CS 61A SQL

Spring 2025 Discussion 11: August 7, 2025

Switch to Pensieve:

- **Everyone**: Go to pensieve.co, log in with your @berkeley.edu email, and **enter your group number** as the room number (which was in the email that assigned you to this discussion). As long as you all enter the same number (any number), you'll all be using a shared document.

Once you're on Pensieve, you don't need to return to this page; Pensieve has all the same content (but more features). If for some reason Penseive doesn't work, return to this page and continue with the discussion.

Attendance

Fill out this discussion attendance form with the unique number you receive from your TA. As soon as you get your number, fill out the form, selecting *arrival* (not *departure* – that's later).

Getting Started

If there are fewer than 3 people in your group, feel free to merge your group with another group in the room.

Everybody say your name, and then your favorite kind of pie or cookie.

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 paramters 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 (pair? e)
      (cons (mystery-helper (car e) o n) (mystery-helper (cdr e) o n))
      (if (eq? e o) n e)))
```

Pro Tip: Please don't just look at the hints right away. Hints are for when you get stuck.

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 ___ ___)))

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

Call eval on expr2 so that its value is included in the define expression created by assign: ,(eval expr2). That way, the define for expr1 won't affect the value of expr2, because expr2 will already have been evaluated.

Presentation Time: Come up with a one-sentence explanation of why the second define line has to be different from the first define line in this implementation. Choose someone from your group who hasn't presented recently to say this explanation to your TA for feedback in person or on Zoom.

For an **optional extra challenge**, try these additional tests that make sure assign works correctly even when the value of expr2 is not a number, but instead a symbol.

```
(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)
```

In order to ensure that the value of expr2 is not evaluated a second time, quote the result of evaluating it.

For example, (assign v w 2 z) should be equivalent to:

```
(begin
  (define v 2)
  (define w (quote x)))
```

In this begin expression, (quote x) comes from first evaluating z and then quoting the result.

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 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:

```
(let ((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)
    `(let ((val ,expr))
      ,(cons
        'YOUR-CODE-HERE



        (map (lambda (case) (cons
               'YOUR-CODE-HERE



               (cdr case)))
             cases))))
```

Document the Occasion

Let your TA know you're done so that you can each get a **departure** number, and fill out the attendance form again (this time selecting *departure* instead of *arrival*). If your TA isn't in the room, go find them next door.

Note: This worksheet is a problem bank—most TAs will not cover all the problems in discussion section.