CS 61A
Fall 2023

Macros

Discussion 12: November 15, 2023

Getting Started

Someone from your group should join Discord.

To get help from a TA, send a message to the `discuss-queue` channel with the @discuss tag and your discussion group number.

If you have only 1 or 2 people in your group, you can join the other group in the room with you.

Q1: Ice Breaker

Everybody say your name, and then figure out who has the most elaborate Thanksgiving tradition. They're your group's facilitator for the day.

Facilitator: If you ever don't want to be facilitator anymore, just ask: "Would someone else be facilitator?" You can even do that now.

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 frame 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
```

Q2: Mystery Macro

Figure out what `mystery-macro` does.

```
(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)))
```

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

```
scm> (define a-very-long-symbol 5)
a-very-long-symbol
scm> (mystery-macro (+ (* x x) (* 2 x) 1) x a-very-long-symbol)
36
scm> (mystery-macro (+ (* x x) (* 2 x) 1) x (+ a-very-long-symbol 1))
49
scm> (mystery-macro '(+ (* x x) (* 2 x) 1) x y)
(+ (* y y) (* 2 y) 1)
scm> (mystery-macro (> (x) (> (y) (+ x y))) > lambda)
(lambda (x) (lambda (y) (+ x y)))
scm> (mystery-macro (begin e e e) e (print 5))
5
5
5
```

Pro tip: Please don't just look at the hint right away. Try to figure out what it does just by talking about the code as a group and discussing examples. Hints are for when you get stuck.

Facilitator: Ask your group, "What does `mystery-macro` do?" Once you all agree, work together to come up with an interesting use of this macro. Then send a message to the `#discuss-queue` channel with the `@discuss` tag, your discussion group number, and the message "We made some changes!". A member of the course staff will join your voice channel to hear you explain your use case and give feedback. The facilitator can decide who will present your explanation (or ask for volunteers).

Q3: Repeat

Define `repeat`, a macro that is called on a number `n` and an expression `expr`. Calling it evaluates `expr` in a local frame `n` times, and its value is the final result.

For example, `(repeat (+ 2 3) (print 1))` is equivalent to:

```
(repeated-call (+ 2 3) (lambda () (print 1)))
```

```
(define-macro (repeat n expr)
  `(repeated-call ,n ___))

; Call zero-argument procedure f n times and return the final result.
(define (repeated-call n f)
  (if (= n 1) ___ (begin ___ ___)))

(repeat (+ 1 2) (print 'three))       ; This should print three 3 times
(repeat 2 (repeat 2 (print 'four)))  ; This should print four 4 times
```

The `repeated-call` procedure takes a zero-argument procedure, so `(lambda () ___)` must appear in the blank. The body of the lambda is `expr`, which must be unquoted.

Call `f` on no arguments with `(f)`. If `n` is 1, just call `f`. If `n` is greater than 1, first call `f` and then call `(repeated-call (- n 1) f)`.

Discussion time: The `repeat` macro from lecture (*repeat*ed below) included `(eval n)`, while this version just has `n` in the body of the macro. Why does `n` not need to be evaluated in this implementation of `repeat`? If you talk about it for more than 5 minutes and still aren't sure, click the explanation below to find out.

In the version of `repeat` with `lambda`, `n` is part of the call to `repeated-call` returned from the macro, and so it will be evaluated automatically when that call expression is evaluated. For example, `(repeat (+ 2 3) (print 1))` returns `(repeated-call (+ 2 3) (lambda () (print 1)))`, and when this call is evaluated, the operand `(+ 2 3)` is evaluated to 5, which is passed to `repeated-call`.

In the lecture version of `repeat`, the `repeated-expr` procedure is being called within the macro, and so `n` will be passed as an expression unless it is evaluated using `(eval n)`.

Version of `repeat` from lecture that contains `(eval n)`:

```
; Evaluate expr n times and return the last value.
(define-macro (repeat n expr)
  (cons 'begin (repeated-expr (eval n) expr)))

; Return a list containing expr n times.
; scm> (repeated-expr 4 '(print 2))
; ((print 2) (print 2) (print 2) (print 2))
(define (repeated-expr n expr)
  (if (zero? n) nil (cons expr (repeated-expr (- n 1) expr))))
```

Facilitator: Ask your group, "What do you think of macros so far?" Together, write a sentence that starts with "Macros are …" in your group's Discord channel's text chat.

Q4: 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.

For an **optional extra challenge**, paste in 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.

Document the Occasion

Please all fill out the attendance form (one submission per person per week).

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