Scheme Tutorial

Introduction

Scheme is an imperative language with a functional core. The functional core is based on the lambda calculus. In this chapter only the functional core and some simple I/O is presented.

In functional programming, parameters play the same role that assignments do in imperative programming. Scheme is an applicative programming language. By applicative, we mean that a Scheme function is applied to its arguments and returns the answer. Scheme is a descendent of LISP. It shares most of its syntax with LISP but it provides lexical rather than dynamic scope rules. LISP and Scheme have found their main application in the field of artificial intelligence.

The purely functional part of Scheme has the semantics we expect of mathematical expressions. One word of caution: Scheme evaluates the arguments of functions prior to entering the body of the function (eager evaluation). This causes no difficulty when the arguments are numeric values. However, non-numeric arguments must be preceded with a single quote to prevent evaluation of the arguments. The examples in the following sections should clarify this issue.

Scheme is a weakly typed language with dynamic type checking and lexical scope rules.

The Structure of Scheme Programs

A Scheme program consists of a set of function definitions. There is no structure imposed on the program and there is no main function. Function definition may be nested. A Scheme program is executed by submitting an expression for evaluation. Functions and expressions are written in the form

```
(function_name arguments)
```

This syntax differs from the usual mathematical syntax in that the function name is moved inside the parentheses and the arguments are separated by spaces rather than commas. For example, the mathematical expression 3 + 4 * 5 is written in Scheme as

```
(+ 3 (* 4 5))
```

Syntax

The programming language Scheme is syntactically close to the lambda calculus.

Scheme Syntax

E in Expressions
I in Identifiers (variables)
K in Constants

E ::= K | I | (E_0 E^*) | (lambda (I^*) E_2) | (define I E')

The star `*' following a syntactic category indicates zero or more repetitions of elements of that category thus Scheme permits lambda abstractions of more than one parameter. Scheme departs from standard mathematical notation for functions in that functions are written in the form (*Function-name Arguments*...) and the *arguments are separated by spaces and not commas*.

For example,

```
(+ 3 5)
(fac 6)
(append '(a b c) '(1 2 3 4))
```

The first expression is the sum of 3 and 5, the second presupposes the existence of a function `fac` to which the argument of 6 is presented and the third presupposes the existence of the function append to which two lists are presented. Note that the quote is required to prevent the (eager) evaluation of the lists. Note uniform use of the standard prefix notation for functions.

Types

Among the constants (atoms) provided in Scheme are numbers, the boolean constants #T and #F, the empty list (), and strings. Here are some examples of atoms and a string:

```
A, abcd, THISISANATOM, AB12, 123, 9Ai3n, "A string"
```

Atoms are used for variable names and names of functions. A list is an ordered set of elements consisting of atoms or other lists. Lists are enclosed by parenthesis in Scheme as in LISP. Here are some examples of lists.

```
(A B C)

(138 abcde 54 18)
```

```
(SOMETIMES (PARENTHESIS (GET MORE)) COMPLICATED)

()
```

Lists are can be represented in functional notation. There is the empty list represented by () and the list *cons*truction function `cons` which constructs lists from elements and lists as follows: a list of one element is (cons X ()) and a list of two elements is (cons X (cons Y ())).

Simple Types

The simple types provided in Scheme are summarized in this table.

TYPE & VALUES
boolean & #T, #F
number & integers and floating point
symbol & character sequences
pair & lists and dotted pairs
procedure & functions and procedures

Composite Types

The composite types provided in Scheme are summarized in this table.

TYPE & REPRESENTATION & VALUES
list & (*space separated sequence of items*) & any
in function& defined in a later section &
in

Type Predicates

A predicate is a boolean function which is used to determine membership. Since Scheme is weakly typed, Scheme provides a wide variety of type checking predicates. Here are some of them.

PREDICATE & CHECKS IF
(boolean? arg) & arg is a boolean
in (number? arg) & arg is a number
in (pair? arg) & arg is a pair
in (symbol? arg) & arg is a symbol
in (procedure? arg) & arg is a function
in (null? arg) & arg is empty list
in (zero? arg) & arg is zero
in (odd? arg) & arg is odd
in (even? arg) & arg is even
in

Numbers, Arithmetic Operations, and Functions

Scheme provides the data type, `number`, which includes the integer, rational, real, and complex numbers.

Some Examples:

```
(+ 4 5 2 7 5 2) -  is equivalent to \( 4 + 5 + 2 + 7 + 5 + 2 \)
(/ 36 6 2) -  is equivalent to \(\frac{\frac{36}{6}}{2} \)
```

```
(+ (* 2 2 2 2 2) (* 5 5)) - is equivalent to \( 2^{5} + 5^{2} \)
```

Arithmetic Operators

SYMBOL & OPERATION
+ & addition
in - & subtraction
in * & multiplication
in / & real division
in quotient & integer division
in modulo & modulus
in

Lists

Lists are the basic structured data type in Scheme. Note that in the following examples the parameters are quoted. The quote prevents Scheme from evaluating the arguments. Here are examples of some of the built in list handling functions in Scheme.

cons
takes two arguments and returns a pair (list).

```
(cons '1 '2)          is   (1 . 2)
(cons '1 '(2 3 4))      is   (1 2 3 4)
(cons '(1 2 3) '(4 5 6))  is   ((1 2 3) 4 5 6)
```

The first example is a dotted pair and the others are lists. \marginpar{expand} Either lists or dotted pairs can be used to implement records.

car
returns the first member of a list or dotted pair.

```
(car '(123 245 564 898))            is   123
(car '(first second third))          is   first
(car '(this (is no) more difficult))   is  this
```

cdr
returns the list without its first item, or the second member of a dotted pair.

```
(cdr '(7 6 5))               is  (6 5)
(cdr '(it rains every day))  is  (rains every day)
(cdr (cdr '(a b c d e f)))   is   (c d e f)
(car (cdr '(a b c d e f)))   is   b
```

null?
returns \#t if the {\bf obj}ect is the null list, (). It returns the null list, (), if the object is anything else.

list
returns a list constructed from its arguments.

```
(list 'a)                      is  (a)
(list 'a 'b 'c 'd 'e 'f)      is  (a b c d e f)
(list '(a b c))             is  ((a b c))
(list '(a b c) '(d e f) '(g h i)) is  ((a b c)(d e f)(g h i))
```

length
returns the length of a list.

```
(length '(1 3 5 9 11))                is  5
```

reverse
returns the list reversed.

```
(reverse '(1 3 5 9 11)) is  (11 9 5 3 1)
```

append
returns the concatenation of two lists.

```
(append '(1 3 5)  '(9 11))  is  (1 3 5 9 11)
```

Boolean Expressions

The standard boolean objects for true and false are written \verb+#t+ and \verb+#f+. However, Scheme treats any value other than \verb+#f+ and the empty list \verb+()+ as true and both \verb+#f+ and \verb+()+ as false. Scheme provides {\bf not, and, or} and several tests for equality among objects.

Logical Operators

SYMBOL & OPERATION not & negation
and & logical conjunction
or & logical disjunction

Relational Operators

SYMBOL & OPERATION
= & equal (numbers)
(<) & less than
(<=) & less or equal
(>) & greater than
(>=) & greater or equal
eq? & args are identical
eqv? & args are operationally equivalent
equal? & args have same structure and contents

Conditional Expressions

Conditional expressions are of the form:

```
(if test-exp then-exp)

(if test-exp then-exp else-exp).
```

The `test-exp` is a boolean expression while the `then-exp` and `else-exp` are expressions. If the value of the `test-exp` is true then the `then-exp` is returned else the `else-exp` is returned. Some examples include:

```
(if (> n 0) (= n 10))
(if (null? list) list (cdr list))
```

The **list** is the then-exp while **(cdr list)** is the else-exp. Scheme has an alternative conditional expression which is much like a case statement in that several test-result pairs may be listed. It takes one of two forms:

```
(cond
   (test-exp1 exp ...)
    (test-exp2 exp ...)
    ...)
```

```
(cond
   (test-exp exp ...)
    ...
   (else exp ...))
```

The following conditional expressions are equivalent.

```
(cond
       ((= n 10) (= m 1))
       ((> n 10) (= m 2) (= n (* n m)))
       ((< n 10) (= n 0)))

(cond
       ((= n 10) (=m 1))
       ((> n 10) (= m 2) (= n (* n m)))
       (else (= n 0)))
```

Functions

Definition expressions bind names and values and are of the form:

```
(define id exp)
```

Here is an example of a definition.

```
(define pi 3.14)
```

This defines `pi` to have the value 3.14. This is not an assignment statement since it cannot be used to rebind a name to a new value.

Lambda Expressions

User defined functions are defined using lambda expressions. Lambda expressions are unnamed functions of the form:

```
(lambda (id...) exp )
```

The expression `(id...)` is the list of formal parameters and *exp* represents the body of the lambda expression. Here are two examples the application of lambda expressions.

```
((lambda (x) (* x x)) 3)      is 9
((lambda (x y) (+ x y)) 3 4) is  7
```

Here is a definition of a squaring function.

```
(define square (lambda (x)  (* x x)))
```

Here is an example of an application of the function.

```
1 ]=> (square 3)
;Value: 9
```

Here are function definitions for the factorial function, gcd function, Fibonacci function and Ackerman's function.

```
(define fac
          (lambda (n)
                 (if (= n 0)
                        1
```

```
                        (* n (fac (- n 1))))))
%

%

(define fib
        (lambda (n)
                (if (= n 0)
                        0
                        (if (= n 1)
                                1
                                (+ (fib (- n 1)) (fib (- n 2)))))))
%

%

(define ack
        (lambda (m n)
                (if (= m 0)
                        (+ n 1)
                        (if (= n 0)
                                (ack (- m 1) 1)
                                (ack (- m 1) (ack m (- n 1)))))))
%

%

(define gcd
        (lambda (a b)
                (if (= a b)
                        a
                        (if (> a b)
                                (gcd (- a b) b)
                                (gcd a (- b a))))))
```

Here are definitions of the list processing functions, sum, product, length and reverse.

```
(define sum
        (lambda (l)
                (if (null? l)
                        0
                        (+ (car l) (sum (cdr l))))))
%

%

(define product
        (lambda (l)
                (if (null? l)
                        1
                        (* (car l) (sum (cdr l))))))
%

%

(define length
        (lambda (l)
                (if (null? l)
                        0
                        (+ 1 (length (cdr l))))))
(define reverse
        (lambda (l)
                (if (null? l)
```

```
                                nil
                                (append (reverse (cdr l)) (list (car l))))))
```

Nested Definitions

Scheme provides for local definitions by permitting definitions to be nested. Local definitions are introduced using the functions `let`, `let*` and `letrec`. The syntax for the `define` function is expanded to permit local definitions. The syntax of the `define` function and the `let` functions is

> Scheme Syntax
>
> E in Expressions
> I in Identifier(variable)
> ...
> B in Bindings
> ...
>
> E ::= ...| (lambda (I...) E...) |
> (let B_0 E_0) | (let* B_1 E_1) | (letrec B_2 E_2) |...
> B ::= ((I E)...)

Note that there may be a sequence of bindings. For purposes of efficiency the bindings are interpreted differently in each of the ``let'' functions. The `let` values are computed and bindings are done in parallel, this means that the definitions are independent. The `let*` values and bindings are computed sequentially, this means that later definitions may be dependant on the earlier ones. The `letrec` bindings are in effect while values are being computed to permit mutually recursive definitions. As an illustration of local definitions here is a definition of insertion sort definition with the insert function defined locally. Note that the body of `isort` contains two expressions, the first is a `letrec` expression and the second is the expression whose value is to be returned.

```
(define isort (lambda (l)
           (letrec
              ((insert  (lambda (x l)
                 (if (null? l)
                    (list x)
                    (if (<= x (car l))
                       (cons x l)
                       (cons (car l) (insert x (cdr l))))))))

           (if (null? l)
                  nil
                  (insert (car l) (isort (cdr l)))))))
```

{`letrec` is used since `insert` is recursively defined. Here are some additional examples:

```
; this binds x to 5 and yields  10
(let ((x 5)) (* x 2))
; this bind x to 10, z to 5 and yields 50.
(let ((x 10) (z 5)) (* x z))
```

`Lets may be nested. For example,` the expression

```
(let ((a 3) (b 4)
     (let ((double (* 2 a))
          (triple (* 3 b)))
        (+ double triple))))
```

`is 18.`

Input and Output Expressions

Scheme does not readily support the functional style of interactive programming since input is not passed as a parameter but obtained by successive evaluations of the builtin function **read**. For example,

```
(+ 3 (read))
```

returns the sum of 3 and the next item from the input. A succeeding call to {\tt read} will return the next item from the standard input, thus, {\tt read} is not a true function. The function {\bf display} prints its argument to the standard output. For example,

```
(display (+ 3 (read)))
```

displays the result of the previous function. The following is an example of an interactive program. It displays a prompt and returns the next value from the standard input.

```
(define prompt-read (lambda (Prompt)
   (display Prompt)
   (read)))
```

Higher Order Functions

A higher order function (or functional) is a function that either expects a function as a parameter or returns a function as a result. In Scheme functions may be passed as parameters and returned as results. Scheme does not have an extensive repertoire of higher order functions but {\tt apply} and {\tt map} are two builtin higher order functions.

- The function **apply** returns the result of applying its first argument to its second argument.

```
1 ]=>  (apply + '(7 5))

;Value:   12

1 ]=>  (apply max '(3 7 2 9))

;Value:   9
```

- The function **map** returns a list which is the result of applying its first argument to each element of its second argument.

```
1 ]=>   (map odd? '(2 3 4 5 6))

;Value: (() #T () #T ())
```

- Here is an example of a ``curried'' function passed as a parameter. **dbl** is a dubbling function.

```
1 ]=> (define dbl (lambda (x) (* 2 x)))

;Value: dbl

1 ]=> (map dbl '(1 2 3 4))

;Value: (2 4 6 8)

1 ]=>
```

- The previous example could also be written as:

```
1 ]=> (map (* 2) '(1 2 3 4))
```

```
;Value: (2 4 6 8)

1 ]=>
```

An Example Program

The purpose of the following function is to help balance a checkbook. The function prompts the user for an initial balance. Then it enters the loop in which it requests a number from the user, subtracts it from the current balance, and keeps track of the new balance. Deposits are entered by inputting a negative number. Entering zero (0) causes the procedure to terminate and print the final balance.

```
(define checkbook (lambda ()

; This check book balancing program was written to illustrate
; i/o in Scheme. It uses the purely functional part of Scheme.

        ; These definitions are local to checkbook
        (letrec

            ; These strings are used as prompts

           ((IB "Enter initial balance: ")
            (AT "Enter transaction (- for withdrawal): ")
            (FB "Your final balance is: ")

            ; This function displays a prompt then returns
            ; a value read.

            (prompt-read (lambda (Prompt)

                  (display Prompt)
                  (read)))

            ; This function recursively computes the new
            ; balance given an initial balance init and
            ; a new value t.  Termination occurs when the
            ; new value is 0.

            (newbal (lambda (Init t)
                  (if (= t 0)
                      (list FB Init)
                      (transaction (+ Init t)))))

            ; This function prompts for and reads the next
            ; transaction and passes the information to newbal

            (transaction (lambda (Init)
                      (newbal Init (prompt-read AT)))))

; This is the body of checkbook;  it prompts for the
; starting balance

  (transaction (prompt-read IB)))))
```

Appendix

DERIVED EXPRESSIONS

(cond (test1 exp1) (test2 exp2) ...)
a generalization of the conditional expression.

ARITHMETIC EXPRESSIONS

(exp x)
which returns the value of \(e^{x}\)
(log x)
which returns the value of the natural logarithm of {\bf x}
(sin x)
which returns the value of the sine of {\bf x}
(cos x)
(tan x)
(asin x)
which returns the value of the arcsine of {\bf x}
(acos x)
(atan x)
(sqrt x)
which returns the principle square root of x
(max x_1 x_2...)
which returns the largest number from the list of given {\bf num}bers
(min x_1 x_2...)
(quotient x_1 x_2)
which returns the quotient of \(\frac {x_1}{x_2}\)
(remainder x_1 x_2)
which returns the integer remainder of \(\frac {x_1}{x_2}\)
(modulo x_1 x_2)
returns x_1 modulo x_2
(gcd num1 num2 ...)
which returns the greatest common divider from the list of given {\bf num}bers
(lcm num1 num2 ...)
which returns the least common multiple from the list of given {\bf num}bers
(expt base power)
which returns the value of {\bf base} raised to {\bf power}

{\bf note: For all the trigonometric functions above, the x value should be in radians}

LIST EXPRESSIONS

(list obj)
returns a list given any number of {\bf obj}ects.
(make-list n)
returns a list of length {\bf n} and every atom is an empty list ().

HIGHER ORDER FUNCTIONS

(apply procedure obj ... list)
returns the result of applying {\it procedure} to {\it object} and returns the elements of {\it list}. It passes the first obj as the first parameter to procedure, the second obj as the second and so on. List is the remag arguments into a list to procedure. This is useful when some or all of the arguments are in a list.
(map procedure list)
returns a list which is the result of applying procedure to each element of {\bf list}.

I/O

(read)
returns the next item from the standard input file.

(write obj)
prints {\bf obj} to the screen.

(display obj)
prints {\bf obj} to the screen. Display is mainly for printing messages that do not have to show the type of object that is being printed. Thus, it is better for standard output.

(newline)
sends a newline character to the screen.

(transcript-on filename)
opens the file filename and takes all input and pipes the output to this file. An error is displayed if the file cannot be opened.

(transcript-off)
`ends transcription and closes the file.`

References

Abelson, Harold.
Structure and Interpretation of Computer Programs. MIT Press, Cambridge, Mass. 1985.

Dybvig, R. Kent.
The Scheme Programming Language. Prentice Hall, Inc. Englewood Cliffs, New Jersey, 1987.

Springer, G. and Friedman, D.,
Scheme and the Art of Programming. The MIT Press, 1989.

Last update: 02/16/2018 04:32:44

Send comments to: webmaster@cs.wwc.edu