until` expression has the following form:

```
(until (= x 0)      ; <- test
  (display x)       ; <- expressions
  (set! x (dec x)))
```

Fill in the following:

```
(define (until? exp)
  (tagged-list? exp 'until))

(define (until-test exp)


(define (until-expressions exp)
```

4.2 Creating a New Evaluation Function

After we've got the syntax procedures, we can write the function that actually evaluates the expression we're given. We also need to add a clause to the evaluator that recognizes the expression type and calls the appropriate evaluation function.

```
; in (m-eval exp env)
      ((until? exp) (eval-until exp env))
...

(define (eval-until exp env)
```

4.3 Desugar the Expression

Alternately, we can desugar the expression into another expression that we already know how to evaluate. In the case of a `let`, that's a `lambda` and a combination.

```
...
      ((until? exp) (m-eval (until->desugared exp) env))
...

(define (until->desugared exp)
```

Note: a desugaring for until is in project 5, but the specific desugaring we will use in tutorial will be different, using recursion.

You can make your desugaring functions much simpler if you use `quasiquote`. `quasiquote` is like `quote`, except that you can tell Scheme to evaluate parts of the quoted expression. A few examples:

```
‘(1 2 3) ; (1 2 3)

‘(1 (+ 1 1) 3) ; (1 (+ 1 1) 3)

‘(1 ,(+ 1 1) 3) ; (1 2 3)
```

The `,` is used to tell Scheme to “unquote” an expression – that is, to evaluate it.

```
‘(1 2 ,(list 3 4)) ; (1 2 (3 4))

‘(1 2 ,@(list 3 4)) ; (1 2 3 4)
```

The `,@` means “unquote” and “splice” – evaluate it, and the thing had better be a list, and then insert the contents of the list into the result.

Using quasiquote, we can write desugaring functions very easily.

```
(define (until->desugared exp)
```