Michael Kennedy

CS 301 -- Lab 2

Scheme Journal

Chapter 1:

==================================================================

Section 1.1

==================================================================

Scheme is made up of keywords, variables, structure forms, data, whitespace, and comments

Keywords anv variables are called Identifiers.

Characters used:

\* a-z, A-Z, 0-9, and characters.

Identifiers can't start with the @ symbol, and normally can't start with character that

may start with a number.

True and False booleans are written as #t and #f.

Lists are written: (a b c)

Vctors: #(name)

Strings: "String"

Char: #\x

Comments; ';'

==================================================================

Section 1.2

==================================================================

\* Predicate names end in a question mark (?). Predicate names are procedures that return

a true or false answer.

\* Type predicates, e.g. 'pair?' are created from the name of the type, which in this case

is 'pair'

\* The names of most char, string, and vector procecdures start with prefixes such as 'char-',

'string-', and 'vector-' (e.g. 'string-append').

\* Names of procedures that convert an object from one type into an object of another type is

written as type1->type2.

\* Procedures that end with a side effect end with '!'

Questions:

1. What's an example of converting an object from one type into an object of another type? As in

converting an int- to a string-?

2. What does "side effects" mean?

==================================================================

Section 1.3

==================================================================

\* The value of a procedure is said to be unspecefied\*.

Question: What does that even mean?

Chapter 2:

==================================================================

Section 2.1

==================================================================

Scheme follows a 'Read-evaluate-print' cycle, aka REPL. Perhaps similar to Python.

You can write from the keyboard and get the expression back immediately.

To save for later use:

\* transcript-on/off

Lists can contain more than one object, so an int and a string can be in a single list.

'=>' means "evaluates to"

==================================================================

Section 2.2

==================================================================

Scheme expressions are constant data objects.

\* Numbers are constant. Scheme echoes numbers back. (e.g +1/2 1/2 returns 1)

Scheme provides names for arithmatic proceduress: '+, -, \*, /'

Each procedure accepts 2 arguments. Arithmatic procedures take 2 arguments.

'cdr' and 'car' returns the first element of a list and the rest of the elements of a list

respectively.

'cons' constructs a list.

Exercise 2.2.1

A. 1.2 \* (2-1/3) + -8.7 -> (+ (\* 1/2 ( - 2 1/3)) -8.7)

Answer: -6.6999999999

B. (2/3 + 4/9) / (5/11 - 4/3) -> (/ (+ 2/3 4/9) (- 5/11 4/3))

Answer: -1 23/87

C. 1 + 1 / (2 + 1 / (1 + 1/2)) -> (+ 1 (/ 1 (+ 2 (/ 1 (+ 1 1/2)))))

Answer: 1 3/8

D. 1 \* -2 \* 3 \* -4 \* 5 \* -6 \* 7 = (\* 1 (\* -2 (\* 3 (\* -4 (\* 5 (\* -6 7))))))

Answer: -5040

Exercise 2.2.2

(+ 2 3/4) = 2 3/4

(+ 2 3/4 5) = 7 3/4

Exercise 2.2.3

A. '(car . cdr)

B. '(this (is silly))

C. '(is silly)

D. '(+ 2 3)

E. '(+ 2 3)

F. '+

G. '(2 3)

H. #<procedure:cons>

I. 'cons

J. '' cons

K. 'quote

L. 5

M. 5

N. 5

O. 5

Exercise 2.2.4

(car (car '((b a) (c d)))) = b

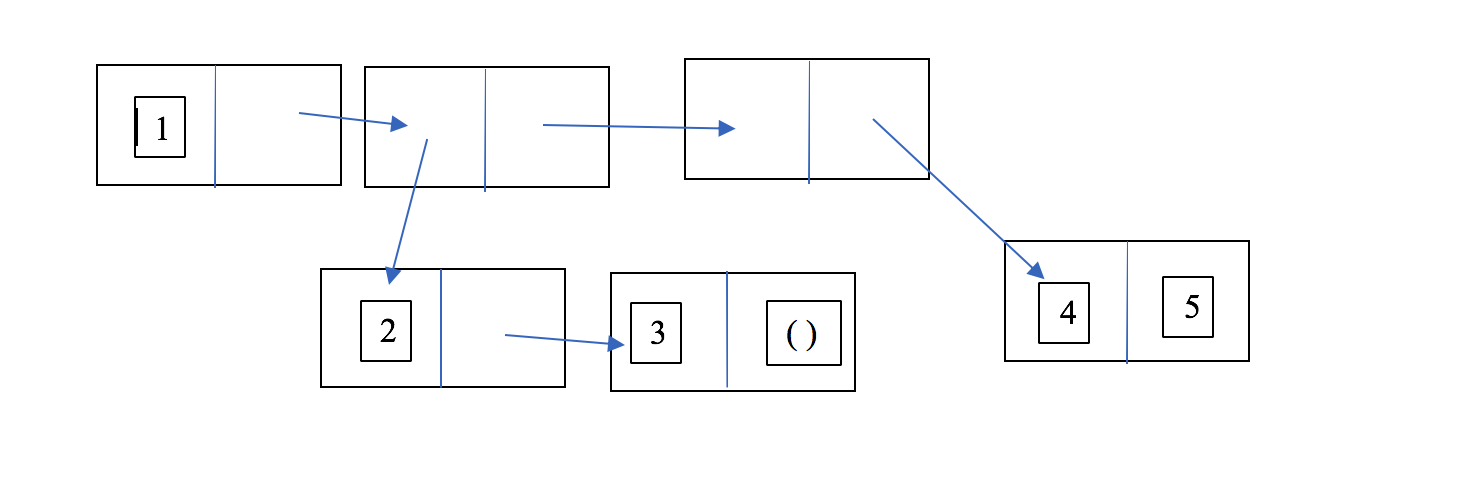
(car (car '((c d) (a b)))) = c

(car (car '((d c) (a b)))) = d

Exercise 2.2.5

(list (cons 'a 'b) (cons (cons 'c '()) (cons 'd '())) (cons '() '()))

Exercise 2.2.6



Exercise 2.2.7

(car '((a b) (c d)))

(cdr '((a b) (c d)))

(car(car '((a b) (c d))))

(car (cdr ‘((a b) (c d))))

(car (cdr ‘((a b) (c d))))

(cdr (cdr ‘((a b) (c d))))

Exercise 2.2.8

Scheme reads, evaluates, then prints.

==================================================================

Section 2.3

==================================================================

Scheme evaluates things in a pretty straightforward fashion. For example, it processes (+ 3 4) like it applies the addition operation (+) to 3 and 4.

Exercise 2.3.1

First, a list of + - \* / is created. ***Cdr*** grabs every element except the first element in that list, which in this case are the symbols + - \* /. Then, of the elements selected with ***cdr, car*** grabs the first element of that list, which in this case is the subtraction symbol. This makes the procedure (- 17 5) which results in 12.

==================================================================

Section 2.4

==================================================================

To assign values to variables use ***let*** (i.e. (let ((x 2))(+ x 3)) → 5 )

You can also assign other mathematical operations with ***let*** (i.e. (let ((+ \*)) (+ 2 3)) → 6)

Exercise 2.4.1

1. (let ((a 1) (b 2))  
    (let ((x (\* a 3)))  
    (+ (- x b) (+ x b))))
2. (let ((x (list a b c)))  
    (cons (car x) (cdr x)))

Exercise 2.4.2

The program assigns 9 to x, re-assigns 9/3 to x, multiplies 9 and 3, and finally adds 27 to itself resulting in 54.

Exercise 2.4.3

1. Results in ‘((c . b) (a . d)) – because the ***let*** statements only change within the scope of the parenthesis. In this case, the ***let*** statements only changes x to c in the scope of that statement. The second ***let*** only changes the value of y to be d.
2. Results in '(c b a b). This block grabs the last value of the list (a b) c (which in this case is c) and assigns c to x as a new value of a list. Next, it grabs the first element of the list which is (a b), then the function grabs the last element of that list. Then after that, it grabs the first element of the list (a b) – a, and lastly it grabs the last element of (a b) – b to form ‘(c b a b).

==================================================================

Section 2.5

==================================================================

I can use ***lambda*** to create a new procedure that has x as a parameter in the same body as a ***let*** expression.

­So really it’s basically another way of completing a procedure.

(let ((double (lambda (x) (+ x x))))  
  (list (double (\* 3 4))  
        (double (/ 99 11))  
        (double (- 2 7)))) <graphic> (24 18 -10)

This function specifically completes some mathematical operation then uses the lambda to execute yet another operation. It looks like it loops. This function creates a function called ‘double’ which takes in the result of (\* 3 4) or 12 and doubles that using lambda. It then repeats that operation for each item in the list.

Lambda can take any kind of input.

Exercise 2.5.1

1. This just prints out ‘a.
2. This returns ‘(a)
3. This returns ‘a.
4. This returns ‘() because it only prints out x which is an empty value.

Exercise 2.5.2

A list creates an ordered set of elements that don’t necessarily have to be related.

Exercise 2.5.3

1. Both x’s are printed out, and the function f takes in 2 arguments.
2. This takes in 2 integer values and adds them together.
3. This function takes in 2 parameters and prints them out twice.

==================================================================

Section 2.6

==================================================================

In order to have expressions visible outside of let and lambda, you use ***define***. Similar to declaring variables in a parent function. This can be used to declare primitive types or it can be used to create functions.

Exercise 2.6.1

This function would recursively call double-any.

==================================================================

Section 2.7

==================================================================

Conditional expressions using if. For example defining a function ***abs*** that finds the absolute value of an integer value.

(define abs  
  (lambda (n)  
    (if (< n 0)  
        (- 0 n)  
        n)))

Exercise 2.7.1

(define atom?

(lambda (x y)

(= x y)))

==================================================================

Section 2.8

==================================================================

Recursive procedures act on itself until a condition is met to return. That’s all there is to it, it’s not rocket science. Okay maybe it is rocket science but you get the idea.

==================================================================

Section 2.9

==================================================================

Use of top-level variables is useful. You can set those through ***define*** or ***set.***

==================================================================

Section 2.9

==================================================================

The core syntactic forms include top-level define forms, constants, variables, procedure applications, quote expressions, lambda expressions, if expressions, and set! expressions. The grammar below describes the core syntax of Scheme in terms of these definitions and expressions. In the grammar, vertical bars (|) separate alternatives, and a form followed by an asterisk (\*) represents zero or more occurrences of the form. <variable> is any Scheme identifier. <datum> is any Scheme object, such as a number, list, symbol, or vector. <boolean> is either #t or #f, <number> is any number, <character> is any character, and <string> is any string. We have already seen examples of numbers, strings, lists, symbols, and booleans.

|  |  |  |
| --- | --- | --- |
| <program> |  | <form>\* |
| <form> |  | <definition> | <expression> |
| <definition> |  | <variable definition> | (begin <definition>\*) |
| <variable definition> |  | (define <variable> <expression>) |
| <expression> |  | <constant> |
|  | | | <variable> |
|  | | | (quote <datum>) |
|  | | | (lambda <formals> <expression> <expression>\*) |
|  | | | (if <expression> <expression> <expression>) |
|  | | | (set! <variable> <expression>) |
|  | | | <application> |
| <constant> |  | <boolean> | <number> | <character> | <string> |
| <formals> |  | <variable> |
|  | | | (<variable>\*) |
|  | | | (<variable> <variable>\* . <variable>) |
| <application> |  | (<expression> <expression>\*) |

==================================================================

Section 3.1

==================================================================

The let syntactic form is merely a *syntactic extension* defined in terms of a lambda expression and a procedure application, both core syntactic forms. At this point, you might be wondering which syntactic forms are core forms and which are syntactic extensions, and how new syntactic extensions may be defined. This section provides some answers to these questions.

(begin *e1* *e2* ...)

is equivalent to the lambda application

((lambda () *e1* *e2* ...))

Identifiers appearing within a pattern are *pattern variables*, unless they are listed as auxiliary keywords. Pattern variables match any substructure and are bound to that substructure within the corresponding template.

The definition of and below is somewhat more complex than the one for let.

(define-syntax and  
  (syntax-rules ()  
    ((\_) #t)  
    ((\_ e) e)  
    ((\_ e1 e2 e3 ...)  
     (if e1 (and e2 e3 ...) #f))))

This definition is recursive and involves more than one rule. Recall that (and) evaluates to #t; the first rule takes care of this case. The second and third rules specify the base case and recursion steps of the recursion and together translate and expressions with two or more subexpressions into nested if expressions. For example, (and a b c) expands first into

(if a (and b c) #f)

then

(if a (if b (and c) #f) #f)

and finally

(if a (if b c #f) #f)

With this expansion, if a and b evaluate to a true value, then the value is the value of c, otherwise #f, as desired.

==================================================================

Section 3.2

==================================================================

Like let, the letrec syntactic form includes a set of variable-value pairs, along with a sequence of expressions referred to as the *body* of the letrec.

(letrec ((*var* *val*) ...) *exp1* *exp2* ...)

Unlike let, the variables *var* ... are visible not only within the body of the letrec but also within *val* .... Thus, we can rewrite the expression above as follows.

(letrec ((sum (lambda (ls)  
                (if (null? ls)  
                    0  
                    (+ (car ls) (sum (cdr ls)))))))  
  (sum '(1 2 3 4 5))) <graphic> 15

Using letrec, we can also define mutually recursive procedures, such as the procedures even? and odd? that were the subject of Exercise [2.8.6](https://www.scheme.com/tspl3/start.html#g38).

(letrec ((even?  
          (lambda (x)  
            (or (= x 0)  
                (odd? (- x 1)))))  
         (odd?  
          (lambda (x)  
            (and (not (= x 0))  
                 (even? (- x 1))))))  
  (list (even? 20) (odd? 20))) <graphic> (#t #f)

Exercise 3.2.1

Factorial and Fibonacci are tail-end recursive

==================================================================

Section 3.3

==================================================================

During the evaluation of a Scheme expression, the implementation must keep track of two things: (1) what to evaluate and (2) what to do with the value. Consider the evaluation of (null? x) within the expression below.

(if (null? x) (quote ()) (cdr x))

The implementation must first evaluate (null? x) and, based on its value, evaluate either (quote ()) or (cdr x). "What to evaluate" is (null? x), and "what to do with the value" is to make the decision which of (quote ()) and (cdr x) to evaluate and to do so. We call "what to do with the value" the *continuation* of a computation.

(define product  
  (lambda (ls)  
    (call/cc  
      (lambda (break)  
        (let f ((ls ls))  
          (cond  
            ((null? ls) 1)  
            ((= (car ls) 0) (break 0))  
            (else (\* (car ls) (f (cdr ls))))))))))

(product '(1 2 3 4 5))  120  
(product '(7 3 8 0 1 9 5))  0

The nonlocal exit allows product to return immediately, without performing the pending multiplications, when a zero value is detected.

Exercise 3.3.1

(let ((k.n (call/cc (lambda (k) (cons k 0)))))  
  (let ((k (car k.n)) (n (cdr k.n)))  
    (write n)  
    (newline)  
    (k (cons k (+ n 1)))))

Exercise 3.3.2

(define product  
  (lambda (ls)  
    (if (null? ls)  
        1  
        (if (= (car ls) 0)  
            0  
            (let ((n (product (cdr ls))))  
              (if (= n 0) 0 (\* n (car ls))))))))

==================================================================

Section 3.4

==================================================================

When one procedure invokes another via a nontail call, the called procedure receives an implicit continuation that is responsible for completing what is left of the calling procedure's body plus returning to the calling procedure's continuation. If the call is a tail call, the called procedure simply receives the continuation of the calling procedure.

(letrec ((f (lambda (x) (cons 'a x)))  
         (g (lambda (x) (cons 'b (f x))))  
         (h (lambda (x) (g (cons 'c x)))))  
  (cons 'd (h '()))) <graphic> (d b a c)

conses the symbol b onto the value returned to it, then returns the result of this cons to the continuation of the call to g. This continuation is the same as the continuation of the call to h, which conses the symbol d onto the value returned to it. We can rewrite this in *continuation-passing style*, or CPS, by replacing these implicit continuations with explicit procedures.

(letrec ((f (lambda (x k) (k (cons 'a x))))  
         (g (lambda (x k)  
              (f x (lambda (v) (k (cons 'b v))))))  
         (h (lambda (x k) (g (cons 'c x) k))))  
  (h '() (lambda (v) (cons 'd v))))

The procedure quotient, employed by integer-divide, returns the quotient of its two arguments, truncated toward zero.

Exercise 3.4.1

(define reciprocal  
  (lambda (n success failure)  
    (if (= n 0)  
        ‘()  
        (success (/ 1 n)))))

Exercise 3.4.2

(define retry #f)   
  
(define factorial  
  (lambda (x)  
    (let f ((x x) (k (lambda (x) x)))  
      (if (= x 0)  
          (begin (set! retry k) (k 1))  
          (f (- x 1) (lambda (y) (k (\* x y))))))))

==================================================================

Section 3.5

==================================================================

Definitions may also appear at the front of a lambda, let, or letrec body, in which case the bindings they create are local to the body.

(define f (lambda (x) (\* x x)))  
(let ((x 3))  
  (define f (lambda (y) (+ y x)))

Procedures bound by internal definitions can be mutually recursive, as with letrec

(let ()  
  (define even?  
    (lambda (x)  
      (or (= x 0)  
          (odd? (- x 1)))))  
  (define odd?  
    (lambda (x)  
      (and (not (= x 0))  
           (even? (- x 1)))))

In fact, internal definitions and letrec are practically interchangeable. It should not be surprising, therefore, that a lambda, let, or letrec body containing internal definitions can be replaced with an equivalent letrec expression.

(define *var* *val*)  
  <graphic>  
*exp1*  
*exp2*  
  <graphic>

is equivalent to a letrec expression binding the defined variables to the associated values in a body comprising the expressions.

(letrec ((*var* *val*) ...) *exp1* *exp2* ...)

Syntax definitions may also appear at the front of a lambda, let, or letrec body.

(let ((x 3))  
  (define-syntax set-x!  
    (syntax-rules ()  
      ((\_ e) (set! x e))))  
  (set-x! (+ x x))

Exercise 3.5.1

(define-syntax complain  
  (syntax-rules ()  
    ((\_ ek msg exp) (ek (list msg exp)))))

Exercise 3.5.3

(define calc #f)  
(let ()  
  (define do-calc  
    (lambda (exp)  
      (cond  
        ((number? exp) exp)  
        ((and (list? exp) (= (length exp) 3))  
         (let ((op (car exp)) (args (cdr exp)))  
           (case op  
             ((add) (apply-op + args))  
             ((sub) (apply-op - args))  
             ((mul) (apply-op \* args))  
             ((div) (apply-op / args))  
             (else (complain "invalid operator" op)))))  
        (else (complain "invalid expression" exp)))))  
  (define apply-op  
    (lambda (op args)  
      (op (do-calc (car args)) (do-calc (cadr args)))))  
  (define complain  
    (lambda (msg exp)  
      (error 'calc "~a ~s" msg exp)))  
  (set! calc  
    (lambda (exp)  
      (do-calc exp))))

==================================================================

Section 4.1

==================================================================

Any identifier appearing as an expression in a program is a keyword or variable reference.

list <graphic> #<procedure>  
(define x 'a)  
(list x x) <graphic> (a a)  
(let ((x 'b))  
  (list x x)) <graphic> (b b)  
(let ((let 'let)) let) <graphic> let

It is an error to evaluate a top-level variable reference before the variable is defined at top-level, but it is not an error for such a reference to appear within a part of a program that has not yet been evaluated. This permits mutually recursive procedures to be defined using top-level bindings.

i-am-not-defined <graphic> *error*   
  
(define f  
  (lambda (x)  
    (g x)))  
(define g  
  (lambda (x)  
    (+ x x)))  
(f 3) <graphic> 6

==================================================================

Section 4.2

==================================================================

The lambda syntactic form is used to create procedures. Any operation that creates a procedure or establishes local variable bindings is ultimately defined in terms of lambda.

* If *formals* is a proper list of variables, e.g., (x y z), each variable is bound to the corresponding actual parameter. It is an error if too few or too many actual parameters are supplied.
* If *formals* is a single variable (not in a list), e.g., z, it is bound to a list of the actual parameters.
* If *formals* is an improper list of variables terminated by a variable, e.g., (x y . z), each variable but the last is bound to the corresponding actual parameter. The last variable is bound to a list of the remaining actual parameters. It is an error if too few actual parameters are supplied.

(lambda (x) (+ x 3)) <graphic> #<procedure>

==================================================================

Section 4.2 Local binding

==================================================================

**syntax**: (let ((*var* *val*) ...) *exp1* *exp2* ...)

let establishes local variable bindings. Each variable *var* is bound to the value of the corresponding expression *val*. The body of the let, in which the variables are bound, is the sequence of expressions *exp1* *exp2* ....

The forms let, let\*, and letrec (let\* and letrec are described after let) are similar but serve slightly different purposes. With let, in contrast with let\* and letrec, the expressions *val* ... are all outside the scope of the variables *var* .... Also, in contrast with let\*, no ordering is implied for the evaluation of the expressions *val* .... They may be evaluated from left to right, from right to left, or in any other order at the discretion of the implementation.

(define-syntax let  
  (syntax-rules ()  
    ((\_ ((x v) ...) e1 e2 ...)  
     ((lambda (x ...) e1 e2 ...) v ...))))   
  
(let ((x (\* 3.0 3.0)) (y (\* 4.0 4.0)))  
  (sqrt (+ x y))) <graphic> 5.0   
  
(let ((x 'a) (y '(b c)))  
  (cons x y)) <graphic> (a b c)   
  
(let ((x 0) (y 1))  
  (let ((x y) (y x))  
    (list x y))) <graphic> (1 0)

(define-syntax let\*  
  (syntax-rules ()  
    ((\_ () e1 e2 ...)  
     (let () e1 e2 ...))  
    ((\_ ((x1 v1) (x2 v2) ...) e1 e2 ...)  
     (let ((x1 v1))  
       (let\* ((x2 v2) ...) e1 e2 ...)))))

(letrec ((*var* *val*) ...) *body*)

==================================================================

Section 4.4 – Variable Definitions

==================================================================

In the first form, define creates a new binding of *var* to the value of *exp*. The remaining are shorthand forms for binding variables to procedures; they are identical to the following definition in terms of lambda.

(define *var*  
  (lambda *formals*  
    *exp1* *exp2* ...))

==================================================================

Section 5.1 – Procedure Application

==================================================================

**syntax**: (*procedure* *exp* ...)

(+ 3 4) <graphic> 7

**procedure**: (apply *procedure* *obj* ... *list*)

(apply + '(4 5)) <graphic> 9

(apply min '(6 8 3 2 5)) <graphic> 2

(apply vector 'a 'b '(c d e)) <graphic> #5(a b c d e)

==================================================================

Section 5.2 – Sequencing

==================================================================

**syntax**: (begin *exp1* *exp2* ...)

(let ()  
  (begin (define x 3) (define y 4))  
  (+ x y)) <graphic> 7

==================================================================

Section 5.3 – Conditionals

==================================================================

let ((l '(a b c)))  
  (if (null? l)  
      '()  
      (cdr l))) <graphic> (b c)   
  
(let ((l '()))  
  (if (null? l)  
      '()  
      (cdr l))) <graphic> ()   
  
(let ((abs  
       (lambda (x)  
         (if (< x 0)  
             (- 0 x)  
             x))))  
  (abs -4)) <graphic> 4   
  
(let ((x -4))  
  (if (< x 0)  
      (list 'minus (- 0 x))  
      (list 'plus 4))) <graphic> (minus 4)

(not #f) <graphic> #t  
(not #t) <graphic> #f  
(not '()) <graphic> #f  
(not (< 4 5)) <graphic> #f

(let ((x 3))  
  (or (< x 2) (> x 4))) <graphic> #f   
  
(let ((x 5))  
  (or (< x 2) (> x 4))) <graphic> #t   
  
(or #f '(a b) '(c d)) <graphic> (a b)

(let ((x 0))  
  (cond  
    ((< x 0) (list 'minus (abs x)))  
    ((> x 0) (list 'plus x))  
    (else (list 'zero x)))) <graphic> (zero 0)

(let ((x 4) (y 5))  
  (case (+ x y)  
    ((1 3 5 7 9) 'odd)  
    ((0 2 4 6 8) 'even)  
    (else 'out-of-range))) <graphic> odd

==================================================================

Section 5.4 – Iteration and Mapping

==================================================================

This form of let, called *named* let, is a general-purpose iteration and recursion construct. It is similar to the more common form of let (see Section [4.3](https://www.scheme.com/tspl3/binding.html#g87)) in the binding of the variables *var* ... to the values *val* ... within the body *exp1* *exp2* .... In addition, the variable *name* is bound within the body to a procedure that may be called to recur or iterate; the arguments to the procedure become the new values of the variables *var* ....

(let *name* ((*var* *val*) ...)  
  *exp1* *exp2* ...)

((letrec ((*name* (lambda (*var* ...) *exp1* *exp2* ...)))  
   *name*)  
 *val* ...)

The procedure divisors defined below uses named let to compute the nontrivial divisors of a nonnegative integer.

(define divisors  
  (lambda (n)  
    (let f ((i 2))  
      (cond  
        ((>= i n) '())  
        ((integer? (/ n i))  
         (cons i (f (+ i 1))))  
        (else (f (+ i 1)))))))

do allows a common restricted form of iteration to be expressed succinctly. The variables *var* ... are bound initially to the values of *val* ... and are rebound on each subsequent iteration to the values of *update* .... The expressions *test*, *update* ..., *exp* ..., and *res* ... are all within the scope of the bindings established for *var* ....

(define factorial  
  (lambda (n)  
    (do ((i n (- i 1)) (a 1 (\* a i)))  
        ((zero? i) a))))

(define fibonacci  
  (lambda (n)  
    (if (= n 0)  
        0  
        (do ((i n (- i 1)) (a1 1 (+ a1 a2)) (a2 0 a1))  
            ((= i 1) a1)))))

map applies *procedure* to corresponding elements of the lists *list1* *list2* ... and returns a list of the resulting values. The lists *list1* *list2* ... must be of the same length, and*procedure* must accept as many arguments as there are lists.

(map abs '(1 -2 3 -4 5 -6)) <graphic> (1 2 3 4 5 6)  
(map (lambda (x y) (\* x y))  
     '(1 2 3 4)  
     '(8 7 6 5)) <graphic> (8 14 18 20)

for-each is similar to map except that for-each does not create and return a list of the resulting values, and for-each guarantees to perform the applications in sequence over the lists from left to right. for-each may be defined as follows.

(define for-each  
  (lambda (f ls . more)  
    (do ((ls ls (cdr ls)) (more more (map cdr more)))  
        ((null? ls))  
        (apply f (car ls) (map car more)))))

==================================================================

Section 5.5 – Continuations

==================================================================

Continuations in Scheme are procedures that represent the remainder of a computation from a given point in the continuation. They may be obtained with call-with-current-continuation, which can be abbreviated call/cc in many Scheme implementations.

f *procedure* returns normally when passed the continuation procedure, the value returned by call/cc is the value returned by *procedure*.

Continuations allow the implementation of nonlocal exits, backtracking [[9](https://www.scheme.com/tspl3/bibliography.html#g189),[21](https://www.scheme.com/tspl3/bibliography.html#g201)], coroutines [[11](https://www.scheme.com/tspl3/bibliography.html#g191)], and multitasking [[6](https://www.scheme.com/tspl3/bibliography.html#g186),[24](https://www.scheme.com/tspl3/bibliography.html#g204)].

The example below illustrates the use of a continuation to perform a nonlocal exit from a loop.

(define member  
  (lambda (x ls)  
    (call/cc  
      (lambda (break)  
        (do ((ls ls (cdr ls)))  
            ((null? ls) #f)  
            (if (equal? x (car ls))  
                (break ls)))))))   
  
(member 'd '(a b c)) <graphic> #f  
(member 'b '(a b c)) <graphic> (b c)

dynamic-wind offers "protection" from continuation invocation. It is useful for performing tasks that must be performed whenever control enters or leaves *body*, either normally or by continuation application.

let ((p (open-input-file "input-file")))  
  (dynamic-wind  
    (lambda () #f)  
    (lambda () (process p))  
    (lambda () (close-input-port p))))

Some Scheme implementations support a controlled form of assignment known as *fluid binding*, in which a variable takes on a temporary value during a given computation and reverts to the old value after the computation has completed. The syntactic form fluid-let defined below in terms of dynamic-wind permits the fluid binding of a single variable x to a value v within a sequence of expressions e1 e2 ....

(define-syntax fluid-let  
  (syntax-rules ()  
    ((\_ ((x v)) e1 e2 ...)  
     (let ((y v))  
       (let ((swap (lambda () (let ((t x)) (set! x y) (set! y t)))))  
         (dynamic-wind swap (lambda () e1 e2 ...) swap))))))

==================================================================

Section 5.6 – Delayed Evaluation

==================================================================

The syntactic form delay and the procedure force may be used in combination to implement *lazy evaluation*. An expression subject to lazy evaluation is not evaluated until its value is required and once evaluated is never reevaluated. delay and force appear in the Revised5 Report but not in the ANSI/IEEE standard.

define-syntax delay  
  (syntax-rules ()  
    ((\_ exp) (make-promise (lambda () exp)))))

where make-promise is defined as

(define make-promise  
  (lambda (p)  
    (let ((val #f) (set? #f))  
      (lambda ()  
        (if (not set?)  
            (let ((x (p)))  
              (if (not set?)  
                  (begin (set! val x)  
                         (set! set? #t)))))  
        val))))

The benefit of using delay and force is that some amount of computation might be avoided altogether if it is delayed until absolutely required. Delayed evaluation may be used to construct conceptually infinite lists, or *streams*. The example below shows how a stream abstraction may be built with delay and force. A stream is a promise that, when forced, returns a pair whose cdr is a stream.

(define stream-car  
  (lambda (s)  
    (car (force s))))   
  
(define stream-cdr  
  (lambda (s)  
    (cdr (force s))))   
  
(define counters  
  (let next ((n 1))  
    (delay (cons n (next (+ n 1))))))

==================================================================

Section 5.7 – Multiple Values

==================================================================

(values 1 2 3) <graphic> 1  
               /var/folders/rk/14wfbd996h34t6g64dwsvtxh0000gn/T/com.microsoft.Word/WebArchiveCopyPasteTempFiles/ghostRightarrow.gif 2  
               /var/folders/rk/14wfbd996h34t6g64dwsvtxh0000gn/T/com.microsoft.Word/WebArchiveCopyPasteTempFiles/ghostRightarrow.gif 3

(define head&tail  
  (lambda (ls)  
    (values (car ls) (cdr ls))))   
  
(head&tail '(a b c)) <graphic> a  
                     /var/folders/rk/14wfbd996h34t6g64dwsvtxh0000gn/T/com.microsoft.Word/WebArchiveCopyPasteTempFiles/ghostRightarrow.gif (b c)

*producer* and *consumer* must be procedures. call-with-values applies *consumer* to the values returned by invoking *producer* without arguments.

(call-with-values  
  (lambda () (values 'bond 'james))  
  (lambda (x y) (cons y x))) <graphic> (james . bond)   
  
(call-with-values values list) <graphic> '()

The example below employs multiple values to divide a list nondestructively into two sublists of alternating elements.

(define split  
  (lambda (ls)  
    (if (or (null? ls) (null? (cdr ls)))  
        (values ls '())  
        (call-with-values  
          (lambda () (split (cddr ls)))  
          (lambda (odds evens)  
            (values (cons (car ls) odds)  
                    (cons (cadr ls) evens)))))))   
  
(split '(a b c d e f)) <graphic> (a c e)  
                       /var/folders/rk/14wfbd996h34t6g64dwsvtxh0000gn/T/com.microsoft.Word/WebArchiveCopyPasteTempFiles/ghostRightarrow.gif (b d f)

The continuation of a call to values need not be one established by a call to call-with-values, nor must only values be used to return to a continuation established by call-with-values. In particular, (values *v*) and *v* are equivalent in all situations. For example:

(+ (values 2) 4) <graphic> 6   
  
(if (values #t) 1 2) <graphic> 1   
  
(call-with-values  
  (lambda () 4)  
  (lambda (x) x)) <graphic> 4

Similarly, since there is no requirement to signal an error when the wrong number of arguments is passed to a procedure (although most implementations do so), the behavior of each of the following expressions is also unspecified.

(call-with-values  
  (lambda () (values 2 3 4))  
  (lambda (x y) x))   
  
(call-with-values  
  (lambda () (call/cc (lambda (k) (k 0))))  
  (lambda (x y) x))

==================================================================

Section 5.8 – Eval

==================================================================

*env-spec* must be an environment specifier returned by interaction-environment, scheme-report-environment, or null-environment. eval treats *obj* as the representation of an expression. It evaluates the expression in the specified environment and returns its value.

(define cons 'not-cons)  
(eval '(let ((x 3)) (cons x 4))  
      (scheme-report-environment 5)) <graphic> (3 . 4)

(define lambda 'not-lambda)  
(eval '(lambda (x) x) (null-environment)) <graphic> #<procedure>   
  
(eval '(cons 3 4) (null-environment)) <graphic> *error*