Macros Review

May 2, 2018

Discussion questions

1. All Scheme expressions are either atomic expressions or combinations.

2. How are call expressions evaluated? (*Hint: There are three steps.)*
    1. Evaluate the operator
    2. Evaluate the operands
    3. Apply the operator to the operands

3. What do call expressions and special form expressions (e.g. `if`, `cond`, `define`) have in common?
    They are both lists/combinations

4. How are special form expressions evaluated? Describe the evaluation of the `if` special form (e.g. `(if (< 2 3) (+ 2 3) 'false)`). How is this different than evaluating a call expression?
    Special forms are evaluated differently according to the specific special form.
    In this case, we only evaluate the first and second operands, and we also do not evaluate "if".

Problems

Scheme expressions as data

What would Scheme display? Write **`Error`** if you think the expression would produce an error.

```
scm> (cons '+ (list 1 2))
(+ 1 2)

scm> (eval (list 'if #t 3 4)) → (eval (if #t 3 4))
3

scm> (list begin '(+ 1 2) '(- 3 4))
Error: Unknown identifier: begin

scm> (define exprs (list '(+ 1 2) '(if (> 2 3) 't 'f)))
exprs

scm> (eval (cons 'begin exprs))
f

scm> (cdr '(define (foo x) (+ x 1)))
((foo x) (+ x 1))
```

Macros vs. procedures

What would Scheme display?

```
scm> (define-macro (or-macro expr1 expr2)
         `(let ((v1 ,expr1)) (if v1 v1 ,expr2)))
or-macro
scm> (define (or-proc expr1 expr2)
         `(let ((v1 ,expr1)) (if v1 v1 ,expr2)))
or-proc
scm> (or-proc (> 2 3) (= 1 1))
(let ((v1 #f)) (if #f #f #t))

scm> (or-proc '(> 2 3) '(= 1 1))
(let ((v1 (> 2 3))) (if v1 v1 (= 1 1)))

scm> (or-macro (> 2 3) (= 1 1))
#t

scm> (eval (or-proc '(> 2 3) '(= 1 1)))
#t
```

Writing macros

1. Define a macro that takes in three expressions and evaluates each expression in order. If the last expression evaluates to a truth-y value, return the symbol ok. Otherwise, return `fail`.

```
scm> (eval-and-check #f #f #t)
ok
scm> (eval-and-check (+ 2 3) (print 2) (> 2 3))
2
fail
```

```
(define-macro (eval-and-check expr1 expr2 expr3)
      `(if (begin ,expr1 ,expr2 ,expr3)
           'ok 'fail))
```

2. Define a macro called `and-macro` that takes in two parameters `expr1` and `expr2` and has the same behavior as `(and expr1 expr 2)`. Do **not** use the and special form.

```
scm> (and-macro (< 2 3) (+ 2 3))
5
scm> (and-macro (null? '(1 2 3)) (/ 1 0))
#f
```

```
(define-macro (and-macro expr1 expr2)
      `(let ((v1 ,expr1))
            (if (not v1) v1 ,expr2)))
```

3. Define a macro called `define-if` that takes in an expression `pred`, a symbol `name`, a list of symbols `params`, and another expression `body`. If `pred` evaluates to a truth-y value, define a function with the given name, parameters, and body. Otherwise, define a function with the given name and parameters that returns `'nothing`.

```
scm> (define-if #t foo (x y) (+ x y))
foo
scm> (foo 1 2)
3
scm> (define-if #f bar (x y) (+ x y))
bar
scm> (bar 1 2)
nothing
```

```
(define-macro (define-if pred name params body)
      `(if ,pred
           (define ,(cons name params) ,body)
           (define ,(cons name params) 'nothing)))
```