Introductory Scheme

Revision 1

Joseph W. Lavinus and James D. Arthur
(lavinus@cs.vt.edu and arthur@cs.vt.edu)
Department of Computer Science
Virginia Polytechnic Institute and State University
Blacksburg, Virginia 24060

May 6, 1993

Abstract

This document is an introductory guide to applicative programming using the language Scheme, a modern dialect of Lisp. It assumes familiarity with computers and with basic programming techniques, but no knowledge of Lisp or applicative programming. This manual describes a purely functional subset of Scheme, the use of the XScheme interpreter, and some style guidelines and programming tips.

Contents

1 Introduction

Scheme is a small, versatile dialect of Lisp. Like Lisp, it is an *applicative* or *functional* language. This means that, unlike imperative languages, which are made up of sequences of instructions, applicative programs are based solely on functions and their return values. The only means of control is recursion. Functional languages are characterized by their lack of *side effects*, which are operations that change the state of some variable or environment. That is, an operation never modifies its argument; instead, it returns a copy of that argument that is identical to the original argument except for the changes specified by the function. This provides a property called *referential transparency*, which means that a function called with the same arguments will always exhibit the same behavior. Functions in imperative languages may depend on side effects or global variables, and thus referential transparency need not hold.

Section 2 describes the XScheme interpreter, its invocation, and a few control primitives for using the interpreter. Section 3 describes some fundamentals of the Scheme language, such as atoms, lists, and evaluation. Section 4 describes the boolean predicates available in Scheme, and Section 5 goes on to describe the control structures provided. Section 6 describes the mathematical operations provided in Scheme. Section 7 describes function definition, and Section 8 describes variable bindings and the use of local and global variables. Section 9 describes functions as first-class, i.e., using functions as arguments to, or return values from, functions, and also discusses unnamed (lambda) functions. Section 10 provides two programs as explanatory examples. Section 11 discusses some hints and techniques for simplifying the development and debugging process. Finally, Section 12 provides a few style guidelines and tips for programming in Scheme.

2 The Interpreter

Scheme is an interpreted, or interactive, language, meaning that expressions are entered into the interpreter one at a time, at which time the interpreter immediately evaluates them and outputs the result. The XScheme interpreter is invoked from the command prompt by typing `xscheme`, like so (this assumes that you have already unpacked and installed the XScheme interpreter; if not, Appendix A provides instructions on how to do so):

```
% xscheme
XScheme - Version 0.22

>
```

The '>' symbol is the *prompt*, which is the interpreter's way of saying, 'give me an expression to evaluate.'

The following system-level constructs are provided for manipulation of the interpreter and the programming environment (Section 11 describes the development process and debugging tips in detail):

`(exit)`
Exit the compiler and return to the operating system shell.

`(load "`*filename*`")`
Load the text in the file specified by *filename*, just as though it was entered from the keyboard. The customary way to write Scheme programs is to enter the function definitions in a text file using a text editor such as `vi` or `emacs`, and then to load that text file in this fashion. Scheme files generally are given the extension '`.scm`'.

`(reset)`
The XScheme interpreter, upon encountering an error condition, enters an error-level state. The `reset` command returns the interpreter to the *top level*, where one may continue programming as usual.

`(transcript-on "`*filename*`")`
Turns on the transcripting feature, in which all text entered from the keyboard, and all text output from the interpreter, is echoed to *filename*. This is useful for recording sample runs and the like. Typing `(transcript-off)` turns off the transcripting and closes *filename*.

3 Basics

Everything in Scheme, both programs and data, is an *S-expression*. An S-expression may have one of two types, an *atom* or a *list*. Atoms, as the name suggests, are the fundamental objects in Scheme. An atom may be a number, a symbol, or `nil` (which is synonymous with the empty list, which we'll talk about in a moment). A fundamental distinction to the Scheme interpreter is whether or not an object should be evaluated. Numbers always evaluate to themselves, so this distinction is not important for numeric atoms, such as `27` or `3.14159`.

However, for symbols and lists, this distinction is very important. A symbol, for instance, may be either a literal (if it should remain unevaluated), or a variable name (if evaluated).

The `quote` function controls whether or not an S-expression should be evaluated; for instance, the atom '`foo`' is a variable named **foo**, while `(quote foo)` evaluates to a literal symbol **foo**. In addition, `(quote` *expression*`)` may be abbreviated `'`*expression*.

Throughout this manual, we use a double right arrow ('⇒') to denote 'evaluates to.' For instance, the following relations denote the behavior of the Scheme interpreter for the atoms we have just discussed:

`27`	⇒	`27`
`3.14159`	⇒	`3.14159`
`'foo`	⇒	`foo`
`(quote foo)`	⇒	`foo`
`foo`	⇒	*error: undefined variable*

3.1 Lists

A list is a sequence of S-expressions, separated by whitespace (spaces, tabs, newlines, and formfeeds), and surrounded by parentheses. Note that each S-expression can itself be a list, so the following are all valid lists: `(1 2 3)`, `(1 (2 3))`, `(1 (2 (3)))`, etc. In addition, a special list is defined that has no elements. This is called the *empty list*, or the *null list*, and is written `()` or `nil`.

Lists may be left unevaluated in the same manner as atoms, by quoting them.

`'(1 2 3)`	⇒	`(1 2 3)`

If a list is to be evaluated, its first element is considered to be a function name, and the remaining elements are arguments to that function. The number of arguments a function expects is called its *arity*. For instance, the addition function is called '+', and it has an arity of two. For example,

`(+ 2 3)`	⇒	`5`

As with all lists, the arguments to a function may themselves be function calls. For example,

```
(+ (+ 2 3) (+ 5 6)) ⇒   16
```

3.2 List Manipulation

Scheme, like all Lisp dialects, considers lists to be accessible only from the front (internally, they are represented as linked lists). Thus, Scheme provides a number of functions for accessing the first element of a list (the *head*), the rest of the list not including the head (the *tail*), inserting a new head, and so forth. We describe these here.

The `car` function, given a list, returns the head of that list. Note that the head of a list need not be an atom.

```
(car '(1 2 3))      ⇒   1
(car '((1 2) 3)     ⇒   (1 2)
```

Similarly, the `cdr` function returns the tail of a list. This return value is always a list.

```
(cdr '(1 2 3))      ⇒   (2 3)
(cdr '((1 2) 3))    ⇒   (3)
(cdr '(1 (2 3)))    ⇒   ((2 3))
(cdr '(1))          ⇒   ()
```

Applying `car` or `cdr` to an empty list is an error[1].

The `cons` function is used for list construction; given an element and a list, it creates a new list identical to its list argument, but with the element added as the new head.

```
(cons '1 '(2 3))    ⇒   (1 2 3)
(cons '(1 2) '(3 4) ⇒   ((1 2) 3 4)
```

The `list` function creates a list from a set of s-expressions. It takes a variable number of arguments, and merely formulates those arguments into a list.

```
(list '1 '2 '3)     ⇒   (1 2 3)
(list '(1 2) '3)    ⇒   ((1 2) 3)
```

Several other useful list manipulations are provided as well: `append`, `reverse`, and `length`. Note that these are not primitives; they may easily be defined using the primitives described above.

[1] This is not really true; see Section 12

The `append` function merely concatenates two lists.

```
(append '(1 2) '(3 4)) ⇒  (1 2 3 4)
```

The `reverse` function reverses the order of elements in a list.

```
(reverse '(1 2 3)) ⇒   (3 2 1)
```

The `length` function returns the length of the list in elements. Note that this does not include elements of any sublists; it merely counts the number of s-expressions that make up the list.

```
(length '(1 2 3))   ⇒   3
(length '((1 2) 3)) ⇒   2
```

4 Predicates

Predicates in Scheme are boolean operators. They return either true, denoted `#t`, or false, denoted by the empty list, `()`. The predicates defined in Scheme are:

`(atom?` *expression*`)`
Returns true if *expression* is an atom, or the empty list, false otherwise.

```
(atom? 'a)          ⇒   #t
(atom? 12)          ⇒   #t
(atom? '(1 2 3))    ⇒   ()
(atom? '())         ⇒   #t
```

`(list?` *expression*`)`
Returns true if *expression* is a list, including the empty list. Note that the empty list is the only expression that returns true for both `atom?` and `list?`.

```
(list? '(1 2 3))    ⇒   #t
(list? 'a)          ⇒   ()
(list? '())         ⇒   #t
```

`(null?` *expression*`)`
Returns true only if *expression* is the empty list, `()` or `nil`.

```
(null? '())         ⇒   #t
(null? '(1 2 3))    ⇒   ()
(null? 'a)          ⇒   ()
```

`(not` *predicate*`)`
Returns true if *predicate* returns `()`, false otherwise.

```
(not '())           ⇒   #t
(not #t)            ⇒   ()
```

`(equal?` *expression1 expression2*`)`
Returns true if *expression1* and *expression2* are structurally equivalent; that is, if they have the same list structure and all their atoms are equal. This is not to be confused with `eq?`, which returns true only if its arguments are *the same object*. That is, `equal?` tests for structural equivalence, while `eq?` tests for pointer equivalence. In the following examples, assume that we have already entered `(define foo '((1 2) 3))`.

```
(equal?  '((1 2) 3) '((1 2) 3))     ⇒   #t
(eq?  '((1 2) 3) '((1 2) 3))        ⇒   #t
(equal?  '((1 2) 3) foo)            ⇒   #t
(eq?  '((1 2) 3) foo)               ⇒   ()
(equal?  foo foo)                   ⇒   #t
(eq?  foo foo)                      ⇒   #t
```

`(and` *predicate1 predicate2*`)`
Returns true if *predicate1* and *predicate2* both return true, false otherwise.

`(or` *predicate1 predicate2*`)`
Returns true if either *predicate1* or *predicate2* return true, or false if neither of them return true.

4.1 Numerical Predicates

In all the following examples, e_n denotes an expression that returns a numerical value.

`(=` e_1 e_2`)`
Returns true if the two numbers are equal.

`(<` e_1 e_2`)`
Returns true if e_1 is numerically less than e_2.

`(>` e_1 e_2`)`
Returns true if e_2 is numerically greater than e_1.

`(<=` e_1 e_2`)`
Returns true if e_1 is numerically less than or equal to e_2.

`(>=` e_1 e_2`)`
Returns true if e_2 is numerically greater than or equal to e_1.

5 Control

The fundamental selection function in Scheme is `cond`. The format of `cond` is as follows:

```
(cond (predicate1 consequent1)
      (predicate2 consequent2)
                ...
      (predicateN consequentN))
```

The `cond` function evaluates the predicates of its arguments from left to right, and when *predicateI* evaluates to true, it evaluates *consequentI* and offers its return value as the return value of the `cond`. An error occurs if no predicate evaluates to true; often, *predicateN* is `else`, a predicate variable that always has value true, such that *consequentN* is evaluated if none of the others are true.

The `if` construct is a special case of `cond`, where there are only two possibly paths of execution. The format of the `if` is as follows:

```
(if predicate then-part else-part)
```

In the `if` construct, *predicate* is tested; if it returns true, then *then-part* is evaluated and its value returned as the value of the `if`; otherwise, *else-part* is evaluated and its value is returned as the value of the `if`.

6 Math Operations

As has been described earlier, math functions are called in the same manner as all other functions: a list whose first element is a math operator, and the rest of whose elements are arguments to that operator. The following math operators are defined:

(+ n_1 n_2)
Addition: returns $n_1 + n_2$.

```
(+ 2 3)           ⇒   5
(+ 1.5 2.35)      ⇒   3.85
```

(- n_1 n_2)
Subtraction: returns $n_1 - n_2$.

```
(- 5 2)           ⇒   3
(- 2.7 1.3)       ⇒   1.4
```

(* n_1 n_2)
Multiplication: returns $n_1 \cdot n_2$.

```
(* 3 5)           ⇒   15
(* 1.5 2.5)       ⇒   3.75
```

(/ n_1 n_2)
Division: returns $n_1 \div n_2$.

```
(/ 8 2)           ⇒   4
(/ 1.5 3)         ⇒   0.5
```

(quotient n_1 n_2)
Integer division: returns the integer portion of $n_1 \div n_2$.

```
(quotient 9 4)    ⇒   2
(quotient 9 3)    ⇒   3
```

(remainder n_1 n_2)
Remainder: returns the remainder that results from the division of n_1 by n_2, or n_1 mod n_2.

```
(remainder 9 4)   ⇒   1
(remainder 9 3)   ⇒   0
```

`(sqrt` n`)`
Square root: returns $\sqrt{n}$.

```
(sqrt 4)        ⇒   2
(sqrt 2)        ⇒   1.4142135623731
```

`(expt` n_1 n_2`)`
Exponentiation: returns ${n_1}^{n_2}$.

```
(expt 2 3)      ⇒   8
(expt 4 0.5)    ⇒   2
```

7 Defining Functions

The `define` function is used to define functions. For format of `define` is:

(define (*name args*) *body*)

Where *args* is a list of arguments *arg1 arg2* ... *argN*, and *body* is the body of the function. When the function is called, via (*name arg1 arg2* ... *argN*), the actual arguments in the call are bound to the formal arguments in the `define`, which act as local variables within *body*. For example, the ever-popular factorial function would be defined as follows (n factorial is equal to $n \cdot (n-1) \cdot (n-2) \cdot \ldots \cdot 1$, and 0 factorial is defined to equal 1):

```
(define (fact n)
   (cond
      ((= n 0) 1)
      (else (* n (fact (- n 1))))))
```

A call to this function with `n` equal to 3 proceeds as follows:

```
(fact 3)
⇒    (* 3 (fact 2))
⇒    (* 3 (* 2 (fact 1)))
⇒    (* 3 (* 2 (* 1 (fact 0))))
⇒    (* 3 (* 2 (* 1 1)))
⇒    6
```

A function with a variable number of arguments is defined as follows:

(define (*name args . rest*) *body*)

Within *body*, the arguments *arg1 arg2* ... *argN* are bound to the first *N* arguments to the function, and any remaining arguments are bound as a list to *rest*.

8 Binding

Within a function, the arguments in the `define` statement are scoped as local variables, whose values are determined by the actual arguments in the call to that function. In addition, we may use `define` to declare global variables, and we may also use a construct called `let` to declare local variables in addition to those given as arguments.

Global variables may be defined as follows:

(define *name value*)

This assigns *value* to a the variable *name*. In actuality, this operation defines a function called *name*, of arity 0, that always returns *value*. It should be noted that, as in imperative programming, global variables should be avoided except in the role of constants. That is, it is acceptable to define global variables within a Scheme program, but it is never acceptable to redefine them.

Local variables may be declared using the `let` construct. The syntax of `let` is as follows:

(let ((*var1 init1*) (*var2 init2*) ... (*varN initN*)) *body*)

Within *body*, *var1* is assigned value *init1*, *var2* is assigned *init2*, and so on. Body is evaluated within the scope of these variables, and its value is returned as the value of the `let`.

9 Functions are First-Class

In Scheme, as in most dialects of Lisp, functions are *first-class*. This means that they are values, just like any other data type—they may be passed as arguments to functions and returned as values of functions. This is one of the most powerful aspects of Scheme.

One construct that is useful when passing functions as arguments is `lambda`. This gives one the capability to define functions without names. The `lambda` statement has the same syntax as the `define` statement, but without a function name. For instance, `(lambda (x) (+ 5 x))`, when evaluated, returns a function of one argument, that adds five to that argument. That is,

```
((lambda (x) (+ 5 x)) 10)   ⇒   15
```

This allows us, for example, to define a function called `map` that applies some function to each element of a list:

```
(define (map fn l1)
   (cond
      ((null? l1) '())
      (else (cons (fn (car l1)) (map fn (cdr l1))))))
```

We may now apply `map` like so:

```
(map (lambda (x) (+ 1 x)) '(1 2 3))    ⇒   (2 3 4)
(map (lambda (x) (* 2 x)) '(1 2 3))    ⇒   (2 4 6)
(map (lambda (x) (list x)) '(1 2 3))   ⇒   ((1) (2) (3))
(map sqrt '(1 4 9))                    ⇒   (1 2 3)
```

10 Examples

10.1 Example One: Flatten

The `flatten` function, given a list, returns a list of the atoms in that list. In other words, it ‘flattens’ the recursive list structure of its argument. For instance,

```
(flatten '(1 2 3))      ⇒   (1 2 3)
(flatten '((1 2) 3)     ⇒   (1 2 3)
(flatten '((1) (2) (3))  ⇒  (1 2 3)
(flatten '())           ⇒   ()
(flatten '((())))       ⇒   ()
```

The `flatten` function may be defined as follows:

```
(define (flatten l1)
   (cond
      ((null? l1) '())
      ((null? (car l1)) (flatten (cdr l1)))
      ((atom? (car l1)) (cons (car l1) (flatten (cdr l1))))
      (else (append (flatten (car l1)) (flatten (cdr l1))))))
```

This function works in the following way:

1. We define a function `flatten` that takes one argument, `l1`.
2. We branch using a `cond` statement, dependent on qualities of `l1`.
3. If `l1` is empty, we return the empty list; `(flatten '())` ⇒ `'()`. In functions that operate on lists, checking the argument for `null?` is typically the base case of the recursion, and the first branch of the `cond`.
4. If the first element in `l1` is the empty list, i.e., `l1` looks like `(() ...)`, we throw away that element and return the `flatten` of the tail of `l1`.
5. If the first element is an atom, it cannot be the empty list, since that case was captured by the previous branch. Thus, it is a normal atom, and we return that atom `cons`’ed onto the `flatten` of the tail of `l1`.
6. Lastly, if none of the previous cases were true, the first element of `l1` is itself a non-empty list, and we return the `flatten` of that list appended to the `flatten` of the tail of `l1`.

10.2 Example Two: Quicksort

The Quicksort algorithm may be easily defined in Scheme. For those not familiar with this algorithm, Quicksort sorts a list (in this case, a list of numbers) in the following manner:

1. If the list is empty, it is obviously already sorted, so we merely return it. Otherwise…
2. Choose a ‘pivot’. The easiest way is to choose the first element of the list as the pivot.
3. Partition the list into three sublists: the elements in the list that are less than the pivot, the elements that are equal to the pivot, and the elements that are greater than the pivot.
4. Recursively call Quicksort to sort the list of lesser elements and the list of greater elements. The list of equal elements all have the same value, so it is already sorted.
5. Concatenate the sorted list of lesser elements, the list of equal elements, and the sorted list of greater elements, and return it as the sorted list.

This may be written in Scheme as follows:

```
(define (partition compare l1)
   (cond
      ((null? l1) '())
      ((compare (car l1)) (cons (car l1) (partition compare (cdr l1))))
      (else (partition compare (cdr l1)))))

(define (quicksort l1)
   (cond
      ((null? l1) '())
      (else (let ((pivot (car l1)))
         (append (append (quicksort (partition (lambda (x) (< x pivot)) l1))
                        (partition (lambda (x) (= x pivot)) l1))
                  (quicksort (partition (lambda (x) (> x pivot)) l1)))))))
```

The `partition` function in this example illustrates the usefulness of passing functions as arguments; by doing so, we may use the same `partition` function for both the lesser and greater sublists. The function does this by taking a predicate function `compare`, which should return true or false, and a list, `l1`. If `l1` is the empty list, it returns the empty list. Otherwise, if the first element in `l1` returns true when `compare` is applied to it, we return that first element `cons`'ed onto the `partition` of the tail of `l1`. If `compare` returns false, we fall through to the last condition, in which we merely return the `partition` of the tail of `l1`, without the first element of `l1`.

Then, in the actual `quicksort` function, we perform the following actions: If `l1` is the empty list, we return the empty list. Otherwise, we bind a local variable `pivot` to the first element in `l1`. We then find the list of elements less than pivot by passing the function `(lambda (x) (< x pivot))` to `partition`.

This lambda function returns true if its argument, `x`, is less than `pivot`. Thus, passing this function to `partition` results in the list of elements in `l1` that are less than `pivot`. We similarly use `partition` to form the list of elements equal to `pivot` and the list of elements greater than `pivot`. We the recursively apply the `quicksort` function to the list of lesser elements and the list of greater elements, and then `append` the three results together.

11 Development and Debugging

This section describes some techniques the authors have found useful in the development and debugging process.

Unix[2] redirection can be useful in testing programs and producing sample output. As with most applications, XScheme may take its input from a file instead of the keyboard, and may send its output to a file instead of to the screen. File input is especially useful, since it allows one to assemble a file of test cases and redirect them into the interpreter, rather than retyping them every time one wants to test the program. For instance, suppose we have typed the Quicksort program from Section 10.2, and stored it in the file '`qsort.scm`'. We may test the program, and obtain output for these test cases, by creating a file like the following:

```
(load "qsort.scm")
(transcript-on "qsort.out")
(qsort '(1 9 5 3 2))
(qsort '(5 4 3 2 1))
(qsort '(1 1 2 2 3 3))
(qsort '(1 2 3 4))
(transcript-off)
(exit)
```

And then, supposing we named this file '`qsort.in`', we may redirect it into the XScheme interpreter like so:

```
% xscheme < qsort.in
```

Note that the last command in the file is `(exit)`. This exits the XScheme interpreter when the commands in the file is completed. Be careful to include `(exit)` as the last command in all such test files, as without it, the interpreter will hang, waiting for further input. After this operation completes, the transcript of the test cases and their output will be in '`qsort.out`'. Output may also be saved to a file by using redirection rather than transcripting, but this method does not echo the test input to the file, so transcripting is preferred.

Debugging is a little tricky. In a sequential language, one often debugs a program by strategically inserting output statements throughout the program to examine the values of variables, to follow flow of control, and so forth. Scheme lacks this sequentiality, so merely inserting an output statement is impossible. However, Scheme offers a construct that allows such sequential execution, called `begin`. Its format is merely:

`(begin` *expression1 expression2 . . . expressionN*`)`

In the `begin` construct, the expressions are evaluated from left to right, and the value of *expressionN* is returned as the value of the `begin` construct. One can easily see that such a construct is meaningless unless

[2] 'Unix' is a registered trademark of AT&T Information Systems

the first $N-1$ expressions have side effects, i.e., they destructively modify the state of some variable (an action that, for our purposes, is considered forbidden) or perform some I/O. This is the purpose for which we will use it: for debugging purposes, we may replace some *expression* with `(begin` *print-statement expression*`)`. This will execute `print-statement`, then evaluate *expression* and return its value as the value of the `begin`; thus, except for the output of *print-statement*, this modification is transparent in the evaluation of the program.

The output statement in XScheme is merely `(write` *expression*`)`. This prints the value of *expression* to the standard output. For example,

`write` statement	Printed to standard output
`(write 13)`	`13`
`(write 'a)`	`a`
`(write "foo")`	`"foo"`
`(write '(1 2 3))`	`(1 2 3)`

The function `(newline)` writes a newline character to the standard output.

For example, suppose we replace the entire body of the `flatten` function described in Section 10.1 with the following:

```
(begin (write l1) (newline) body)
```

Then, when we call the `flatten` function, we get output something like this:

```
> (flatten '((1 (2)) 3 (4) () 5))
((1 (2)) 3 (4) () 5)
(3 (4) () 5)
((4) () 5)
(() 5)
(5)
()
(4)
()
(1 (2))
((2))
()
(2)
()

(1 2 3 4 5)
>
```

Lastly, it should be noted that Control-C will interrupt the XScheme interpreter and drop you back to the operating system at any time.

12 Style Guidelines

This section discusses some style guidelines for Scheme, constructs to avoid, and pointers on good programming style and efficiency.

1. **Variable naming**. In Scheme, variables must begin with a letter, and all subsequent characters may be letters, numbers, or any of the following:

   ```
   + - . * / < = > ! ? : $ % _ & ~ ^
   ```

   The length of variable names is unrestricted. Thus, one should choose descriptive variable names, avoiding abbreviations whenever practical. Customarily, predicates end with a '`?`', with a few exceptions, such as the numerical predicates.

2. **Assignment forms**. Scheme does provide an assignment operator, `set!`. In addition, Scheme provides destructive versions of many functions; for instance, `append!`, given two lists, destructively appends the second list to the end of the first. These side-effecting functions are called *assignment forms*, and their names usually end with '`!`'. While they are often used in real applications, from the point of view of functional programming, they should be considered forbidden.

3. `cadadr` **and friends**. Scheme provides compositions of `car` and `cdr` formed by inserting appropriate `a`'s and `d`'s between the `c` and the `r`. For instance, `(cadr l1)` denotes `(car (cdr l1))`. These compositions should be avoided, especially the longer ones. It is doubtful that anyone, even accomplished Scheme programmers, has an intuitive feel for what `cadadr` means, and using such forms generally means you are doing something in an ugly way.

4. **Pretty-printing**. Most Scheme programmers lay out their functions as I have throughout this manual, indenting the appropriate function calls and adding all the closing parentheses at the end, like so:

   ```
   (define (append l1 l2)
      (cond
         ((null? l1) l2)
         (else (cons (car l1) (append (cdr l1) l2)))))
   ```

   To simplify matching parentheses, however, you may find it convenient to line up the closing parentheses with the expression they are closing, as with the brackets in C, like so:

   ```
   (define (append l1 l2)
      (cond
         ((null? l1) l2)
         (else (cons (car l1) (append (cdr l1) l2)))
      )
   )
   ```

   This is purely a matter of personal preference.

5. **Applying `car` and `cdr` to the empty list**. In many implementations of Scheme, including XScheme, applying `car` or `cdr` to the empty list is valid, and returns `nil` in both cases. However, this behavior is not specified in the Scheme standards, and in other implementations it may cause an error or a warning. Thus, for reasons of portability and general good programming style, one should avoid situations in which `(car '())` or `(cdr '())` get evaluated.

6. **Sequential constructs**. Scheme offers a construct for sequential programming called `begin`. This allows sequential execution of a number of Scheme statements, with the return value of the `begin` being the last statement executed within its body. In addition, some constructs, such as `let`, perform an implicit `begin`. This should be avoided (except for its use in debugging, as described in Section 11), and is basically useless in the absence of assignment forms anyway.

13 REFERENCES

[1] Abelson, Hal and Gerard Sussman, *Structure and Interpretation of Computer Programs*, McGraw-Hill, 1985.

[2] Betz, David, *XScheme: An Object-Oriented Scheme*, v0.17, 1989.

[3] Bloss, Adrienne, *CS5314 Class Notes*, Virginia Polytechnic Institute and State University, Fall 1990.

[4] Clinger, William and Jonathan Rees (eds.), *Revised$^{3.99}$ Report on the Algorithmic Language Scheme*, 1989.

[5] Friedman, Daniel and Matthias Felliesen, *The Little LISPer*, Science Reseach Associates, 1984.

[6] Friedman, Daniel and George Springer, *Scheme and the Art of Programming*, McGraw-Hill, 1990.

[7] Hofstadter, Douglas, "Lisp: Atoms and Lists," *Metamagical Themas: Questing for the Essence of Mind and Pattern*, Basic Books, 1985, pp. 396–409.

[8] Hofstadter, Douglas, "Lisp: Lists and Recursion," *Metamagical Themas: Questing for the Essence of Mind and Pattern*, Basic Books, 1985, pp. 410–424.

[9] Hofstadter, Douglas, "Lisp: Recursion and Generality," *Metamagical Themas: Questing for the Essence of Mind and Pattern*, Basic Books, 1985, pp. 424–454.

A Installation

Installing XScheme:

1. Get a copy of the files `xscheme.tar.Z` and `install.scheme`
2. Move those files to the directory in which the scheme files should be installed (`/usr/local` is good). A new subdirectory of the current directory, called `xscheme`, will be created to hold the XScheme files.
3. Type `install.scheme`
4. To run XScheme, type `(dir)/xscheme/scheme`, where `(dir)` is the directory in which XScheme was installed (such as `/usr/local`). You may find it helpful to add a line like the following to your `.cshrc` file: `alias xscheme /usr/local/xscheme/scheme`
5. You're done!

B Reference

`'` *expression* Section 3

Suppresses evaluation of *expression*. Equivalent to `(quote` *expression*`)`.

`()` Section 3

The empty list, synonymous with `nil`.

`(+` Nex_1 Nex_2`)` Section 6

Returns $Nex_1 + Nex_2$. Nex_1 and Nex_2 must be expressions with numerical values.

`(-` Nex_1 Nex_2`)` Section 6

Returns $Nex_1 - Nex_2$. Nex_1 and Nex_2 must be expressions with numerical values.

`(*` Nex_1 Nex_2`)` Section 6

Returns $Nex_1 \cdot Nex_2$. Nex_1 and Nex_2 must be expressions with numerical values.

`(/` Nex_1 Nex_2`)` Section 6

Returns $Nex_1 \div Nex_2$. Nex_1 and Nex_2 must be expressions with numerical values.

`(=` Nex_1 Nex_2`)` Section 4.1

Returns `#t` if Nex_1 and Nex_2 are numerically equal, `()` otherwise. Nex_1 and Nex_2 must be expressions with numerical values.

`(<` Nex_1 Nex_2`)` Section 4.1

Returns `#t` if Nex_1 is numerically less than Nex_2, `()` otherwise. Nex_1 and Nex_2 must be expressions with numerical values.

`(>` Nex_1 Nex_2`)` Section 4.1

Returns `#t` if Nex_1 is numerically greater than Nex_2, `()` otherwise. Nex_1 and Nex_2 must be expressions with numerical values.

`(<=` Nex_1 Nex_2`)` Section 4.1

Returns `#t` if Nex_1 is numerically less than or equal to Nex_2, `()` otherwise. Nex_1 and Nex_2 must be expressions with numerical values.

`(<=` Nex_1 Nex_2`)` Section 4.1

Returns `#t` if Nex_1 is numerically greater than or equal to Nex_2, `()` otherwise. Nex_1 and Nex_2 must be expressions with numerical values.

`(and` *predicate1 ... predicateN*`)` Section 5

Evaluates its arguments from left to right, and returns `nil` as soon as one of them evaluates to `nil`. If none of the arguments evaluate to `nil`, it returns true.

`(append` *list1 list2*`)` Section 3.2

Returns the concatenation of *list1* and *list2*.

`(atom?` *expression*`)` Section 4
Returns `#t` if *expression* is an atom or the empty list, `()` otherwise.

`(car` *list*`)` Section 3.2
Returns the head, or first element, of *list*.

`(cdr` *list*`)` Section 3.2
Returns the tail of *list*, i.e., a copy of *list* with the first element removed.

`(cond (`*predicate1 consequent1*`) ... (`*predicateN consequentN*`))` Section 5
Evaluates its arguments from left to right; when *predicateI* returns true, the function *consequentI* is evaluated and its return value is returned as the value of the `cond`. An error occurs if none of the predicates returns true.

`(cons` *elem list*`)` Section 3.2
Returns a new list the same as *list*, but with *elem* inserted as the head.

`(eq?` *expression1 expression2*`)` Section 4
Returns true if *expression1* and *expression2* are the same object, false otherwise.

`(equal?` *expression1 expression2*`)` Section 4
Returns true if *expression1* and *expression2* are equivalent objects; i.e., if they have the same list structure and their atoms are equivalent.

`(exit)` Section 2
Exits XScheme and returns to the operating system.

`(if` *predicate then-part else-part*`)` Section 5
A special case of `cond`; if *predicate* evaluates to true, the function *then-part* is evaluated and its return value is returned as the value of the `if`. Otherwise, the same is done with the function *else-part*.

`(lambda` *args body*`)` Section 9
Defines a nameless function, with arguments *args* and function body *body*. *args* should be a list `(`*arg1 arg2 . . . argN*`)`.

`(length` *expression*`)` Section 3.2
Returns the length in number of elements of *expression*. The argument *expression* must evaluate to a list.

`(let ((`*var1 init1*`) (`*var2 init2*`) ... (`*varN initN*`))` *body*`)` Section 8
Within the scope of *body*, binds *var1* to *init1*, *var2* to *init2*, etc. Returns the return value of *body*.

`(list` *arg1 arg2 . . . argN*`)` Section 3
Returns a list of its arguments, `(`*arg1 arg2 . . . argN*`)`.

`(list?` *expression*`)` Section 4
Returns true if *expression* is a list, including the empty list, false otherwise.

`(load ''`*filename*`'')` Section 2
Loads the text file specified by *filename*, just as though that text were entered from the keyboard.

`nil` Section 3
The empty list, synonymous with `()`.

`(not` *predicate*`)` Section 4
Returns `nil` if *predicate* evaluates to true, otherwise returns true.

`(null?` *expression*`)` Section 4
Returns true if *expression* is `nil` or `()`, false otherwise.

`(or` *predicate1 ... predicateN*`)` Section 4
Evaluates its arguments from left to right, and returns true as soon as one of them evaluates to true. If none of the arguments evaluate to true, it returns `nil`.

`(quote` *expression*`)` Section 3
Suppresses evaluation of *expression*. Equivalent to `'`*expression*.

`(reset)` Section 2
Resets any error condition and returns XScheme to the top level prompt.

`(reverse` *expression*`)` Section 3.2
Returns a list made up of the elements in *expression*, in reverse order. The argument *expression* must evaluate to a list.

`(transcript-on` `` ` ``*filename*`'')` Section 2
Begins a transcript, echoing all input to XScheme and output from XScheme to the file specified by *filename*.

`(transcript-off)` Section 2
Ends the transcript, closing the file specified by the earlier `transcript-on` command.

Index