HOW TO RUN SCHEME
Principles of Programming Languages 198:314
Spring 1999

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* by R. Kent Dybvig. A few copies of this text are available on reserve in the SEC library.

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:

\[
\text{S-expression} \rightarrow \begin{cases} \text{S-expression} & 2 \\ \text{Symbol} & 1 \\ \text{Number} & 0 \end{cases} 
\]

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 \((\text{print (eval (read))})\) 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, \((e_1 \ e_2 \ e_3 \ \ldots \ e_k)\), a sequence of events occur in order:

- evaluate \(e_1\) to get the name of a function with a function definition; often this evaluation is trivial because the name of the function is given explicitly.
- evaluate \(e_2, e_3, \ldots, e_k\) 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 system would not have a function named 1. 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. This grammar lists some primitive functions that work on lists, namely \texttt{car}, \texttt{cdr} and \texttt{cons} as well as some predicates, namely \texttt{null?}, and \texttt{zero?}.

The functions \texttt{car} and \texttt{cdr} are used to get at elements or sublists of a list. \texttt{car} is used to extract the first element of a list; that is, \((\text{car '(1 2)})\) is 1 and \((\text{car '((1) 2)})\) 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.

\texttt{cdr} is used to get at the rest of the list, “leftover” after a car operation. Therefore, \((\text{cdr '(1 2)})\) is the list \((2)\) and \((\text{cdr '((1) 2)})\) 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 delineating sublists within a list.

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

\texttt{cons} is used to construct lists from an element and a list (i.e., a car and a cdr). Thus, \((\text{cons 1 '((2) 3)})\) is \((1 \ 2 \ 3)\) and \((\text{cons '((1) 2) 3))\) is \((2 \ 3)\). We can rewrite this combined function in shorthand notation as \((\text{cons '((1 (2) 3) 4)))\), thereby using the d and a to indicate the order of application of the car's and cdr's.

\texttt{if} and \texttt{cond} are the conditional operators in Scheme. The syntax of \texttt{if} and \texttt{cond} are shown below and their semantics explained.

\((\text{if e1 e2})\)
S-expression → Function-appln | Special-form | Quote-expr | Symbol | Number
Special-form → Cond-expr | If-expr | Let-expr | Define-expr | Lambda-expr | Seq-expr
Function-appln → Function-name { S-expression } 2
Function-name → Arithmetic-op | Comparison | Transformation | Predicate
Arithmetic-op → + | - | * | /
Comparison → > | < | = | >= | <=
Transformation → list | car | cdr | cons
Predicate → null? | list? | integer?
If-expr → (if S-expression S-expression 2)
If-expr → (if S-expression S-expression S-expression 2)
Cond-expr → (cond { Cond-clause } 2)
Cond-clause → (S-expression { S-expression } 2)
Cond-clause → (else S-expression 2)
Let-expr → (let ( { Symbol S-expression } 2 { S-expression } 2)
Seq-expr → (begin { S-expression } 2)
Define-expr → (define Symbol S-expression 2)
Define-expr → (define (Symbol { Symbol } 2 { S-expression } 2)
Lambda-expr → (lambda ( { Symbol } 2 { S-expression } 2)
Quote-expr → (quote S-expression 2)
Quote-expr → (quote ( { S-expression } 2 2
Quote-expr → 'S-expression
Quote-expr → ( { S-expression } 2

Figure 1: Incomplete Syntax of Scheme

(if e1 e4 e5)
The meaning of the first of these 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. The second if expression has a similar meaning: e1 is evaluated and if its value is true, the expression e4 is evaluated and returned as the value of the if; otherwise, the expression e5 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 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. Often ek is else, which always evaluates to true so that the last case catches all other cases. If no ei evaluates to true, then the cond expression returns #f.

null? is a unary predicate which returns true (i.e., #t) if its argument is NIL (i.e., empty list or the atom NIL) and #f otherwise.

zero? is a unary predicate which returns true if its argument has value 0 and #f otherwise.
integer? is a unary predicate which returns true if its argument is an integer and #f otherwise.

list? is a unary predicate that returns true if its argument is a list and #f otherwise.

More Syntax: 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 (third l) (car (lambda (z) (cddr z) l)))

Here the lambda expression defines a function which finds the sublist of the original list, headed by the third element of the original list.

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.

- stem, a function that returns the list obtained by removing the last element from the argument list. Do not use the built-in function reverse which reverses the elements of a list.

- rev, a function that returns its argument list with elements in reverse order. Do not use the built-in function reverse which reverses the elements of a list.

- 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.

How to Run 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 funs.scm containing a Scheme program you have written, type the command (load "funs") 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>.

Of course, you can also use a shell buffer in emacs to run Scheme. Some people prefer to run Scheme in an emacs buffer, as then it is easy to change buffers, edit your Scheme function, change buffers again and reload your functions to rerun them.
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:

```
~u1i/csl14/scheme/len.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.

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/scm/require.scm
;loading /usr/local/lib/scm/transcns.scm
;done loading /usr/local/lib/scm/transcns.scm
;Evaluation took 60 mSec (0 in gc) 11475 cells work, 16332 bytes other
> (load "len") ;** loading in a file
;loading len
;done loading len.scm
;Evaluation took 0 mSec (0 in gc) 86 cells work, 123 bytes other
#<unspecified>
> (len '(a b c)) ;** a legal function application
;Evaluation took 0 mSec (0 in gc) 27 cells work, 33 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
> ^C ;**if you forget and type Control-C at Scheme
```

ERROR: user interrupt
;Evaluation took 0 mSec (0 in gc) 1 cells work, 0 bytes other
> ;EXIT ;** what happens when you type Control-d to leave Scheme

Debugging: 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 a specific function named foo by typing: (trace foo)
An example of using trace is provided below:
remus\% scm
SICM 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/Trancen.scm
;done loading /usr/local/lib/scm/Trancen.scm
;Evaluation took 110 mSec (0 in gc) 11475 cells work, 16332 bytes other
> (load "len")
;loading len
;done loading len.scm
;Evaluation took 0 mSec (0 in gc) 85 cells work, 123 bytes other
#<unspecified>
> (len '(1 2 3))
;Evaluation took 0 mSec (0 in gc) 27 cells work, 37 bytes other
3
> (require 'trace)
;loading /usr/local/lib/scm/slib/trace
; loading /usr/local/lib/scm/slib/trace/qp
; done loading /usr/local/lib/scm/slib/trace/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 30 mSec (0 in gc) 2725 cells work, 1933 bytes other
#<unspecified>
> (trace len)
;Evaluation took 0 mSec (0 in gc) 114 cells work, 31 bytes other
#<unspecified>
> (len '(1 2 3))
"CALLED" len (1 2 3)
"CALLED" len (2 3)
"CALLED" len (3)
"CALLED" len ()
"RETURNED" len 0
"RETURNED" len 1
"RETURNED" len 2
"RETURNED" len 3
;Evaluation took 10 mSec (0 in gc) 1316 cells work, 89 bytes other
3
> ;EXIT ;** control-d

If you want to try out breakpoints in your functions, before you load the function definition, you must execute:

(load "debug") ;** "uli/cs314/scheme/debug.scm
(init-debug)

in the interpreter. Then, you can execute a function in which you have embedded break statements. You use the continue expression to continue execution after a break. This can be especially useful to trace flow through a recursive function.
The following is an example of using these commands; first, we show the Scheme function we are using:

; app returns a list which is formed by concatenating its 2 list args.
; e.g. (app '(1 2) '(3)) = '(1 2 3), (app '(1 (2)) '((4))) = '((1 (2) (4)))

(define (app1 x y)
  (begin (break x y) (cond ((null? x) y) ; if one of
    ; the lists is empty, the result
    ((null? y) x); will be the second list.
    (else (cons (car x) (app1 (cdr x) y))))))

; recursive case (both lists are nonempty)

Now for the output of the actual execution:

remus% scm
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 80 msec (0 in gc) 11475 cells work, 16332 bytes other
> (load "debug") ;** "uli/cs314/scheme/debug.scm
;loading debug
;done loading debug.scm
;Evaluation took 10 msec (0 in gc) 1691 cells work, 1409 bytes other
#<unspecified>
> (init-debug)
;Evaluation took 0 msec (0 in gc) 18 cells work, 703 bytes other
#<unspecified>
> (load "app1") ;** "uli/cs314/scheme/app1.scm
;loading app1
;done loading app1.scm
;Evaluation took 0 msec (0 in gc) 98 cells work, 124 bytes other
#<unspecified>
> (app1 '(1) '(2 3))
"BREAK:" (1) (2 3)
;Evaluation took 0 msec (0 in gc) 273 cells work, 851 bytes other
1
> (continue)
"BREAK:" () (2 3)
;Evaluation took 0 msec (0 in gc) 210 cells work, 979 bytes other
1
> (continue)
;Evaluation took 0 msec (0 in gc) 7 cells work, 31 bytes other
(1 2 3)
> ;EXIT ;** control-d