Burt Rosenberg, July 17, 1996

0.0.1 How to Scheme, and Scheme Well.

Scheme is a programming language derived from Lisp and substantially influenced by Algol 60. Lisp was first described by its inventor, John McCarthy in the article "Recursive Functions of Symbolic Expressions and Their Computation by Machine, Part I", appearing in *Communications of the ACM*, **3**(4), 1960.

We will be using the Scheme48 implementation of Scheme. Scheme is generally an interpreted language. You enter the Scheme48 program and then are in an endless read-evaluate-print loop. To exit Scheme48 type `,exit`, the comma is important: it directs the message to the intpreter of Scheme, rather than to Scheme itself. Other comma commands can be explored by asking for help: `,help` or `,?`.
EXAMPLE:

```
petersburg 2% scheme48
Welcome to Scheme 48 0.36 (made by burt on Fri Jul 12 16:18:19 EDT 1996).
Copyright (c) 1993, 1994 by Richard Kelsey and Jonathan Rees.
Please report bugs to scheme-48-bugs@martigny.ai.mit.edu.
Type ,? (comma question-mark) for help.
> ,exit
petersburg 3%
```

The read-evaluate-print loop, or *repl* for short, reads an input line, evaluates it according to the rules of Scheme, prints the result, and then loops back to accept more input. For instance:

```
> 48
48
>
```

This is an example of Scheme evaluating a number. A number evaluates to its own value. More precisely, "48" is the *external representation* of the Scheme object of type *number* of value 48. Scheme recognizes the external representation, evaluates the number and prints the result. Numbers are self-evaluating. They evaluate to their own value. The other self-evaulating Scheme objects are *boolean, character* and *string.* Here are examples of the true and false boolean objects and a string object:

```
> #t
#t
> #f
#f
> "Hello"
"Hello"
>
```

To apply a function to its arguments, write the arguments after the function, enclosed in parentheses:

```
> (+ 3 4)
7
>
```

This is called *application.* The three elements are each evaluated in an unspecified order. The leftmost must evaluate to type *procedure.* The procedure is then applied to the resultant objects on its right.

```
> (+ 1 2 3)
6
> (+ 1)
1
> (+)
0
> +
'#{Procedure 139 +}
>
```

Notice that the plus function takes any number of arguments, with the sum of zero numbers being 0, and evaluates by itself to type procedure.

The evaluation of all arguments before the use of the arguments in the function is called *applicative-order evaluation,* also *eager evaluation.* An alternative is to evaluate arguments "by-need", also called *normal-order* or *lazy evaluation.* Lazy evaluation may be more efficient, but it also can give different answers. The if-function in Scheme takes three arguments, a predicate returning boolean and two further arguments. The first is evaluated if the predicate evaluates true, else the second argument is evaluated. The second argument may be empty.

```
> (if (> 1 0) "True" "False")
"True"
> (if (< 1 0) "True" "False")
"False"
>
```

This format is called a *special form* because it does not follow the rules of applicative-order. Else, this construct could not be used to avoid the computation of an illegal value! There are other special forms, in fact, the language Scheme is exactly the collection of its special forms. Another important special form is the define:

```
> a

Error: undefined variable
       a
       (package user)
1> (define a 3)
1> a
3
1>
```

Scheme, and Lisp, were designed to mimic the Lambda-calculus. The Lambda calculus does not have assigned variables — it was invented to avoid the semantic difficulties associated with assignment to variables by eliminating variables altogether except as place-holders in function definition. Previously, there was the concept of "pure Lisp", without assignment, and Lisp being a superset. We will experiment with the avoidance of assignment, but we do understand how impractical it is to eliminate assignment altogether. The define form creates a variable and sets its value. Define does not evaluate its second argument, but takes it literally. In the above example, we see that evaluating the argument "a" results in an error of undefined variable, until it is defined. The third argument is evaluated:

```
1> (define a (+ 1 2 3))
1> a
6
1> (define a (if (> 1 0) "True" "False"))
1> a
"True"
1>
```

Note that "a" can be sometimes of one type, and sometimes of another. Unlike C or Pascal, which use *manifest types*, scheme variables have *latent types,* also called *dynamic types.* The type of a variable adapts to be that of the value it contains.

The distinction between manifest and latent types gives two different models of the storage environment. With manifest types, such as C or Pascal, it is very useful to consider a variable to be a box, with the variable name written on the side, in which values are placed. The form of the value is determined by that of the box. In Scheme, variables are the binding together of a symbol which identifies the variable, and the variable's current value. I think of a Scheme variable as the location which is the focal point of this binding. A new variable is created by taking a fresh location and associating the identifing symbol with the location and a value with the location. An old variable is updated by changing at the location of the identifying symbol of variable the associated value of the location.

Variables live altogether in an *environment*, which is consulted each time a symbol needs to be evaluated. To evaluate a symbol, the current environment is searched for a location which binds the symbol to a value. For instance, the symbol "+" is associated in the start-up environment with the plus procedure. In the phrase `(+ 3 4 )` the evaluation of "+" yields type procedure, as required of the first object of an application.

Each scheme object has one and exactly one of the following types: *boolean, symbol, char, vector, pair, number, string* or *procedure*. Each type has a scheme predicate evaluating true exactly on objects of that type: `boolean?, symbol?, char?, vector?, pair?, number?, string?` or `procedure?` By convention, scheme predicates end their names with a question mark.

EXAMPLES:

```
> (procedure? (if #t + 1))
#t
> (procedure? (if #f + 1))
#f
> (number? (if #t + 1 ))
#f
```

```
> (number? (if #f + 1 ))
#t
>
```

Objects of type "procedure" are created with the *lambda special form.* The letter lambda refers to the lambda calculus, from which the principles of Lisp were derived. Lambda calculus was invented by Alonzo Church to create a very clear language for the construction and study of functions. See "The Calculi of Lambda-Conversion", by Alonzo Church, Princeton Univ. Press, 1941. A procedure is a body of code with variables indicating the place of substitution for arguments when the procedure is applied. The original lambda calculus notation for the function $f(x, y) = x + y$ is $\lambda xy.x + y$. In Scheme, we write `( lambda (x y) (+ x y) )`. The lambda form evaluates to a procedure. It is applied to its arguments by placing it in the first position of an application:

```
> ( (lambda (x y) (+ x y)) 3 4 )
7
> ( (lambda (x f y) (f x y)) 3 + 4 )
7
>
```

In the second example, the repl first evaluates in an unspecified order the four elements of the application. The first and third evaluate to procedures, the second and fourth evaluate to numbers. The lambda is applied to the arguments. Three new locations are created, and the *formal arguments* of the lambda are bound to their corresponding values. In the lambda body, this environment is refered to in order to evaluate the application `(f x y)`. The completion of the lambda returns the value of the last form evaluated, and dissolves the formal argument bindings.

0.0.2 Church Numerals

```
appomattox 71% more church.scm
;;;
;;; Church numerals in scheme
;;; taken in part from page 84 of Struc. & Interp.
;;;
;;; Burt Rosenberg
;;; 6 November 1992
;;;

(define zero (lambda (f) (lambda (x) x)))

(define (1+ n)
   (lambda (f)
      (lambda (x)
         (f ( (n f) x))
      )
   )
```

```
)

(define (succ x) (+ x 1 ))

(define (the-value n)
   ( (n succ) 0 )
)

(define (add-cn n m )
   (lambda (f)
      (lambda (x)
         ( (n f) ((m f) x) )
      )
   )
)

(define (mult-cn n m)
   (lambda (f)
      (lambda (x)
         ( (n (m f)) x )
      )
   )
)

appomattox 72% scheme48
Welcome to Scheme 48 0.36 (made by burt on Fri Jul 12 16:18:19 EDT 1996).
Copyright (c) 1993, 1994 by Richard Kelsey and Jonathan Rees.
Please report bugs to scheme-48-bugs@martigny.ai.mit.edu.
Type ,? (comma question-mark) for help.
> ,load church.scm
church.scm ......
> (define three (1+ (1+ (1+ zero))) )
> (define four (1+ (1+ (1+ (1+ zero)))) )
> (the-value three)
3
> (the-value four)
4
> (the-value (add-cn three four))
7
> (the-value (mult-cn three four))
12
> ,exit
appomattox 73%
```

0.0.3 Errara

A Scheme object has *at most one* of the following types, but may be of a type not in this list: *boolean, symbol, char, vector, pair, number, string* or *procedure*. An important example is the empty list. The Scheme Report claims that it is of its own type:

```
> (symbol? '() )
#f
> (pair? '() )
#f
> (boolean? '() )
#f
> (null? '())
#t
>
```

A list whose `car` is `'a` and whose `cdr` is the pair `'(c . d)` has external representation `(a c . d)`. Spaces around the period are very important!

```
> ( cons 'a ( cons 'b 'c ) )
'(a b . c)
>
```

The minus and divide operators do not allow an empty argument list. The plus sign can be redefined.

```
> (-)

Error: wrong number of arguments
       (-)
1> (/)

Error: wrong number of arguments
       (/)
2> ,reset
> (define + -)
> (+ 3 4)
-1
>
```