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

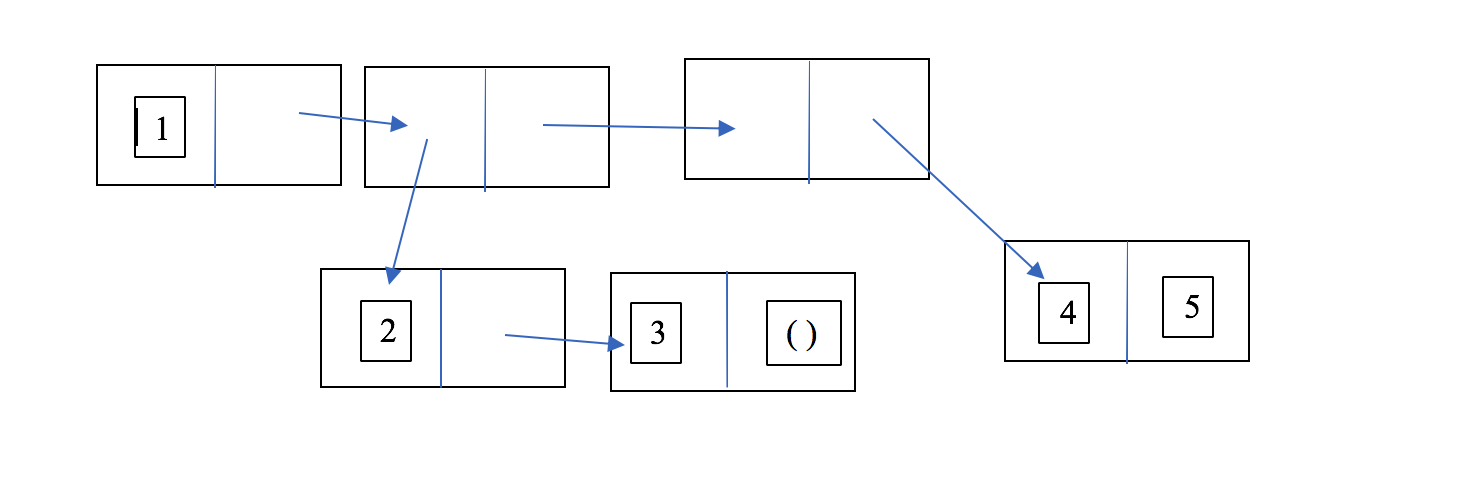
(car (car '((c d) (a b)))) = c

(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. See Chapter [6](https://www.scheme.com/tspl3/objects.html#g99) or the more detailed grammar at the back of this book for more on the object-level syntax of these and other objects.

|  |  |  |
| --- | --- | --- |
| <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

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