Macros

By Chris Allsman from cs61a

Review: Representing Expressions

Representing Expressions

- In Scheme, we can create lists that "look like" combinations
  - In fact, in Scheme, expressions *are* lists (or primitive values)

- Quoting prevents evaluation of an expression

- Calling eval on an unevaluated expression will evaluate that value

```
scm> '(+ 1 2)
(+ 1 2)
scm> (eval '(+ 1 2))
3
```

```
scm> (list 'quotient 10 2)
(quotient 10 2)
scm> (eval (list 'quotient 10 2))
5
```

Expressions In Scheme

Expressions As Data

Recall: programs are composed of expressions, but manipulate values or data

In Scheme, expressions are either primitive expressions or lists - which means they're also a form of data!

This means we can:

- Assign expressions to variables
- Pass expressions into functions
- Create & return new expressions within functions

Begin

begin is a special form takes in any number of expressions, evaluates them in order, and evaluates to the value of the final expression

```
(begin 3 2 1)
1

scm> (begin (define x 2) (define x (+ x 1)) x)
3
```

Let

```
(let ((symbol1 expr1)

      (symbol2 expr2)

      …)

      body)
```

Each symbol is bound to the value of the expression in parallel

Evaluate to the value of the body using the binding

The bindings only exist when evaluating the body

```
scm> (let ((x 2)
           (y 3))
          (+ x y))
5
```

Macros

Example: Double

Let's write a procedure double. We want it to evaluate whatever expression we pass in twice.

```
scm> (double (print 2))
2
2
```

Issues:
- How do we prevent evaluation of the input?
- How do we easily get the intended behavior?

Macros

Macros are a more convenient way to transform or create expressions

The **define-macro** special form will create a macro procedure

Macros take in and return expressions, which are then evaluated **in place of** the call to the macro

A piece of code that hasn't been evaluated

Equivalent to:
(begin (print 2) (print 2))

```scheme
(define-macro (twice expr)
  (list 'begin expr expr))
```

Returns a piece of code that then gets evaluated

```
scm> (twice (print 2))
2
2
```

Evaluating Macros

Recall evaluation procedure used for regular call expressions:

1. Evaluate the operator sub-expression, which evaluates to a regular procedure.
2. Evaluate the operand expressions in order.
3. Apply the procedure to the evaluated operands.

Macros, on the other hand, do the following:

1. Evaluate the operator sub-expression, which evaluates to a macro procedure.
2. Apply the macro procedure to the operand expressions _without evaluating them first_.
3. Evaluate the expression returned by the macro procedure in the frame the macro was called in

Writing Macros

Because macros take in and return expressions, when writing macros you should think about:

1) What types of expressions you'll take in
2) What expression has equivalent behavior to your macro

Consider a macro **add-to** which should take in a symbol and an expression, and increment the value of the variable by the expression.

```
scm> (define x 1)
scm> (add-to x (+ 1 2))
x
scm> x
4
```

What's the equivalent expression?

For Macro

Scheme doesn't have for loops, but thanks to macros, we can add them.

```
scm> (for x in '(1 2 3 4) do (* x x))
(1 4 9 16)
scm> (map (lambda (x) (* x x)) '(1 2 3 4))
(1 4 9 16)



(define-macro (for sym in vals do expr)
  (list 'map (list 'lambda (list sym) expr) vals))
```

Quasi-Quotation

Quasi-quoting

Quasiquotation allows you to have some parts of a list be read literally and some parts be evaluated.
It's especially useful for constructing code in macros.

```
(define-macro (for sym vals expr)
  (list 'map (list 'lambda (list sym) expr) vals))

(define-macro (for sym vals expr)
  `(map (lambda (,sym) ,expr) ,vals))
```

Short for (quasiquote …)

Short for (unquote …)

Much cleaner, right?

You Try:

Write the twice and add-to macros using quasiquotes

```scheme
(define-macro (twice expr)
  (list 'begin expr expr))

(define-macro (add-to sym expr)
  (list 'define sym (list '+ sym expr)))
```