Notes in Learning Scheme for 16.410 and 16.413

1 Resources

1. SCM Scheme manual
   `http://www.swiss.ai.mit.edu/~jaffer/r5rs_toc.html`
   `http://www.swiss.ai.mit.edu/~jaffer/scm_toc`
   `http://www.swiss.ai.mit.edu/~jaffer/SCM.html`
2. Edwin References
   `http://sicp.ai.mit.edu/Spring-2003/edwin-info.html`
3. The Structure and Interpretation of Computer Programs by H. Abelson and G. Sussman with J. Sussman.
   `http://mitpress.mit.edu/sicp/full-text/book/book.html`
4. Secondary resources:
   - Teach Yourself Scheme in Fixnum Days by Dorai Sitaram.
     `http://download.plt-scheme.org/doc/205/pdf/t-y-scheme.pdf`
   - SLIB
     `http://swissnet.ai.mit.edu/~jaffer/SLIB.html`

2 Running Scheme

by Andreas Hoffman

Introduction

This jumpstart shows you the basics of getting Scheme started, running simple programs, simple editing and debugging, and exiting out of Scheme. The jumpstart is oriented towards Windows installations, but can be adapted for Unix and other installations. Note that the Scheme installed in the 16.410 computer lab is the Windows version.

Please note that this jumpstart will give you only the most rudimentary skills in Scheme. To get through the course, you will have to obtain a bit more information. The Scheme user and reference manuals are excellent resources, and are available as part of the Scheme installation; just go to the Windows Start menu, and select: Programs, MIT Scheme, Documentation. Please see the course web page for the url to the Scheme download site and the location and hours of the course lab.

Hello World

Start the Scheme interpreter by going to the Windows start menu and selecting: Programs, MIT Scheme, Scheme. At the prompt, enter the following form:

```
1 ]=> (begin
(display "Hello, World!")
(newline))
```

The interpreter prints out
```
Hello, World
1 ]=>
```

The interpreter also prints out a message indicating ”unspecified return value”. This is not an error message; it simply states that the Scheme form you entered doesn't return any value. Some forms return values, some don't. Now, enter the following form:

```
1 ]=> (begin
(display "Hello, World!")
(newline) 5)
```

Now, the interpreter indicates the returned value (5). Let's exit Scheme (don't worry, we'll come back to it). To exit, simply enter the form:

```
1 ]=> (exit)
```

Using Edwin

Edwin is the editor that comes with Scheme. Edwin is based on Emacs, and therefore, many of the keystroke combinations in Edwin are the same as they are in Emacs. You don't have to use Edwin to edit Scheme programs (you could do it in Wordpad, for example). However, you are highly encouraged to learn the basics of using Edwin; it will save you time later.

To start Edwin, go to the Windows Start menu, and select: Programs, MIT Scheme, Edwin. This will bring up the Edwin editor. Type in one of the previous sequence that we used to generate the hello world message. Then, with the cursor at the end of the form, use the keystroke combination
`ctrl-x ctrl-e`
to run the evaluator. Note that in the Scheme editor, forms are not evaluated automatically.

Now, let's write a Scheme program and save it in a file. First, to get a new buffer, do
`ctrl-x ctrl-f`
Edwin will prompt you for a path. Enter an appropriate path for your directory, and end it with the file name `HelloWorld.scm`. Scheme will indicate that this is a new file. Now, enter one of the previous hello world expressions into the new buffer. When done, do
`ctrl-x ctrl-s`
This saves the file. Next, do
`ctrl-x b`

This switches you back to the Edwin Scheme interpreter. Load the file `HelloWorld.scm` into the interpreter by typing at the prompt
`(load [path])`
where [path] is the complete path that describes where the HelloWorld.scm file is. Then do `ctrl-x ctrl-e` to evaluate. Note that the result is the same as when this was typed in directly to the interpreter. Leave Edwin by doing `ctrl-x ctrl-c`.

3 Programming in Scheme

These notes are not intended to teach a novice programmer Scheme. If you have no experience in programming, or want to you should read the elegant “Structure and Interpretation of Computer Programs”. But, if you want to get started on 16.410/413 assignments in this crazy language, then the following might help.

In Matlab, we give instructions to an interpreter, and they are executed by the interpreter. A variable gets some value, a string is printed to the screen, a window is opened, etc. Each line of code has some intended effect. In C, Java or Ada, there is an intermediate step of compilation, so the effect of each line of code is delayed, but we again write the code intending for each line to do something.

Scheme is different. I find it most useful to think of programming in Scheme as, not so much giving instructions, but asking questions of the Scheme interpreter. In principle, the intention of each line of code in Scheme is not to do something, but to get an answer back. Think of the following line of Scheme code, entered at the prompt (the prompt is the `1 ]=>` thingy):

```
1 ]=> (* 3 2)
;Value: 6

1 ]=>
```

Such a line of code makes no sense in C:

```
#include <math.h>

int main(int argc, char *argv[])
{
  3*2;
}
```

In C (or Ada, or Java), this only makes sense with the effect of printing it out, or storing it in a variable, or writing it to file...:

```
#include <math.h>

int main(int argc, char *argv[]) {
  printf("%d\n", 3*2);
}
```

Unlike other languages you may be used to, where each line of code is an instruction, everything in Scheme is a form or expression that has a value. Scheme expressions come in two forms: primitives that just have some value

```
6
```

and recursive expressions that are formed by an operator and a list of arguments:

```
(operator arg1 arg2 arg3)
```

Lists are the basic (maybe the only) data structure in Scheme. We can get more complicated expressions by nesting lists within lists, but more on that later.

Some expressions are special in that they have side-effects, like printing, or opening a file, or attaching a variable name to the value of a particular expression, but it's important to think less about *doing* instructions than it is about *evaluating* expressions and getting a value back. This difference is reflected in how Scheme operates: the guts of the Scheme interpreter is typically implemented using a "Read-Evaluate-Print" loop. The Scheme interpreter process the prompt line (or an input file) one expression at a time, reading the expression, evaluating it, and returning the value of that expression. That value might be a number we care about, or that value might be the result of defining a subroutine

(or function, or procedure, depending on what you are used to calling them). If the value of the last expression is in fact the result of defining a new subroutine, then you or I might not care about the specific details of that value: we are not going to be able to interpret what Scheme gives us after evaluating the definition of the `sqrt` function the way we can interpret the value of `(* 3 2)`, but the definition of the `sqrt` function *has* a value, and that value is useful to us in performing tasks, even if we cannot actually examine the value and learn anything from it.

Of course, if the objective of every Scheme expression is to get a value back, it's hard to get anything done as complicated ideas need complicated expressions. We therefore need operators that are "special", in that they have side effects.

Naming Values

The first thing we might want to do is to store values in variables. For example,

```
1 ]=> (define name "Nick Roy")
;Value: name
```

Define is the first operator that has a side-effect. The value of the define statement is the name of the newly-defined thing, but (without explicitly saying so), Scheme has attached the name to the value. If I ask the interpreter what the value of `name` is, it tells me:

```
1 ]=> name
;Value 6: "Nick Roy"
```

I can define a list of items:

```
 1 ]=> (define l '(1 2 3 4))
```

I can also define a list of items this way:

```
 1 ]=> (define l (list 1 2 3 4))
```

The difference between these two expressions is that `'(1 2 3 4)` does not take the value of each symbol, whereas `(list 1 2 3 4)` does.

I can do math this way:

```
 1 ]=> (define six (* 3 2))
;Value: six
1 ]=> six
;Value: 6
```

Notice that `six` evaluates to a useful value (as does `6`), but `(six)` does not, since lists are evaluated by taking the value of the first symbol and using it as an operator. `six` evaluates to `6`, which is not an operator. Operators are things like `*`, `-`, `concat`, etc. And of course, `define` is an operator itself, and a `define` expression itself is a well-formed list.

Naming Subroutines

Define is the general-purpose operator for attaching names to things. We use it to attach names to chunks of Scheme code that can be evaluated.

```
1 ]=> (define (six-func) (* 3 2))
;Value: six-func
```

Here, I've defined `six-func` to be a function that, when evaluated, returns the expression `(* 3 2)`. Notice the difference between

```
1 ]=> (define six-func (* 3 2))
;Value: six-func
1 ]=> six-func
;Value 6
```

and

```
1 ]=> (define (six-func) (* 3 2))
;Value: six-func
1 ]=> six-func
;Value 7: #[compound-procedure 7 six-func]
```

If I ask for `six-func` to be evaluated (rather than asking for its value as I just did), I get

```
1 ]=> (six-func)
;Value: 6
```

I have defined `six-func` to evaluate to 2×3, which is a function that doesn't depend on anything else. I can define a new function that multiplies 2 by some number and called it `two-times`. When I evaluate `two-times` on some number, you might think of a similar process in C or Java as calling a function (or method) called `two-times` with some number as a parameter. So,

```
1 ]=> (define (two-times n) (* 2 n))
```

is a lot like

```
double two-times(double n) {
  return 2*n;
}
```

But, in Ada or Java, a function like two-times is a fundamentally different beast than the number 2, whereas in Scheme, the difference is just the specific value that each represents. To the read-print-eval loop, it's perfectly valid to call `two-times` on itself:

```
1 ]=> (two-times two-times)
```

although when this is interpreted, the evaluator complains that the special symbol `*` can't be used with compound-procedures (which is what Scheme calls functions, or procedures, or chunks of Scheme code that we can refer to later). There are other functions that *are* useful with compound-procedures, and that it makes perfect sense to use a function name like `two-times` as a parameter as well as an operator.

The most powerful property of Scheme is how each operand in an evaluation is passed to the evaluation. When an expression is evaluated, the value of each item in the list is given to the operator. For instance, in the expression

```
1 ]=> (* 2 3)
;Value: 6
```

2 and 3 are the operands in the list `(* 2 3)` that are passed to the operator $*$, which in this case multiplies them. Each of those operands needs to have a value. If the operand is a primitive, such as a number, a string or a compound-procedure definition, then that's the value. If, however, the operand is an expression itself, then that expression is evaluated.

```
1 ]=> (* 2 (* 3 4))
;Value: 24
```

The evaluation of operands is done *as needed*. If I have a list of items that I want to preserve as a list (I don't want the list to be evaluated), then I can protect it using a quotation mark. Examine the difference between

```
1 ]=> (define list-of-symbols '(* 3 2))
;Value: list-of-symbols
1 ]=> (define six (* 3 2))
;Value: six
1 ]=> list-of-symbols
;Value 12: (* 3 2)
1 ]=> six
;Value: 6
```

Understanding when to use an actual list, and when a list needs to be evaluated, can be tricky.

Conditionals

Besides being able to abstract Scheme code using compound-procedures, we want to be able to control program flow. For instance, we would like Scheme functions that respond to external input or generate different output depending on different properties of the input. For instance, we might define a `max` operator that returns the maximum of two items. We can use the `cond` special form:

```
(cond (pred clause) (pred clause) ...)
```

So we can define our `max` as

```
(define (max a b)
  (cond ((> a b) a)
        (else b)))
```

`else` is a special form within cond that always evaluates as true. There are a number of ways to combine tests, using `and`, `or`, etc. Although `cond` has no side-effects, it is a special form because evaluation of the expression stops on the first operand with a true predicate. (If it were not a special form, each predicate would be evaluated, which would be ok although possibly inefficient, but each clause would also be evaluated, which would be no good in situations where clauses had side-effects and we only wanted *one* of the side-effects to happen.

List Processing

As you must have realized, Scheme is very list-oriented. We have already seen one way to create a list, or maybe you prefer to think in terms of vectors:

```
(define v1 '(1 2 3 4))
```

This is Scheme short hand for an explicit operation to create lists:

```
(define v2 (list 1 2 3 4))
```

We can extract items from the front of the list, using a operator called (for historical reasons) `car`, and we can get everything *but* the front of the list using `cdr`.

```
]=> (car v1)
;Value: 1
]=> (cdr v1)
;Value 5: (2 3 4)
```

Scheme has a somewhat bizarre shorthand for composing `car`'s and `cdr`'s

```
1 ]=> (caddr v1)
;Value: 3
```

We can use the list operators to do things like take the dot product of two vectors.

```
1 ]=> (define (dot a b)
         (if (or (null? a) (null? b)) 0
             (+ (* (car a) (car b)) (dot (cdr a) (cdr b)))))
;Value: dot
1 ]=> (dot v1 v2)
;Value: 70
```

We can assemble lists using the `append` operator, which takes the elements of two or more lists and creates a new list of those elements:

```
1 ]=> (append v1 v2)
;Value 7: (1 2 3 4 5 6 7 8)
```

You have to be careful to pass lists to append. This works:

```
1 ]=> (append '(* 3 2) '(* 4 5))
;Value 11: (* 3 2 * 4 5)
```

but might not be what you want. This might be what you want:

```
1 ]=> (append (* 3 2) (* 4 5))
;The object 6, passed as an argument to append, is not a list.
```

but doesn't work. This is probably what you want:

```
1 ]=> (append (list (* 3 2)) (list (* 4 5)))
;Value 12: (6 20)
```

The `cons` operator takes a value and a list, and returns a new list `cons`tructed from the value and the list:

```
1 ]=> (cons (* 3 2) (list (* 4 5)))
;Value 12: (6 20)
```

Notice that the first element after the operator evaluates to a value, not a list.

Iteration and Recursion

Iteration is something that can take a long time to get accustomed to in Scheme. There is no iteration operator (although you can define one). Instead, doing something over and over again is often performed using recursion: a function (or compound-procedure) calling itself with new parameters. A common example of recursion is the factorial function:

```
(define (factorial n)
  (cond ((= n 1) 1)
        (else (* n (factorial (- n 1))))))
```

Notice how factorial calls itself with a parameter whose value is one less than whatever it was called with?

4 Functions as parameters

In addition to recursion, many things that you might be used to doing in C or Java within by iteration can be done with list operators. In Scheme, there are a number of helper functions that can operate across the list. For example,

```
1 ]=> (define a '("boston" "washington" "chicago" "pittsburgh"))
;Value: a
1 ]=> (map string-capitalize a)
;Value 2: ("Boston" "Washington" "Chicago" "Pittsburgh")
```

Notice how the second argument is the name of what you might ordinarily thing of as a function? This is one of the most powerful ideas in Scheme: that functions are not syntactically different than anything else. The only way functions (or compound procedures) differ is that they can be used to evaluate the rest of the list.

We can make store functions in lists along with data:

```
1 ]=> (define (complex-add z1 z2)
         (list (+ (caar z1) (caar z2)) (+ (cadar z1) (cadar z2))))
;Value: complex-add
1 ]=> (define (add a1 a2) ((cadr a1) a1 a2))
;Value: add
1 ]=> (define c1 (list '(1 2) complex-add))
;Value: c1
1 ]=> (define c2 (list '(3 4) complex-add))
;Value: c2
1 ]=> (add c1 c2)
;Value 4: (4 6)
```

And thus object-oriented programming is born. We get polymorphism in that last function call; the add function looks at the 2nd argument for the definition of how to add these data types.

```
1 ]=> (define (real-add z1 z2)
         (list (+ (caar z1) (caar z2))))
;Value: complex-add
1 ]=> (define (add a1 a2) ((cadr a1) a1 a2))
;Value: add
1 ]=> (define r1 (list '(1) real-add))
```

```
;Value: c1
1 ]=> (define r2 (list '(3) real-add))
;Value: c2
1 ]=> (add c1 c2)
;Value 4: (4)
```

5 Anonymous Functions

The preceding section described expressions that take functions as parameters, and we supplied the functions by naming them, i.e., real-add, complex-add, capitalize, etc.. There are situations, however, we might only ever need a function once, just to use it as an argument in some other expression. When you call quick-sort, for example, you have to supply a comparator function that takes 2 parameters and returns true if the first term should come before the second in the full ordering. If you are sorting a list of just numbers, then you can supply the `>` function, but if you are sorting a list of some abstract data, then you need to supply a function that can order the data appropriately, and often, this is the *only* time you would need such a function.

For example, suppose you were sorting a list of student records, and you wanted to sort on the last name, which was the 3rd item in each student record list. You would have to write function that knew to call `string-compare` on the `caddr` each item. But, since it can be a pain to define a new function for every single-purpose use, scheme supports the use of anonymous functions, using the special form `lambda`. Instead of using `real-add` to refer to our real addition function, we can refer to it in the following way:

```
1 ]=> (lambda (z1 z2) (list (+ (caar z1) (caar z2))))
;Value 7: #[compound-procedure 7]
```

Of course, once we evaluated the lambda, the value is gone since we didn't define a symbol to hold the procedure. But we can use an anonymous function wherever we might use a named function:

```
1 ]=> (define anon-c1 (list '(1 2) (lambda (z1 z2)
(list (+ (caar z1) (caar z2)) (+ (cadar z1) (cadar z2))))))
;Value: anon-c1
1 ]=> (define anon-c2 (list '(3 4) (lambda (z1 z2)
(list (+ (caar z1) (caar z2)) (+ (cadar z1) (cadar z2))))))
;Value: anon-c2
1 ]=> (add anon-c1 anon-c2)
;Value 4: (4 6)
```

This is unwieldy for multiple instances of the data, so we usually use named functions whenever we might use it more than once. But, we can define a complex number quick-sort that sorts numbers on their magnitude in the Argand plane:

```
(define (complex-sort l) (quick-sort l
  (lambda (a b)
    (< (+ (* (caar a) (caar a)) (* (cadar a) (cadar a)))
       (+ (* (caar b) (caar b)) (* (cadar b) (cadar b)))))))
```

6 Doing something useful

This example is taken from the Structure and Interpretation of Computer Programs. We want to solve for square roots using Newton's Method. Given some number, we repeatedly guess at its square root, improving the guess over and

over until some numerical accuracy is achieved. Let's say our number is x, and our current guess is y. At each step t, (iteration?), we improve our guess using

$$y_{t+1} = \begin{cases} \frac{1}{2}\left(y_t + \frac{y_t}{x}\right) & : |y_t^2 - x| > \epsilon \\ y_t & : \text{otherwise} \end{cases} \tag{1}$$

Notice two features. We have a conditional, and we have an iterative procedure. Let's first define the improvement step of Newton's method:

```
(define (improve guess x)
  (average guess (/ x guess)))

(define (average x y)
  (/ (+ x y) 2))
```

Given a guess of 5 for a square root of 36, we get

```
1 ]=> (improve 5 36)
;Value: 61/10
```

which is much closer to $\sqrt{36} = 6$ than 5 was.

Let's now define a test to see if we are within range of the desired numerical precision, of finding a square root of x that, when squared, is within .001 of x:

```
(define (good-enough?  guess x)
  (< (abs (- (square guess) x)) 0.001))

1 ]=> (good-enough? (improve 5 36) 36)
;Value: ()
```

and we see the after improving 5 to $\frac{61}{10}$, it's still not good enough yet as it returned the empty list `()` rather than value `#t`.

We define the iteration procedure using recursion: if the `good-enough?` test fails, we call `sqrt-iter` again using an improved guess.

```
(define (sqrt-iter guess x)
  (cond ((good-enough? guess x) guess)
        (else (sqrt-iter (improve guess x) x)))
)
```

We define a wrapper function `my-sqrt` that provides an initial guess.

```
(define (my-sqrt x)
  (sqrt-iter 1.0 x))
```

And we can now find square roots.

```
1 ]=> (my-sqrt 36)
;Value: 6.000000005333189
```

7 Doing something even more useful – Linear Regression

Let's finally do something somewhat involved. Let's generate some data from a noisy linear model, and then use linear regression to recover the line parameters.

The equation of a line is of course

$$y = m \cdot x + b \tag{2}$$

Let's assume our measurements of y are corrupted with some Gaussian noise w :

$$p(w) = \frac{1}{\sqrt{2 * \pi * \sigma^2}} e^{\frac{-(x-\mu)^2}{2\sigma^2}} \tag{3}$$

Without going into the details of the mathematics (which can be found in Luc Devroye. Non-uniform random variate generation, Springer-Velag, New York, 1986.), we can sample a point from the domain $(-\infty, \infty)$ according to the distribution $\mathcal{N}(\mu, \sigma^2)$ using the following:

$$u \sim \text{uniform}(0, 1.0] \tag{4}$$
$$v \sim \text{uniform}[0, 1.0) \tag{5}$$
$$z = \sqrt{-2 \log u} \cdot \sqrt{\cos 2\pi v} \tag{6}$$
$$x = \mu + \sigma \cdot z \tag{7}$$

```
(define pi (* 4 (atan 1)))

(define (sample_gaussian mean variance)
  (lambda ()
    (define norm (/ 1.0 2.0))
    (define u (- 1.0 (* (random 1.0) norm)))
    (define v (* (random 1.0) norm))
    (define z (* (sqrt (* -2.0 (log u))) (cos (* 2.0 pi v))))
    (+ mean (* (sqrt variance) z)))
)
```

Let's define a function `g` that makes calls to our `sample_gaussian` function with mean 0 and variance 0.01. Let's also define a function for the line that returns $y = mx + b + g$ where g is a call to our sampling function.

```
(define (line m b noise) (lambda (t) (+ (* m t) b (noise))))

(define g (sample_gaussian 0 0.01))

(define l (line 2 .25 g))
```

Now let's create an array of noisy line data evaluated from -1 to 1 in increments of 0.01.

```
(define (create_vector start stop increment)
  (if (> start stop) '()
      (cons (list start (l start))
      (create_vector (+ start increment) stop increment))))

(define v (create_vector -1 1 0.01))
```

At this point, v is a list of 2-item lists, where each pair of items is an x and a y co-ordinate.

Given our noisy line data, we can use linear regression to recover the parameters m and b:

$$b = \frac{\sum_{i=1}^{n} x_i \cdot y_i - \frac{1}{n}\sum_{i=1}^{n} x_i - \sum_{i=1}^{n} y_i}{\sum_{i=1}^{n} x_i \cdot x_i - \frac{1}{n}(\sum_{i=1}^{n} x_i)^2} \quad (8)$$

$$a = \sum_{i=1}^{n} y_i - b \cdot \sum_{i=1}^{n} x_i \quad (9)$$

Let's implement each of those terms separately as a function that takes in our array of data, and returns the appropriate sum.

```
(define (sum-x m)
  (if (eqv? m '()) 0
      (+ (caar m) (sum-x (cdr m)))))

(define (sum-y m)
  (if (eqv? m '()) 0
      (+ (cadar m) (sum-y (cdr m)))))

(define (sum-sqd-x m)
  (if (eqv? m '()) 0
      (+ (* (caar m) (caar m)) (sum-sqd-x (cdr m)))))

(define (sum-xy m)
  (if (eqv? m '()) 0
      (+ (* (caar m) (cadar m)) (sum-xy (cdr m)))))

(define (b m) (/ (- (sum-xy m) (/ (* (sum-x m) (sum-y m)) (length m)))
                 (- (sum-sqd-x m) (/ (* (sum-x m) (sum-x m)) (length m)))))

(define (mean-x m)
  (/ (sum-x m) (length m)))

(define (mean-y m)
  (/ (sum-y m) (length m)))

(define (a m) (- (mean-y m) (* (b m) (mean-x m))))
```

If we call a and b on the array v we should recover the original slope and intercept of our line.

We can plot these under Unix using the following commands.

```
(define xwin (make-graphics-device 'x))

(define (plot v)
  (graphics-operation xwin 'fill-circle (car v) (cadr v) 0.01)
  () )

(define (do-plot m)
  (if (eqv? m '()) '()
```

```
      (begin (plot (car m)) (do-plot (cdr m)))))

(define (do-line m)
  (graphics-draw-line xwin -1 (+ (a m) (* (b m) -1)) 1 (+ (a m) (* (b m) 1))))

(do-plot v)

(do-line v)
```

I/O

The following subsection on Input/Output will not be required for any assignment in this class. It is given for general interest, and some of this may help you in debugging your code. You will *never* need any of these operators, especially the ones from SLIB, for assignments in this class.

Something that is often skipped over in Scheme tutorials and textbooks is input and output. The simplest output operator is `display`, as in

```
1 ]=> (display "\nHello, World!\n")
 Hello, World!
;Unspecified return value
```

If you want to get input from the user, `read-line` is probably what you want. For instance, to turn scheme into string capitalizer (an odd thing to do, I suppose),

```
1 ]=> (define (capitalizer)
  (begin (define string (read-line))
         (cond ((not (string-null? string))
                (begin (display (string-capitalize string))
                       (newline)
                       (capitalizer)))
               (else '()))))
1 ]=> (capitalizer)
hello world
Hello world
hello WORLD
Hello world
[hit return on a blank line to end]
;Value: ()
```

Something we haven't talked about is loading into Scheme extra packages. There is a Scheme package called "SLIB", which you can download and install on your local machine, and will give you lots of extra functionality such as text parsing and printf's of the sort you might be used to, as in:

```
1 ]=> (load "/home/local/lib/slib/mitscheme.init")
;Loading "/home/local/lib/slib/mitscheme.init"
;Loading "/usr/local/lib/slib/require.scm" -- done
 -- done
;Unspecified return value
1 ]=> (require 'printf)
;Loading "/usr/local/lib/slib/printf.scm"
```

```
;Loading "/usr/local/lib/slib/strcase.scm" -- done
 -- done
;Value 3: "/usr/local/lib/slib/printf"
1 ]=> (printf "Hello there %s. %d! is %d\n" "Nick" 5 (factorial 5))
Hello there Nick. 5! is 120
;Value: 28
```