

MINI-TUTORIAL AND HOW TO RUN SCHEME

Principles of Programming Languages 198:314

2002

This document is an informal definition of the subset of Scheme which we will use in the class. If you wish to consult references, look for those that describe the Scheme dialect of Lisp or use the initial chapters of our recommended text *The Scheme Programming Language*, Second Edition, by R. Kent Dybvig or see

http://www.swiss.ai.mit.edu/~jaffer/r4rs_toc.html

(Revised(4) Report on the Algorithmic Language Scheme) for the official language definition and

http://www.swiss.ai.mit.edu/~jaffer/scm_toc.html

(SCM Scheme Implementation) for the documentation on SCM, the implementation of scheme that we are using.

Informally, Scheme programs consist of S-expressions (or forms), entities that have the structure of lists. In Scheme, everything — data structures, function definitions, and function calls — is an S-expression; therefore, everything has the syntax of a list. The syntax of Scheme is so simple that we can give an EBNF grammar for the whole language in one rule:

S-expression \rightarrow $(\{ \text{S-expression} \}) \mid \text{Symbol} \mid \text{Number}$

We will later give an extended grammar with more semantic details. Remember that we are only studying the *pure* functional aspects of Scheme.

We are executing Scheme in an interpreter, although compilers exist for this language. The execution model to keep in mind is the `(print (eval (read)))` loop in the interpreter; it reads from the input an S-expression, evaluates it and then prints the result. When you evaluate an S-expression, $(e_1 e_2 e_3 \dots e_k)$, a sequence of events occur in order:

- evaluate e_1 to get a function to apply. Usually, e_1 is just an atom, and evaluating it just means getting its value; note that in scheme, a “function name” like `car` is really just a variable whose value is the a data structure representing the code for the function.
- evaluate $e_2, e_3, \dots e_k$ to get values of the arguments of this function.
- apply the function to these values.

A major hangup for beginning Scheme programmers is remembering that this evaluation takes place. For example, typing `(1 2)` at the Scheme prompt would result in an error because the value of 1 is 1, which is not a valid data structure to use as a function.

We can inhibit this evaluation in Scheme by *quoting* the object. For example, typing `'(1 2)` would have resulted in Scheme writing back to us the list `(1 2)`.

Primitive Functions: Figure 1 shows a EBNF grammar for Scheme for which we discuss some semantic details in this section.

List manipulation: This grammar lists some primitive functions that work on lists, namely `car`, `cdr` and `cons` as well as some predicates, namely `null?`, and `zero?`.

The functions `car` and `cdr` are used to get at elements or sublists of a list. `car` is used to extract the first element of a list; that is, `(car '(1 2))` is 1 and `(car '((1) 2))` is (1). The difference between these two lists `(1 2)` and `((1) 2)` is that the first one has two elements, the atoms 1 and 2, whereas the second one has two elements, the list (1) and the atom 2.

`cdr` is used to get at the rest of the list, “leftover” after a `car` operation. Therefore, `(cdr '(1 2))` is the list (2) and `(cdr '((1) 2))` is the list (2) as well. Note that after we take `car` or `cdr` of a list that list remains unchanged; nothing is destroyed in it. This is sometimes called “copy semantics”. These functions are merely a way of accessing sublists within a list.

These two functions can be nested; for example, `(car (cdr '(1 (2 3) 4)))` is the list (2 3) since `(cdr '(1 (2 3) 4))` is `((2 3) 4)` whose `car` is (2 3). One can write this combined function in shorthand notation as `(cadr '(1 (2 3) 4))`, thereby using the `d` and `a` to indicate the order of application of the `car`'s and `cdr`'s.

`cons` is used to construct lists from an element and a list (i.e., a `car` and a `cdr`). Thus, `(cons 1 '(2 3))` is (1 2 3) and `(cons '(1) '(2 3))` is `((1) 2 3)`.

`list` is used to construct a list from an enumeration of its values. Thus, `(list 1 2 3)` is (1 2 3), and `(list 1 '(2 3))` is (1 (2 3)). It can be defined in terms of `cons`.

S-expression	→	Function-appln Special-form Quote-expr Symbol Number
Special-form	→	Cond-expr If-expr Let-expr Define-expr Lambda-expr
Function-appln	→	(Function-name { S-expression })
Function-name	→	Arithmetic-op Comparison Transformation Predicate
Arithmetic-op	→	+ - * /
Comparison	→	> < = >= <=
Transformation	→	list car cdr cons
Predicate	→	null? list? pair?
	→	integer? symbol? predicate?
If-expr	→	(if S-expression S-expression)
If-expr	→	(if S-expression S-expression S-expression)
Cond-expr	→	(cond { Cond-clause })
Cond-clause	→	(S-expression { S-expression })
Cond-clause	→	(else S-expression)
Let-expr	→	(let ({ (Symbol S-expression) }) { S-expression })
Define-expr	→	(define Symbol S-expression)
Define-expr	→	(define (Symbol { Symbol }) { S-expression })
Lambda-expr	→	(lambda ({ Symbol }) { S-expression })
Quote-expr	→	(quote S-expression)
Quote-expr	→	(quote ({ S-expression }))
Quote-expr	→	'S-expression
Quote-expr	→	' ({ S-expression })

Figure 1: Incomplete Syntax of Scheme

Conditional expressions: `if` and `cond` are the conditional operators in Scheme. The syntax of `if` and `cond` are shown below and their semantics explained.

`(if e1 e2 e3)`

The meaning of the `if` expressions is that `e1` is evaluated, and if its value is true, the expression `e2` is then evaluated and returned as the value of the `if` expression; otherwise `e3` is evaluated and returned as the value of the `if`.

`(cond (e1 c1) (e2 c2) ... (ek ck) .`

The meaning of this construct is as follows: The `ei` are S-expressions which should evaluate to true or false. The `ci` are arbitrary Scheme S-expressions. First `e1` is evaluated. If it is true, then `c1` is evaluated and returned as the value of the `cond`. Otherwise, `e2` is evaluated and checked for truth value etc. We continue to evaluate `e3`, `e4`, `e5`,... until we find the first true value. Then the corresponding expression `ci` is evaluated and its value returned as the value of the `cond`. For good programming style, `ek` should be `else`, which always evaluates to true so that the last case “catches” all other cases.

Predicates: (by convention, identifier ends with `?` in Scheme):

- `list?` is a unary predicate that returns true if its argument is a list and `#f` otherwise.
- `null?` is a unary predicate which returns true (i.e., `#t`) if its argument is the empty list `'()`, and `#f` otherwise.
- `pair?` is a unary predicate which returns true (i.e., `#t`) if its argument is a pair (a cons cell), and `#f` otherwise.
- `symbol?` is a unary predicate which returns true if its argument is a symbol, and `#f` otherwise. (Symbols begin with a quote `'`, followed by anything that cannot start a number, and followed by almost anything else.)

- `integer?` is a unary predicate which returns true if its argument is an integer and `#f` otherwise.
`zero?` is a unary predicate which returns true if its argument has value 0 and `#f` otherwise.
- `procedure?` is a unary predicate which returns true if its argument is a function (that has been defined and can be executed), and `#f` otherwise.

Functions: The special forms `define` and `lambda` allow you to define Scheme functions, named and unnamed, respectively. For example, we can define the function that increments integer values by 1:

```
(define (add1 x) (+ 1 x))
```

Then we can call this function; for example, `(add1 2)` would return 3.

There are sometimes situations when we need a function locally but we don't need to name it.

```
>(define (one-ups l) (map (lambda (x) (+ x 1)) l))
> (one-ups '(2 4 5))
(3 5 6)
```

Here the lambda expression defines a function which adds one to its argument. Since we only need this function here, as an argument to map, there is no reason to give it a name.

The comment character is a semicolon (;). If it appears anywhere on a line, the characters to the right of it are ignored as they are a comment.

Examples to Code: Try coding the following simple Scheme functions:

- `last`, a function that returns the last element in a list. Do not use the built-in function `reverse` which reverses the elements of a list or the reverse function from your textbook.
- `stem`, a function that returns the list obtained by removing the last element from the argument list. Again, do not use the built-in function `reverse` which reverses the elements of a list or the reverse function from your textbook.
- `second`, a function that returns the second element of its argument list.
- `and2`, a function that will take the logical “and” of its two arguments. Do not use the built-in function `and`.
- `or2`, a function that will take the logical “or” of its two arguments. Do not use the built-in function `or`.

1 How to Run and Debug Scheme

You can run Scheme directly from the Unix shell by typing the command `scm` at the shell prompt. To make the Scheme interpreter read a file `funcs.scm` containing a Scheme program you have written, type the command `(load "funcs")` at the Scheme prompt. To make the Scheme interpreter evaluate an expression, for example in order to call a function you have written, type the expression to the interpreter followed by `<return>`.

We recommend that you use a shell buffer in `emacs` to run Scheme, as then it is easy to change buffers, edit your Scheme function, change buffers again and reload your functions to rerun them. Also, you can cut-and-paste expressions to be evaluated.

A useful built-in function is pretty-print command for files, which can print out their contents. To use this you type `(require 'pprint-file)` at the system prompt when you start up scheme. Then, you can invoke this command on a file by typing `(pprint-file "app1.scm")`.

Sample Execution: This section presents a sample execution in our Scheme interpreter of the following Scheme function:

```
; len computes the length of its argument list (only top level
; elements are counted, so len is a "shallow" function i.e. it
; doesn't splice into sublists. For example (len '(1 (2 (3)) 5)) = 3.
```

```
(define (len x)
  (cond ((null? x) 0)
        (else (+ 1 (len (cdr x))))))
```

which is available in the file:

www/Public/cs314/f2001/projs/scheme/students/len-app-atom-new.scm

on the undergrad SUNs. Here is a copy of the screen image. Notice that the Scheme prompt is >; all else is typed by the system, or added as comments (with ;** afterwards. Note that the directory paths in this example maybe different than those you will see this year.

```
remus%scm
SCM version 5b1, Copyright (C) 1990, 1991, 1992, 1993, 1994, 1995, 1996, 1997 Free Software Foundation.
SCM comes with ABSOLUTELY NO WARRANTY; for details type '(terms)'.
This is free software, and you are welcome to redistribute it
under certain conditions; type '(terms)' for details.
;loading /usr/local/lib/scm/require
; loading /usr/local/lib/scm/slib/require
; done loading /usr/local/lib/scm/slib/require.scm
;done loading /usr/local/lib/scm/require.scm
;loading /usr/local/lib/scm/Transcen.scm
;done loading /usr/local/lib/scm/Transcen.scm
;Evaluation took 40 mSec (0 in gc) 11475 cells work, 16344 bytes other

> (load "len-app-atom-new")
; loading len-app-atom-new
; done loading len-app-atom-new.scm
;Evaluation took 0 mSec (0 in gc) 170 cells work, 254 bytes other
#<unspecified>

> (len '(a b c))      ;** a legal function application
;Evaluation took 0 mSec (0 in gc) 24 cells work, 31 bytes other
3
> (len 'z)           ;** an illegal function application and error
                      ;** message from the system
ERROR: cdr: Wrong type in arg1 z
;Evaluation took 0 mSec (0 in gc) 10 cells work, 31 bytes other
> (len '(1 2 3 (4))) ;** another legal function application
;Evaluation took 0 mSec (0 in gc) 31 cells work, 39 bytes other
4
> (exit)
;EXIT **to leave scm gracefully
remus%
```

Debugging: What to do if scm hangs?: You may be missing a closing parenthesis somewhere, and scm is waiting for it to be closed. *ALWAYS make sure that a function's last paren matches its beginning.* (Use tab to get automatic indentation, when editing a .scm file.)

If you cannot find the problem and are desperate, inside emacs you can always kill-buffer the *shell* where scm was launched from.

To debug your Scheme programs, you have two options: either you can trace sequences of function calls using `trace` or you can set arbitrary breakpoints in your code and examine expressions defined during execution using `break`.

To use `trace`, you need to first execute:
`(require 'trace)`
in the interpreter. This allows you to trace calls/returns to specific functions, such as `foo`, by typing:
`(trace foo)`
When you want to stop tracing a function, invoke
`(untrace foo)`
An example of using `trace` is provided below:

```
> (define (len ls) (if (null? ls) 0 (+ 1 (len (cdr ls)))))
;Evaluation took 0 mSec (0 in gc) 36 cells work, 42 bytes other
#<unspecified>
> (len '(a b c))
;Evaluation took 0 mSec (0 in gc) 27 cells work, 33 bytes other
3
> (require 'trace)
;loading /usr/local/lib/scm/slib/trace
; loading /usr/local/lib/scm/slib/qp
; done loading /usr/local/lib/scm/slib/qp.scm
; loading /usr/local/lib/scm/slib/alist
; done loading /usr/local/lib/scm/slib/alist.scm
;done loading /usr/local/lib/scm/slib/trace.scm
;Evaluation took 50 mSec (0 in gc) 2725 cells work, 1933 bytes other
#<unspecified>

> (trace len)
;Evaluation took 0 mSec (0 in gc) 106 cells work, 31 bytes other
#<unspecified>
> (len '(a b c))
"CALLED" len (a b c)
"CALLED" len (b c)
"CALLED" len (c)
"CALLED" len ()
"RETURNED" len 0
"RETURNED" len 1
"RETURNED" len 2
"RETURNED" len 3
;Evaluation took 0 mSec (0 in gc) 1313 cells work, 83 bytes other
3
> (untrace len)
;Evaluation took 0 mSec (0 in gc) 85 cells work, 31 bytes other
#<unspecified>
> (len '(a b c))
;Evaluation took 10 mSec (0 in gc) 22 cells work, 31 bytes other
3

;;*****end of execution*****
```