

6.001 Tutorial 11

David Ziegler

November 22, 2004

1 Administrivia

- Problem set 9 is due tomorrow.
- Project 4 is due Wednesday (lucky you!).
- Project 5 is coming out on Wednesday, will be due Friday of next week.

2 Meta-Circular Evaluator

The meta-circular 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!

```
(define (eval-assignment exp env)
  (set-variable-value! (assignment-variable exp)
                       (m-eval (assignment-value exp) env)
                       env))

(define (set-variable-value! var val env)
  (if (eq? env the-empty-environment)

      (let* ((frame (first-frame env))
             (binding (find-in-frame var frame)))
        (if binding

            ))))

(define (set-binding-value! binding val)
  (set-car! (cdr binding) val))
```

3 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.

3.1 Writing New Selectors

Let's work on handling `let` 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 `let`, we've got the *names*, *expressions*, and *body*.

```
(define (let? exp)
```

```
(define (let-names exp)
```

```
(define (let-exprs exp)
```

```
(define (let-body exp)
```

3.2 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.

```
... ((let? exp) (eval-let exp env))  
...
```

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

3.3 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.

```
...  
    ((let? exp)  
...
```

```
(define (let->combination exp)
```

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 (let->combination exp)
```