CS 61A
Spring 2026

Macros
Discussion 11: April 22, 2026

Macros

A macro is a code transformation that is created using `define-macro` and applied using a call expression. A macro call is evaluated by:

1. Binding the formal parameters of the macro to the unevaluated operand expressions of the macro call.
2. Evaluating the body of the macro, which returns an expression.
3. Evaluating the expression returned by the macro in the environment of the original macro call.

```
scm> (define-macro (twice expr) (list 'begin expr expr))
twice
scm> (twice (+ 2 2))  ; evaluates (begin (+ 2 2) (+ 2 2))
4
scm> (twice (print (+ 2 2)))  ; evaluates (begin (print (+ 2 2)) (print (+ 2 2)))
4
4
```

Debugging tip: In order to see what expression a macro creates, change it to a regular procedure, then call it with quoted arguments.

```
scm> (define (twice expr) (list 'begin expr expr))  ; Same definition, but with define
    instead of define-macro
twice
scm> (twice '(print (+ 2 2)))                        ; Called with a quoted argument
(begin (print (+ 2 2)) (print (+ 2 2)))
scm> (eval (twice '(print (+ 2 2))))                 ; Evaluating the result has the same
    behavior as the original macro
4
4
```

Quasiquotation uses the backtick (below the tilde) to quote the next expression. Sub-expressions within the quasiquoted expression can be unquoted using a comma. Here are examples:

```
scm> (define x (+ 2 1))
x
scm> `(x ,x)
(x 3)
scm> (define s '(1 2 3))
s
scm> `(+ x ,(cons '* s))
(+ x (* 1 2 3))
```

Q1: Mystery Macro

Figure out what this `mystery-macro` does. Try to describe what it does by reading the code and discussing examples as a group.

```
(define-macro (mystery-macro expr old new)
    (mystery-helper expr old new))

(define (mystery-helper e o n)
  (if (and (list? e) (not (null? e)))
      (cons (mystery-helper (car e) o n) (mystery-helper (cdr e) o n))
      (if (eq? e o) n e)))
```

Here are some example uses of `mystery-macro` that could help you understand what it does and how it might be used.

```
scm> (define five 5)
five
scm> (mystery-macro (* x x) x five)
25
scm> (mystery-macro (* x x) x (+ five 1))
36
scm> (mystery-macro '(* x x) x y)
(* y y)
scm> (mystery-macro (> (x) (> (y) (+ x y))) > lambda)
(lambda (x) (lambda (y) (+ x y)))
scm> (mystery-macro (begin e e e) e (print five))
5
5
5
```

The `mystery-macro` replaces all instances of an `old` symbol with a `new` expression before evaluating the expression `expr`.

Q2: Multiple Assignment

In Scheme, the expression returned by a macro procedure is evaluated in the same environment in which the macro was called. Therefore, it's possible to return a `define` expression from a macro and have it affect the environment in which the macro was called. This differs from a regular scheme procedure that contains a `define` expression, which would only affect the procedure's local frame.

In Python, we can bind two names to values in one line as follows:

```
>>> x, y = 1 + 1, 3  # now x is bound to 2 and y is bound to 3
>>> x, y = y, x      # swap the values of x and y
>>> x
3
>>> y
2
```

Implement the `assign` Scheme macro, which takes in two symbols `sym1` and `sym2` as well as two expressions `expr1` and `expr2`. It should bind `sym1` to the value of `expr1` and `sym2` to the value of `expr2` in the environment from which the macro was called.

```
scm> (assign x y (+ 1 1) 3)  ; now x is bound to 2 and y is bound to 3
scm> (assign x y y x)        ; swap the values of x and y
scm> x
3
scm> y
2
```

Make sure that `expr2` is evaluated before `sym1` is changed. Assume that `expr1` and `expr2` do not have side effects (and so do not contain `define` or `assign` expressions).

```
(define-macro (assign sym1 sym2 expr1 expr2)
  `(begin
     (define ,sym1 ,expr1)
     (define ,sym2 ',(eval expr2))))

(assign x y (+ 1 1) 3)
(assign x y y x)
(expect x 3)
(expect y 2)

; Extra tests
(define z 'x)      ; z is bound to the symbol x
(assign v w 2 z)   ; now v is bound to 2 and w is bound to the symbol x
(assign v w w v)   ; swap the values of v and w
(expect v x)
(expect w 2)
```

Q3: Switch

Define the macro `switch`, which takes in an expression `expr` and a list of pairs called `cases` where the first element of the pair is some number and the second element is a single expression. `switch` will evaluate the expression contained in of `cases` that corresponds to the number that `expr` evaluates to.

```
scm> (switch (+ 1 1) ((1 (print 'a))
                      (2 (print 'b))
                      (3 (print 'c))))
b
```

You may assume that the value `expr` evaluates to is always a number and is always the first element of one of the pairs in `cases`. You can also assume that the first value of each pair in `cases` is a number and the second expression does not contain the symbol `val`.

Use `equal?` to check if two numbers are equal.

For the example shown above, build the following expression:

```
(begin
     (define val (+ 1 1))
     (cond ((equal? val 1) (print 'a))
           ((equal? val 2) (print 'b))
           ((equal? val 3) (print 'c))))
```

This expression first assigns `val` to 3 and then compares `val` to the first element in each pair in `cases`.

```
(define-macro (switch expr cases)
  `(begin
    (define val ,expr)
    ,(cons
    'cond
    (map (lambda (case) (cons
            `(equal? val ,(car case))
            (cdr case)))
        cases))))
```