

61A Lecture 29

Announcements

Programs as Data

A Scheme Expression is a Scheme List

Scheme programs consist of expressions, which can be:

- Primitive expressions: 2 3.3 true + quotient
- Combinations: (quotient 10 2) (not true)

The built-in Scheme list data structure (which is a linked list) can represent combinations

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

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

In such a language, it is straightforward to write a program that writes a program

(Demo)

```
scm> (+ 1 2)
3
scm> (list + 1 2)
(#[+] 1 2)
scm> (list '+ 1 2)
(+ 1 2)
scm> (+ 1 2)
3
scm> (list '+ 1 (+ 2 3))
(+ 1 5)
```

```
(define (fact n)
  (if (= n 0) 1 (* n (fact (- n 1)))))

(define (fact-exp n)
  (if (= n 0) 1 (list '* n (fact-exp (- n 1)))))
```

```
scm> (fact 5)
120
scm> (fact-exp 5)
(* 5 (* 4 (* 3 (* 2 (* 1 1)))))
scm> (eval (fact-exp 5))
120
```

```
(define (fib n)
  (if (<= n 1) n (+ (fib (- n 2)) (fib (- n 1)))))

(define (fib-exp n)
  (if (<= n 1) n (list '+ (fib-exp (- n 2)) (fib-exp (- n 1)))))
```

```
scm> (fib 6)
8
scm> (fib-exp 6)
(+ (+ (+ 0 1) (+ 1 (+ 0 1))) (+ (+ 1 (+ 0 1)) (+ (+ 0 1) (+ 1 (+ 0 1)))))
```

Macros

Macros Perform Code Transformations

A macro is an operation performed on the source code of a program before evaluation

Macros exist in many languages, but are easiest to define correctly in a language like Lisp

Scheme has a **define-macro** special form that defines a source code transformation

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

> (twice (print 2)) ► (begin (print 2) (print 2))
2
2

Evaluation procedure of a macro call expression:

- Evaluate the operator sub-expression, which evaluates to a macro
- Call the macro procedure on the operand expressions without evaluating them first
- Evaluate the expression returned from the macro procedure

(Demo)

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

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

```
2
```

```
(begin None None) The value of (print 2) is evaluated and shown instead.
```

```
scm> (twice '(print 2)) To not evaluate it, a quote can be used.
```

```
(begin (print 2) (print 2))
```

```
scm> (eval (twice '(print 2))) However, eval is still needed to finish the work.
```

```
2
```

```
2
```

Using define-macro can save the job.

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

```
twice
```

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

```
2
```

```
2
```


Now let's see another application.

```
scm> (define (check val) (if val 'passed 'failed))
check
scm> (define x -2)
x
scm> (check (> x 0))
failed
```

We want check to tell what expression is failed. -> define-macro

```
scm> (check (> x 0))
(failed: (> x 0))
```

```
(define-macro (check expr) eval('passed) -> 'passed
  (list 'if expr 'passed
        (list 'quote (list 'failed: expr)))))
```

Here we return a list of expression.

If you want to see what expressions define-macro creates, just delete -macro and quote the arg.

```
(define (check expr)
  (list 'if expr 'passed
        (list 'quote (list 'failed: expr)))))
```

```
scm> (check '(> x 0))
(if (> x 0) (quote passed) (quote (failed: (> x 0))))
scm> (if (> x 0) (quote passed) (quote (failed: (> x 0))))
(failed: (> x 0))
```

You have to evaluate it to get the result.

For Macro

Discussion Question

Define a macro that evaluates an expression for each value in a sequence

```
(define (map fn vals)
  (if (null? vals)
      ()
      (cons (fn (car vals))
             (map fn (cdr vals)))))
```

```
scm> (map (lambda (x) (* x x)) '(2 3 4 5))
(4 9 16 25)
```

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

```
scm> (for x '(2 3 4 5) (* x x))
(4 9 16 25)
```

(Demo)

Quasi-Quotation

(Demo)

```

scm> (define b 2)
b
scm> '(a b c)
(a b c)
scm> `(a b c)
(a b c)
scm> `(a ,b c)  unquote part of the expression
(a 2 c)
scm> `(a ,(+ b 5) c)
(a 7 c)
scm> `(a b ,c)
Traceback (most recent call last):
  0      (quasiquote (a b (unquote c)))
  1      c
Error: unknown identifier: c
scm> '(a ,b c)
(a (unquote b) c)  Quote cannot do so.
scm> (define expr '(* x x))
expr
scm> `(lambda (x) ,expr)  Quasi-quote makes it more convenient to create some expressions.
(lambda (x) (* x x))

```

We can use quasi-quote to rewrite check, so that it is more readable.

```

(define-macro (check expr)
  (list 'if expr 'passed
        (list 'quote (list 'failed: expr))))

(define-macro (check expr)
  `(if ,expr 'passed '(failed: ,expr)))

```