Burt Rosenberg, July 18, 1996

0.0.1 How to Scheme, the sequel.

The fundamental data structure in Lisp and Scheme is the *list.* Lists are built from *pairs.* A pair is an object with a car field and a cdr field (pronounced could-er). A pair is constructed with the procedure `cons`, and the procedures `car` and `cdr` deconstructs the pair into the two fields:

```
> (cons 1 2)
'(1 . 2)
> (car (cons 1 2))
1
> (cdr (cons 1 2))
2
> (pair? (cons 1 2))
#t
>
```

A list of length n is a pair whose cdr is a list of length $n-1$. The zero-length list is special: it is not a pair, it is the empty list, with external representation `()`.

```
> (pair? '())
#f
> (null? '())
#t
> (list? '())
#t
>
```

Here is an example of how lists are used. We find the minimum integer on a list of integers by comparing the car of the current list with the minimum of the integers in the cdr of the list. Recall that the cdr of a list is also a list, making the recursion complete. It would be convenient to have a plus infinity, the minimum of an empty list, which is larger than any integer. We represent this with the boolean value false and a special min function:

```
> (define (lmin l) ; find the minimum integer on list l
          ( if (null? l) #f   ; will serve as +infinity
               (my-min (car l) (lmin (cdr l)))
          )
  )
> (define (my-min a b)
          (if b (min a b)  ; anything other than #f is true
                a          ; b is +infinity, minimum is a
          )
  )
```

```
> (lmin '(3 -1 2))
-1
> (lmin '())
#f
>
```

Procedures used as first-class objects give rise to various difficulties. A procedure is defined in a context. Certain of the variables which appear in the procedure are connected with memory cells according to this context. When a procedure is passed into another procedure, it is passed into another context. Using *dynamic scoping*, it is this new environment which determines the locations to which these names relate. Original Lisp used dynamic scoping, and it is widely recognized to be more of a bug than a feature. Programs written using *static scoping* are more easily analyzed.

The block structuring introduced by Algol 60, and followed by C, Pascal and Scheme is an example of static scoping. It is also called *lexical scoping*, because the binding of variable names to storage locations can be ascertained by looking only at the physical text of the program. Fixed at compile time, these associations hold no matter where the function is invoked.

In Scheme, a name is bound to the location created by the innermost matching formal in the nesting of lambda's which define the function. If no matching formal exists, there is a catch-all global environment in which the name is to be bound. The *define* construct builds bindings in the global environment. The lambda construct builds bindings in local environments. Scheme implementations may or may not allow define to be used to create local bindings. Helpful versions of lambda, such as `let, letrec` and `let*` are available for forming local bindings.

EXAMPLE:

```
> (define f (lambda(y)(+ x y)) ) ; the x is bound to the global envir.
> (define x 2)                   ; this is the global envir.'s x.
> (f 3)
5
> ( (lambda(x)(f 3)) 1 )         ; this x is not the x inside of f
5
> (define x 1)
> ( (lambda(x)(f 3)) 2 )
4
>
```

For Scheme, since procedure can build and return procedures, local environments must "remain alive" so that the new procedure can use its location when invoked:

```
> (define f (lambda(x) (lambda() x) ) )
> f
'#{Procedure 6724 f}
> (f 1)
'#{Procedure 6725 (unnamed in f)}
> (define one (f 1))
> one
```

```
'#{Procedure 6725 (unnamed in f)}
> (one)
1
> (define two (f 2))
> two
'#{Procedure 6725 (unnamed in f)}
> (two)
2
> (define xx (f (lambda() x)) )
> (xx)
'#{Procedure 6744}
> (define x 4)
> ((xx))
4
> (define x 5)
> ((xx))
5
>
```

But, in fact, Scheme does not make a special effort to keep such variables "alive." All variables in Scheme are said to have *infinite extent,* once created, they are never destroyed. As a matter of implementation, an interpreter is free recover the storage locations claimed by the variable if the interpreter has incontrovertible evidence that the variables are of no further use to the computation. Various *garbage collection* strategies exist to accomplish this task.

With lambda and if statements, we can write all functions, even without the help of define, although define helps cut down typing-fatigue. The prototypical example is the self-reproducing function which, when applied to a copy of itself, copies the copy and applies the copy to the copy of the copy, and so on:

```
> ( (lambda(x)(x x)) (lambda(x)(x x)) )

Interrupt: keyboard
1>
```

This gives us an important reminder of the difference between applications and special forms. The meaning of the first form is the constant value 1, the second form has meaning "non-termination":

```
> (if #t 1 ((lambda(x)(x x))(lambda(x)(x x))) )
1
> (define (applicative-if c s1 s2)(if c s1 s2))
> (applicative-if #t 1 ((lambda(x)(x x))(lambda(x)(x x))) )

Interrupt: keyboard
1>
```

This self-reproducing function idea can be put to good use in creating recursive functions without the need for function names. Here is a standard recursive algorithm for summing integers up to n. We define it using the name `sum`:

```
> (define (sum n) (if (= n 0) 0 (+ n (sum (- n 1)))))
> (sum 1)
1
> (sum 5)
15
> (sum 10)
55
>
```

It can be written anonymously if it is written twice, and the second writing is passed inside the first, in case of a recursive call:

```
> (  (lambda (f n) ( if (= n 0) 0 (+ n (f f (- n 1)))))
     (lambda (f n) ( if (= n 0) 0 (+ n (f f (- n 1)))))
      5 )
15
> (  (lambda (f n) ( if (= n 0) 0 (+ n (f f (- n 1)))))
     (lambda (f n) ( if (= n 0) 0 (+ n (f f (- n 1)))))
      10 )
55
>
```

Actually, we can do without writing the function twice, using a recursive shell that applies the recursive function to itself and the integer argument:

```
> (  (lambda (R m)(R R m))     ; recursive shell applied to...
     (lambda (f n)             ; a function taking a function, integer pair
          ( if (= n 0)
               0
               (+ n (f f (- n 1)))
          ))
     5                         ; and the integer
  )
15
>
```

0.0.2 Tail recursion

The Scheme language can be said to be the collection of its special forms. The special forms are indicated by a *syntactic keyword* appearing in the first position of the list to be evaluated, that is, in the position of a function. It is possible to confuse syntax and procedure for this reason. However, the reserved syntactic keywords, as we have mentioned, implies important changes in the ordering in which the remaining list elements are evaluated.

Some syntactic constructs are primitive, others are derived. The `lambda, if` and `set!` constructs are primitive. Other syntatic constructs have meanings which refer back to these primitives. Derived

expressions include `let` for allocation of local variables inside a procedure, and `do`, a flexible construct for iteration.

```
> (let ((x 2)(y 3))    ; a derived expression
       (+ x y)
  )
5
> ( (lambda(x y)(+ x y)) 2 3 )  ; an equivalent primitive expression
5
> ( do (               ; a derived expression
           (flag #f)   ; variable initial-value
           (retval 0 (+ retval 1)) ; variable init-value step
        )
        (flag retval)  ; ( test evaluate-and-exit-if-test-true )
        (set! flag #t) ; evaluate these expressions if test false
   )
1
> ( (lambda (DO-LOOP) (DO-LOOP DO-LOOP #f 0)) ; an equiv. primit. expr.
    (lambda (continuation flag retval)
             ( if flag retval   ; if flag true
                       ((lambda()
                           (set! flag #t)
                           (continuation continuation flag (+ retval 1))
                       ))
             )
     )
  )
1
>
```

Although we have demonstrated the possibility for iterative constructs done recursively, practice presents us with an important distinction. Iterative programs use minimal stack. With each recursive call, local variables and a return address is pushed onto the stack. Space requirements for a recursive procedure is proportional to recursion depth.

A style of recursion, called *tail recursion* does not need to place information on the stack. A tail recursive recursion has the recursive call as the very last action performed. Hence there is no reason to return to this level of recursion. Local variables and a return address are not pushed; when the recursive call returns, it returns immediately to the caller of the entire recursion.

Scheme requires that the interpreter make use of tail recursion. The previous rewritings of iterative constructs with recursion were tail recursive, hence Scheme will use recursion for iterative constructs and not require excessive stack.

It is often easy to change recursive programs to tail recursive programs. Here is an example of our program to find the minimum in a list, written in tail recursive style:

```
> (define (trmin n l) ; find the minimum integer on list l,
```

```
                    ; tail recursive, passing n, the minimum
                    ; so far found
      ( if (null? l) n
           (trmin (my-min (car l) n) (cdr l))
      )
  )
> (define (my-min a b)
          (if b (min a b)  ; anything other than #f is true
                a          ; b is +infinity, minimum is a
          )
  )
> (trmin #f '(3 -1 2))
1
> (trmin #f '() )
#f
>
```

0.0.3 The Halting Problem

Is it possible to write a program that can analyze other programs, in particular, to determine if the program will not go into an infinite loop when given a certain input? No. To write a program which is guaranteed not to loop, either the program must be specially conceived, or the proof of halting must be specially conceived: there is no "one size fits all" answer to this problem.

If there were, we could express the program in Scheme as a lambda construction `H` of two variables:

$$\texttt{(H x y)} = \begin{cases} \texttt{\#t} & \text{if } \texttt{(x y)} \text{ halts} \\ \texttt{\#f} & \text{if } \texttt{(x y)} \text{ does not halt.} \end{cases}$$

Consider now the definition of `F`, which twists `H` upon itself in an unavoidable self-contradiction:

```
> (define F
       (lambda(x)
            ( if ( H x x )
                 ((lambda(x)(x x))(lambda(x)(x x)))
               1
            )
       )
  )
> (define (H x y) #f )
> (F F)
1
> (define (H x y) #t)
> (F F)

Interrupt: keyboard
```

```
1>
```

The last expression either loops forever or returns 1, depending on whether `(H F F)` returns true or false, respectively. If `(F F)` loops forever, it is because `(H F F)` is true, meaning that `(F F)` halts; if `(F F)` returns 1, it is because `(H F F)` is false, meaning that `(F F)` does not halt. The existence of `H` is not a logically consistent possibility. The ability of Scheme and the lambda calculus to clearly notate arguments such as these is the reason it is preferred in the study of semantics.