CS 61A Fall 2023

Macros

Discussion 12: November 15, 2023

Note: For extended explanations of the concepts on this discussion, feel free to look at the **Appendix** section on the back of the worksheet.

Macros

- Evaluate the operator to get a macro.
- Apply the macro to the unevaluated operands. This involves the following steps.
  - Bind the unevaluated operands to the formal parameters in a new frame.
  - Evaluate each expression in the body of the macro using normal Scheme evaluation rules.
  - The value of the last expression is returned.
- Evaluate the expression produced by the macro in the frame it was called in.

Q1: Shapeshifting Expressions

When writing macros in Scheme, the goal is to create a list of symbols that represents a certain Scheme expression. In this question, we'll practice different methods of creating such Scheme lists.

We have executed the following code to define `x` and `y` in our current environment.

```
(define x '(+ 1 1))
(define y '(+ 2 3))
```

We want to use `x` and `y` to build a list that represents the following expression:

```
(begin (+ 1 1) (+ 2 3))
```

What would be the result of calling `eval` on a quoted version of the expression above?

```
(eval '(begin (+ 1 1) (+ 2 3)))
```

5

Now that we know what this expression should evaluate to, let's build our scheme list.

How would we construct the scheme list for the expression `(begin (+ 1 1) (+ 2 3))` using quasiquotation?

```
`(begin ,x ,y)
```

How would we construct this scheme list using the `list` procedure?

```
(list 'begin x y)
```

How would we construct this scheme list using the `cons` procedure?

```
(cons 'begin (cons x (cons y nil)))
```

Q2: WWSD: Macros

For each expression, write what the Scheme interpreter would output.

```
scm> (define-macro (f x) (car x))
```

f

```
scm> (f (2 3 4))
```

2

```
scm> (define x 2000)
```

x

```
scm> (f (x y z))
```

2000

```
scm> (define-macro (g x) (+ x 2))
```

g

```
scm> (g 2)
```

4

```
scm> (g (+ 2 3))
```

SchemeError

```
scm> (define-macro (h x) (list '+ x 2))
```

h

```
scm> (h (+ 2 3))
```

7

```
scm> (define-macro (if-else-5 condition consequent) `(if ,condition ,consequent 5))
```

if-else-5

```
scm> (if-else-5 #f (/ 1 0))
```

5

```
scm> (if-else-5 (= 1 1) 2)
```

2

Q3: Mystery Macro

For this question, we'll consider the following macro:

```
(define-macro (mystery expr)
    `(let ((/ (lambda (a b) (if (= b 0) 1 (/ a b))))) ,expr))
```

What does this macro do?

The macro evaluates the given `expr`, except that all division is replaced with a "safe division" operation so that division by 0 does not give an error.

Why can't we do the same thing with a regular procedure?

```
(define (mystery-proc expr)
    ... )
```

For a procedure, `expr` will be evaluated before `mystery-proc` is applied - so if there are any divisions by zero, they won't be caught by `mystery-proc`.

Professor Oppenheimer has written a procedure `letter-grade` to determine a student's letter grade on an assignment given a number of points `earned` and a number of points `possible`.

```
(define (letter-grade earned possible)
    (cond
        ((>= (/ earned possible) 0.9) 'A)
        ((>= (/ earned possible) 0.8) 'B)
        ((>= (/ earned possible) 0.7) 'C)
        ((>= (/ earned possible) 0.6) 'D)
        (else 'F)))
```

The procedure works well, but Professor Oppenheimer has noticed that some optional assignments have 0 points possible, which causes a division by zero error. Professor Oppenheimer wants to define `letter-grade` such that all students will receive an A for assignments with 0 points possible. Can you help him out using `mystery`?

```
(define-macro (mystery expr)
    `(let ((/ (lambda (a b) (if (= b 0) 1 (/ a b))))) ,expr))

(define (letter-grade earned possible)
    (mystery
        (cond
            ((>= (/ earned possible) 0.9) 'A)
            ((>= (/ earned possible) 0.8) 'B)
            ((>= (/ earned possible) 0.7) 'C)
            ((>= (/ earned possible) 0.6) 'D)
            (else 'F)))
    )

; Tests
(expect (letter-grade 100 0) A)
(expect (letter-grade 95 100) A)
(expect (letter-grade 85 100) B)
(expect (letter-grade 75 100) C)
(expect (letter-grade 65 100) D)
(expect (letter-grade 55 100) F)
```

Q4: Max Macro

Define the macro `max`, which takes in two expressions `expr1` and `expr2` and returns the maximum of their values. If they have the same value, return the value of the first expression. **For this question, it's okay if your solution evaluates `expr1` and `expr2` more than once.** As an extra challenge, think about how you could use the `let` special form to ensure that `expr1` and `expr2` are evaluated only once.

```
scm> (max 5 10)
10
scm> (max 12 12)
12
scm> (max 100 99)
100
```

First, try using quasiquotation to implement this macro procedure.

```
(define-macro (max expr1 expr2)
    `(if (>= ,expr1 ,expr2) ,expr1 ,expr2)
    )

; Test
(expect (max -3 (+ 1 2)) 3)
(expect (max 1 1) 1)
```

Now, try writing this macro using `list`.

```
(define-macro (max expr1 expr2)
    (list 'if (list '>= expr1 expr2) expr1 expr2)
    )

; Test
(expect (max -3 (+ 1 2)) 3)
(expect (max 1 1) 1)
```

Finally, write this macro using `cons`.

```
(define-macro (max expr1 expr2)
    (cons 'if
          (cons
                (cons '>= (cons expr1 (cons expr2 nil)))
                (cons expr1 (cons expr2 nil))
          )
    )
    )

; Test
(expect (max -3 (+ 1 2)) 3)
(expect (max 1 1) 1)
```

Reflect: was it necessary to use macros to do this? Or could we have done the same thing with a procedure?

It was not necessary to use macros. We could have defined a procedure that does the same thing:

```
(define (max a b)
    (if (>= a b) a b))
```

Q5: Multiple Assignment

Recall that in Scheme, the expression returned by a macro procedure is evaluated in the environment that called the macro. This concept allows us to set variables in the calling environment using calls to macro procedures! This is not possible with regular scheme procedures because any `define` expressions would be evaluated in the procedure's environment (and thus bind a symbol in that environment rather than the calling environment). In this problem, we'll explore this idea in more detail.

In Python, we can bind two variables in one line as follows:

```
>>> x, y = 1, 2
>>> x
1
>>> y
2
>>> x, y = y, x # swap the values of x and y
>>> x
2
>>> y
1
```

The expressions on the right of the assignment are first evaluated, then assigned to the variables on the left. Let's try to implement a similar feature in Scheme using macros.

Write a macro `multi-assign` 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> (multi-assign x y 1 (- 3 1))
scm> x
1
scm> y
2
```

First, implement a version of `multi-assign` which evaluates `expr1` first, binds it to `sym1`, then evaluates `expr2` and binds it to `sym2` (the order here is important).

```
(define-macro (multi-assign sym1 sym2 expr1 expr2)
    `(begin (define ,sym1 ,expr1) (define ,sym2 ,expr2) undefined)
)

; Tests
(multi-assign x y 1 2)
(expect (= x 1) #t)
(expect (= y 2) #t)
```

Note that we use undefined as the last expression in the `begin` form so that nothing is output to the terminal.

This solution is great, but it doesn't quite behave in quite the same way that it does in Python:

```
scm> (multi-assign x y 1 (+ 2 3))
scm> x
1
scm> y
5
scm> (multi-assign x y y x)
scm> x
5
scm> y
5
```

Notice that `x` and `y` were not swapped like we wanted. This is because of the order of evaluation and bindings: first, the value of `y` is bound to `x`. Afterwards, `x` is evaluated and bound to `y`, but at this point, `x` no longer has its old value, it is actually the value of `y`!

Now, try writing a version of `multi-assign` which matches the behavior in Python, i.e. `expr1` and `expr2` should be both evaluated before being assigned to `sym1` and `sym2`.

```
scm> (multi-assign x y 5 6)
scm> x
5
scm> y
6
scm> (multi-assign x y y x)
scm> x
6
scm> y
5
```

```
(define-macro (multi-assign sym1 sym2 expr1 expr2)
    `(begin (define ,sym2 (list ,expr1 ,expr2))
            (define ,sym1 (car ,sym2))
            (define ,sym2 (car (cdr ,sym2)))
            undefined)
)

; Tests
(multi-assign x y 1 2)
(expect (= x 1) #t)
(expect (= y 2) #t)
(multi-assign x y y x)
(expect (= x 2) #t)
(expect (= y 1) #t)
```

The idea behind this solution is to make `sym2` hold the values of *both* `expr1` and `expr2`, so that even after `sym1` gets bound to the value of `expr1`, `sym2` will have access to the previous value of `expr2` (even if `expr2` happens to be `sym1` itself, as it is in the variable swapping case). We can achieve this by binding `sym2` to a list containing the values of `expr1` and `expr2`!

To understand the motivation behind this somewhat strange looking solution, let's take a look at a different, and perhaps more intuitive, way to write this macro:

```
(define-macro (multi-assign sym1 sym2 expr1 expr2)
    `(begin (define val1 ,expr1) (define val2 ,expr2)
            (define ,sym1 val1) (define ,sym2 val2) undefined))
```

This achieves the desired behavior for the most part. It improves upon the solution in the first part of the problem by evaluating `expr1` and `expr2` before assigning them to `sym1` and `sym2`, allowing us to do variable swapping. However, there's one issue with this solution which is that it uses `val1` and `val2` to store the values of `expr1` and `expr2` in the current environment. If these symbols have already been defined in this environment (and were perhaps being used for something else), they will be overwritten!

Therefore, we need to write `multi-assign` so that it only binds symbols that were passed into the macro.

Q6: Replace

Write the macro `replace`, which takes in a Scheme expression `expr`, a Scheme symbol or number `old`, and a Scheme expression `new`. The macro replaces all instances of `old` with `new` before running the modified code.

```
(define (replace-helper e o n)
  (if (pair? e)
      (cons (replace-helper (car e) o n) (replace-helper (cdr e) o n))
      (if (eq? e o) n e)))
(define-macro (replace expr old new)
    (replace-helper expr old new))

; Tests
(expect (replace (define x 2) x y) y)
(expect (= y 2) #t)
(expect (replace (+ 1 2 (or 2 3)) 2 0) 1)
```

Appendix: Explanation of Material

Macros

So far we've been able to define our own procedures in Scheme using the `define` special form. This doesn't allow us to do anything, however. Consider what happens when we try to write a function that evaluates a given expression twice:

```
scm> (define (twice expr) (begin expr expr))
twice
scm> (twice (print 'woof))
woof
```

Since `twice` is a procedure, we evaluate its call expression by first evaluating the operator and then each operand. That means that `woof` was printed when we evaluated the operand `(print 'woof)`. Inside the body of `twice`, the name `expr` is bound to the value `undefined`, so the expression `(begin expr expr)` does nothing at all!

The problem here is clear: we need to prevent the given expression from evaluating until we're inside the body of the procedure. Wouldn't it be cool if we could define our own special forms where we could avoid such the pitfalls of call expressions? This is where the `define-macro` special form, which has identical syntax to the regular `define` form, comes in:

```
scm> (define-macro (twice expr) (list 'begin expr expr))
twice
```

`define-macro` allows us to define what's known as a `macro`, which is simply a way for us to process unevaluated input expressions together into another expression. The rules for evaluating a macro expression are:

- Evaluate the operator to get a macro.
- Apply the macro to the unevaluated operands. This involves the following steps.
  - Bind the unevaluated operands to the formal parameters in a new frame.
  - Evaluate each expression in the body of the macro using normal Scheme evaluation rules.
  - The value of the last expression is returned.
- Evaluate the expression produced by the macro in the frame it was called in.

Note the key differences here between macros and regular procedures: the operands are not evaluated before being passed in to the macro. Additionally, the output of the body of the macro is evaluated after.

To evaluate `(twice (print 'woof))`:

- We evaluate `twice`, which evaluates to a macro procedure.
- We apply that macro procedure to the unevaluated operands:
  - Bind `(print 'woof)` to `expr` in a new frame.
  - Evaluate the body of the macro: `(list 'begin expr expr)` evaluates to `(begin (print 'woof) (print 'woof))`.
  - Return `(begin (print 'woof) (print 'woof))`
- Evaluate `(begin (print 'woof) (print 'woof))` in the current frame. `(print 'woof)` is evaluated twice, and woof is printed twice.