CS61A: Scheme Basic

Lin Sheng

January 2022

1 Scheme fundamentals

Scheme programs consist of expressions, which can be primitive expressions and combinations.

Numbers are self-evaluating; symbols are bound to values.

Call expressions include an operator and 0 or more operands in parentheses.

Note: Scheme interpreter doesn't care about indentation at all. Combinations can span multiple lines (spacing doesn't matter)

```
> (quotient 10 2)
5
> (quotient (+ 8 7) 5)
3
> (+ (* 3
        (+ (* 2 4)
            (+ 3 5)))
        (+ (- 10 7)
            6))
```

2 Special forms

Special form is a combination that is not a call expression.

- **if** expression: (if ¡predicate¿ ¡consequent¿ ¡alternative¿)
- **and** and **or**: (and <e1> <e2> ...) (or <e1> <e2> ...)
- Building symbols: (define <symbol> <expression>)
- New procedures: (define (<symbol> <formal parameters>) <body>)
- **begin**: do all of the things after it

3 Lambda expressions

Lambda expressions evaluate to anonymous procedures.

(lambda (<formal-parameters>) <body>)

4 Pairs and lists

- **cons**: Two-arguments procedure that creates a pair
- **car**: Procedure that returns the first element of a pair
- **cdr**: Procedure that returns the second element of a pair
- **nil**: The empty list

Important! Scheme lists are written in parentheses separated by spaces.

A dotted list has some value for the second element of the last pair that is not a list. It doesn't have a recursive structure.

```
> (cons 1 (cons 2 nil))
(1 2)
> (define x (cons 1 2)) # 2 is not a list!
> x
(1 . 2) # not well-formed list!
> (car x)
1
> (cdr x)
2
> (cons 1 (cons 2 3))
(1 2 . 3)
> (define y (cons 1 (cons 2 nil)))
> y
(1 2)
> (cdr y)
(2)
> (cdr (cdr y))
()
> (list 1 2 3 4)
(1 2 3 4)
> (pair? (list 1 2 3 4))
True
```

5 Symbolic programming

Symbols normally refer to values; Quotation is used to refer to symbols directly in Lisp.

```
> (define a 1)
> (define b 2)
> (list a b)
(1 2)
> (list 'a 'b)
```

```
(a b)
> (list 'a b)
(a 2)
```

Quotation can also be applied to combinations to form lists.

```
> (car '(a b c))
a
> (cdr '(a b c))
(b c)
```

Dots can be used in a quoted list to specify the second element of the final pair. However, dots appear in the output only of ill-formed lists.

```
> (cdr (cdr '(1 2 . 3)))
3
> '(1 2 . 3)
(1 2 . 3)
> '(1 2 . (3 4))
(1 2 3 4)
```

6 Interpreters

A scheme list is written as elements in parentheses: (`<element_0>` `<element_1>` ... `<element_n>`). Each element can be a combination or primitive.

The task of parsing a language involves coercing a string representation of an expression to the expression itself. Parsers must validate that expressions are well-formed.

7 Tail calls

A procedure call that has not yet returned is active. Some procedure calls are tail calls. A Scheme interpreter should support an unbounded number of active tail calls using only a constant amount of space.

A tail call is a call expression in a tail context:

- The last body sub-expression in a lambda expression
- Sub-expression 2 & 3 in a tail context **if** expression
- All non-predicate sub-expressions in a tail context **cond**
- The last sub-expression in a tail context **and** or **or**
- The last sub-expression in a tail context **begin**

A call expression is not a tail call if more computation is still required in the calling procedure.

Linear recursive procedures can often be re-written to use tail calls.

8 Reduce and map

8.1 Reduce

```
(define (reduce procedure s start)
  (if (null? s) start
    (reduce procedure
      (cdr s)
      (procedure start (car s)))))
```

It uses **procedure** to combine every element iteratively in **s** with **start** as the beginning.

8.2 Map

```
(define (map procedure s)
  (if (null? s)
    nil
    (cons (procedure (car s))
        (map procedure (cdr s)))))
```

map is a function that applies a procedure to every element in a list and construct a list containing all the results.

Tail call version:

```
(define (map procedure s)
  (define (map-reverse s m)
    (if (null? s)
      m
      (map-reverse (cdr s)
            (cons (procedure (car s))
                m))))
  (reverse (map-reverse s nil)))
```

9 Macros

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

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

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

> (twice (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

9.1 For macro

```
(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)
```

10 Quasi-quotation

Quasi-quoting can choose to selectively unquote certain parts of the expression that was quoted.

```
scm> '(a b c)
(a b c)
scm> `(a b c)
(a b c)
scm> `(a ,b c)
(a 2 c)
scm> '(a ,b c)
(a (unquote b) c)
scm> (define expr '(* x x))
scm> `(lambda (x) ,expr)
(lambda (x) (* x x))

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

11 Streams

A stream is a list, but the rest of the list is computed only when needed.

```
(car (cons-stream 1 2)) -> 1
(cdr-stream (cons-stream 1 2)) -> 2
(cons 1 (/ 1 0)) -> Error
(cons-stream 1 (/ 1 0)) -> (1 . #[delayed])
(car (cons 1 (/ 1 0))) -> Error
(car (cons-stream 1 (/ 1 0))) -> 1
(cdr (cons 1 (/ 1 0))) -> Error
(cdr-stream (cons-stream 1 (/ 1 0))) -> Error
```

Errors only occur when expressions are evaluated.

A stream can be infinitely long. An integer stream is a stream of consecutive integers. The rest of the stream is not yet computed when the stream is created.

```
(define (int-stream start)
    (cons-stream start (int-stream (+ start 1))))
(define (square s)
    (cons-stream (* (car s) (car s)) (square (cdr-stream s))))
```

11.1 Promises

A promise is an expression, along with an environment in which to evaluate it. Delaying an expression creates a promise to evaluate it later in the current environment. Forcing a promise returns its value in the environment in which it was defined.

```
scm> (define promise (let ((x 2)) (delay (+ x 1))))
scm> (define x 5)
scm> (force promise)
3
```

```
(define-macro (delay expr) `(lambda () ,expr))
(define (force promise) (promise))
(define-macro (cons-stream a b) `(cons ,a (delay ,b)))
(define (cdr-stream s) (force (cdr s)))
```

A stream is a list, but the rest of the list is computed only when forced.