Crossroads
The ACM Student Magazine

An Introduction to Scheme

by ***Shriram Krishnamurthi***

Introduction

Every programming language has a ``machine model,'' which is a philosophy of how that language views the underlying machine as being structured. Traditional languages such as C have a physical machine model, which means they think of the ambient system in terms of the hardware units that it is built up of. Others, such as Prolog, prefer to think of the underlying system as being a logic engine.

Some languages take on a different viewpoint: they perceive the model to be a mathematical one, whereby the machine is a huge mathematical ``brain'' capable of performing certain computations rapidly and unerringly, but of otherwise unknown construction. Such languages are less likely to deal with memory locations and assignments, and more likely to deal with functions and their evaluation. The core of Scheme, which we will discuss here, is one such language.

History

Scheme was designed by Gerald Jay Sussman and Guy L. Steele, Jr. at the MIT AI Lab in 1975. It is a descendant of at least three languages, which we briefly outline here.

In the late fifties, John McCarthy designed Lisp as an approximation to the lambda calculus, a theoretical computation model proposed by logicians in the thirties in which the fundamental computational object was the function, and the fundamental operation was function application. Lisp was a dynamically typed language with an unusual (prefix and fully-parenthesized) syntax which made it particularly amenable to the

rapid prototyping of other languages. Scheme was one such language, originally written wholly in MacLisp (a popular Lisp implementation). Since these prototypes were most easily written when they shared Lisp's syntax, Scheme inherited this and has retained it ever since.

Algol, another seminal language dating to the same period as Lisp, was one of the first languages to introduce static scoping. Since the implementors of Scheme had been studying Algol, and since static scoping seemed necessary for their experiment anyway, Scheme adopted this scoping protocol.

The crucial player of the trio was Carl Hewitt's ``actor'' model of computation. Processes were seen as actors which communicated by passing messages to each other. These messages were themselves actors. Sussman and Steele defined two kinds of objects: functions and actors. Functions returned values, while actors took an actor to which they passed the result of their computation. However, the process of creating these similar entities was almost identical, and indeed, the act of message-passing looked exactly like that of calling a function.

On studying their implementation, Sussman and Steele discovered that apart from the primitives used to write actors and functions, they were in all other respects entirely identical and could therefore be merged. Combining the features of the two, function calls were again made fundamental to computation (as in the lambda calculus), while the actor style of computation lead to interesting paradigms of computation being expressed by function application alone. Thus Scheme was born [6].

Over the years, Scheme has grown and evolved considerably. Yet, it retains something of the original minimal spirit; the core is kept relatively small, and new primitives are rarely introduced. There are now committees that have standardized Scheme, and several high-performance compilers are available for it.

Now we explore the rudiments of this language, touching along the way some of the original ideas mentioned above.

Evaluation and Reduction

Consider the arithmetic expression

```
2 + 3 * 5 - 7
```

To determine its value, we would (probably implicitly) write down a series of intermediate values before arriving at the final answer, as below:

```
    2 + 3 * 5 - 7
==> 2 + 15 - 7
==> 17 - 7
==> 10
```

Note several properties of the above sequence of steps. First, we determine which operations to perform before others by using well-known rules of precedence (* before + and -, here). Next, when more than one sub-expression can be simplified, we choose one arbitrarily. Finally, we stop when we reach an answer or ``value'' (which cannot be furhter reduced).

The above process of taking an expression and simplifying it until we obtain a value is called ``evaluation.'' Each step along the way is called a ``reduction.'' (The arrow may be read as ``reduces to.'') This process was central to grade-school arithmetic. It is also central to programming in Scheme.

As an example, the reduction above might be written in Scheme as below:

```
    (- (+ 2 (* 3 5)) 7)
==> (- (+ 2 15) 7)
==> (- 17 7)
==> 10
```

Scheme has a ``prefix'' (sometimes referred to as ``Polish'') notation, where the operator (such as +) goes before the operands, not between them. Formally, we write the operator and its operands inside parentheses, with one or more spaces separating them from each other. Also, parentheses are only used to denote function application, not for grouping. Note that we no longer need to know anything about the rules of precedence for determining the result of an expression.

To reiterate: computing in Scheme is very similar to doing grade-school arithmetic. We have the benefit of having an extremely fast friend, the computer, who can determine the value of most expressions much faster than we can determine it ourselves.

Functions

As with other languages, and with mathematics, we can abstract over certain frequently performed operations by defining functions. In Scheme, we would define a function as below:

```
(define <function-name>
  (function (<argument>)
    <body>))
```

For example,

```
(define square
  (function (x)
    (* x x)))
```

defines square to be a function of one argument, x, which returns the value of x multiplied by itself.

How do we compute with functions? Let us try a simple example:

```
(square 2)
```

Scheme first reduces the argument to the function to a value. In this example, since 2 is a number and hence a value, that step is complete. Then the *name* of the argument (x, above) is associated with the *value* of the argument (2), and the body is written with each instance of the name substituted by the value. Hence:

```
==> (square 2)                                  /argument evaluated/
==> ((function (x) (* x x)) 2)                 /x associated with 2/
==> (* 2 2)                                        /body rewritten/
==> 4
```

In short, we reduce the arguments to values, then perform textual substitution. A slightly more complicated example (where / denotes the division operator) is:

```
    (square (/ (+ 3 (- 2 1)) 2))
==> (square (/ (+ 3 1) 2))                  /evaluate argument first/
```

```
==> (square (/ 4 2))
==> (square 2)                                  /we know how to do this/
==> 4
```

Like we said, it's simplification all over again. Chances are this is how you were taught to compute with functions in school, in which case you already know how to compute with Scheme!

Note that while we often wish to name an abstracted computation (such as `square` above), there is no inherent reason for a function to have a name; indeed, Scheme allows functions to be ``anonymous.'' How do we write such a function, however? Actually, we have done just this above. The expression

```
(function (x) (* x x))
```

is an anonymous function of one argument that, when applied, multiplies its argument by itself. We can use it directly if we wish:

```
((function (x) (* x x)) 2)
```

applies it to the argument 2, which reduces as we've seen above. The `define` merely tells Scheme to ``remember'' this anonymous function by associating the name `square` with it. We will return to anonymous functions shortly.

Of course, it is possible for functions to take more than one argument. We will use the `if` construct to make decisions. It has three sub-parts: the test, the ``then'' part and the ``else'' part. We can write a function that returns some quadratic root as follows (note that > performs the expected arithmetic test, while `sqrt` computes square roots of non-negative arguments):

```
(define one-root
  (function (a b c)
    (if (>= (- (* b b) (* 4 a c)) 0)
        (/ (+ (- b) (sqrt (- (* b b) (* 4 a c)))) (* 2 a))
        0)))
```

This function returns one root, -b + sqrt (b^2 - 4ac) / 2a , if any exist, else it returns the number 0.

Some things are notable above. For one thing, Scheme is far more permissive about names than most other languages are. Note that there is no ambiguity as to what `one-root` represents: it is not `root` subtracted from one, since that would be written as

```
(- one root)
```

instead!

More importantly, however, we notice that one expression -- the determinant -- is being written (and, in most cases, evaluated) two times, though its value is obviously the same in both instances. How can we avoid this?

Binding

When we introduced functions, we said they abstracted patterns of computation. While computing with them, however, we noticed that they performed an auxiliary task, that of ``binding'' (associating) a name to a value. For example, in the `square` evaluations above, the argument `x` was ``bound'' to the value 2. Furthermore, the expression given as an argument to `square`, no matter how complex, was evaluated only once to produce a value; this value was then used two times in the body.

We ought to be able to exploit this idea to rewrite `one-root` and eliminate the extra, redundant computation. First, let us express the core of the function in terms of the computation we want performed only once:

```
(function (determinant)
  (if (>= determinant 0)
        (/ (+ (- b) (sqrt determinant)) (* 2 a))
        0))
```

Of course, as expressed, the body of this function has no way of knowing what `a` and `b` are, so we ought to add them as arguments:

```
(function (determinant a b)
  (if (>= determinant 0)
```

```
            (/ (+ (- b) (sqrt determinant)) (* 2 a))
            0))
```

Now we're set: so long as we can pass this function the right arguments, the determinant will be evaluated only once (as per our rule for computing with functions) and will then be used once or twice, depending on its value. We know that the determinant expression is

```
    (- (* b b) (* 4 a c))
```

so we ought to be able to wrap all this in a function that takes `a`, `b` and `c`:

```
    (function (a b c)
      ((function (determinant a b)
         (if (>= determinant 0)
               (/ (+ (- b) (sqrt determinant)) (* 2 a))
               0))
       (- (* b b) (* 4 a c)) a b))
```

Finally, ``remembering'' this function by the name `better-root` (with a `define`) gives us the same function as we had before, except the determinant is computed only once. Let us try this out on some sample values:

```
    (better-root 1 2 1)
==> ((function (a b c) ((function  ... )
           ... )) 1 2 1)
==> ((function (determinant a b)  ... )
     (- (* b b) (* 4 a c)) a b)                  /remembering a, b, c/
==> ((function  ... ) 0 1 2)
                                              /evaluating the arguments/
==> (if ( >= determinant 0)  ...  0)
                                        /remembering determinant, a, b/
==> (/ (+ (- b) (sqrt determinant)) (* 2 a))          /true condition/
==> (/ (+ (- 2) (sqrt 0)) (* 2 1))                      /substituting/
==> (/ -2 2)
==> -1
```

and indeed, substituting `x = -1` in `1 * x^2 + 2 * x + 1` gives `1 + -2 + 1 = 0`, so -1 is indeed a root. Hence the anonymous function introduced above is both a computational and a documentation aid: it helps reduce the computation done on a sub-expression, and it associates a name with an important expression, making the inner computation more readable (``if the determinant is non-zero, use it appropriately'').

It would, of course, be inconvenient if we had to introduce anonymous functions every time we wished to abstract over some sub-computation in this manner. Hence, Scheme provides a convenient mechanism for doing this:

```
(define best-root
  (function (a b c)
    (let ((determinant (- (* b b) (* 4 a c))))
      (if (>= determinant 0)  ...  0))))
```

is the same as the above function. A `let` is converted into the application of an anonymous function; Scheme takes care of doing this automatically for us.

Scope

We briefly mentioned that Scheme is a statically (or ``lexically'') scoped language, which means that when a name is referenced, it is looked up in the nearest lexical environment, not in the current dynamic one.

The reason we desire static scoping is so that we can reason about program fragments without having to resort to executing them. This ability is crucial. Consider, for example, the following pre:

```
(let ((x 1))
  (let ((f (function (y) (+ x y))))
    (+ (let ((x 2)) (f 2))
       (f 3))))
```

In a statically scoped system, what is the value of this expression? Well, initially `x` and `f` are bound to 1 and to the function, respectively. Now we perform the addition. Say the first branch is evaluated first so `x` is bound to 2. When we invoke `(f 2)`, however, recall that the function looks `x` up at its nearest upward *static* binding, which in this case is to the value 1. Hence the result of that application is `(+ 1 2) ==> 3` . Likewise,

the next application of f yields the value 4, for a final value of 7. Note that the function f could just as well be renamed to something like add1, since it *always* adds 1 to its argument.

Now view the program with dynamic scope. First x is bound to 1, then f to the function. But now we are in a quandary: which branch of the + is evaluated first? We don't know, and above, it wouldn't have made any difference. Here, however, if the first branch is evaluated first, then f becomes a function that adds 2 to its argument, so that the sum would yield 4 + 5 = 9. If the second branch were evaluated first, however, we would get 4 from the (f 3) call, then 4 from the (f 2) call, for a sum of 8.

Hence, dynamic scope is problematic for several reasons. We cannot determine what some functions do merely by looking at their text; we must also know of all the contexts in which they are used. This is a non-trivial task. Furthermore, we are also required to resolve ambiguities such as operator argument evaluation order, which may not be desirable.

Closures

Consider again the following declaration:

```
(let ((x 1))
  (let ((add1 (function (y) (+ x y))))
    \xbc))
```

We mentioned earlier that when the function add1 (previously called f) was used, it looked x up in environments that statically enclosed it. A use of add1 in the part denoted by the ellipsis would have no difficulty doing this, since the binding for x is still ``active'' at that point.

Say, however, we had an application of this form:

```
((let ((x 1))
   (let ((add1 (function (y) (+ x y))))
    add1))
 3)
```

What happens here? The internal `let` returns `add1` as its value; when `add1` function is applied, the `let` which bound `x` has been closed, so no binding for `x` currently exists. Hence, the function `add1` needs to look up a binding that no longer exists. Is this an error?

The answer is no. A Scheme ``function'' is really a pair of items: the pre representing the action of the functions, and some representation of the bindings present at the time the function was created. This composite object is known as a ``closure.'' Happily for us, closures are automatically created by Scheme (they are the ``value'' of a `(function ...)`), and we may use them in comfort. In the above example,

```
    ((let ((x 1)) (let ((add1  ... )) add1)) 3)
==> ((let ((add1  ... )) add1) 3)      /x bound to 1/
```

At this stage, when the value of `add1` is returned as the value of the inner `let`, the closure retains the binding of `x` to 1. Hence, we obtain

```
    (add1 3)
==> ((function (y) (+ x y)) 3)
==> (+ x 3)                                    /replacing 3 for y/
==> (+ 1 3)                              /using stored binding for x/
==> 4
```

Closures are extremely powerful objects. They are essential to the maintenance of static scoping in the presence of first-class functions. Furthermore, they provide us with a powerful encapsulation mechanism: in the above example, while `add1` knows of the value of `x`, when the returned function is applied the context of application has no way of determining the `x` binding, or even of knowing of its existence. Thus, programs can be written in a highly modular fashion, and yet present a single, unified interface to a user.

Aggregate Objects

Mathematically, there are usually both ordered and unordered aggregate objects. (Vectors are an example of the former, and sets of the latter.) However, since unordered objects can be easily derived from ordered objects by merely ignoring the order, it suffices to have only ordered objects. Scheme provides the mechanism called

a ``list'' for creating such aggregates.

Lists follow a simple recursive structure. A list is either empty, or it is some value prepended onto a list:

```
    l --> empty
       |  (join v l)
```

where v is any Scheme value (such as numbers, closures and lists). There are functions for creating both types of lists: `empty` (the `list` function called with no arguments) creates the empty list, and `(join v l)` creates an augmented list. Notice that as per the definition above, all lists must end in the empty list.

```
    empty
==> ()
    (join 3 empty)
==> (3)
    (join 2 (join 1 (join 0 empty)))
==> (2 1 0)
```

(The parentheses in the output delimit the list.)

Similarly, there are two functions for extracting elements from lists. `first` obtains the first element (which was the last one joined on), and `rest` gives the remainder of the list as if the first element has been removed.

```
    (define l (join 3 (join 2 empty)))    /remember this value for l/
    (first l)
==> 3
    (rest l)
==> (2)
```

However, any arbitrary Scheme value can be stored in a list. For instance, we can do the following:

```
    (define l (join first (join 3 (join 2 empty))))
```

which makes the first element in `l` to be the `first` function. Then

```
    ((first l) (rest l))
==> ((first (join first (join 3  ... ))) (rest l))
                                      /replacing l in the first part/
==> (first (rest l))                       /evaluating function part/
==> (first (rest (join first (join 3  ... ))))
                                                    /replacing l/
==> (first (join 3  ... )) /evaluating argument part/
==> 3
```

Lists provide a simple and powerful way of building up all sorts of data structures such as queues, trees and graphs, in addition to being useful in their own right. We will return to them later.

Recursion

The factorial function is n! = n * n-1 !, where 0! = 1 and n is non-negative. In Scheme, to compute 5!, we could write

```
    (define factorial
      (function (n)
        (if (= n 0)
            1
            (* n (factorial (- n 1))))))
    (factorial 5)
==> (* 5 (factorial 4))                              /since 5 =/= 0/
==> (* 5 4 (factorial 3))                                /and so on/
==> (* 5 4 3 2 1)
==> 120
```

(The function = tests whether its two (numeric) arguments are equal.)

Most programming languages support such ``recursive'' function definitions. However, they also support other control constructs such as while and do loops. The core of

Scheme eschews this verbosity, since such loops are really a special case of recursion. For instance, we could easily write a function that provided until loops:

```
(define until-loop
  (function (done? n next-n
                   action final-value)
    (if (done? n)
        final-value
        (action n
                (until-loop done?
                            (next-n n)
                            next-n
                            action
                            final-value)))))
```

where done? is a test to determine termination, n holds the value of the loop variable, next-n is a function for generating a new value of the loop variable, final-value is a value to return upon completion of the loop, and action is a function for combining the current value with the result of the rest of the loop.

As an application, we can use the above to sum numbers from 1 to 100:

```
(until-loop (function (v) (= v 100))
            1
            (function (v) (+ 1 v))
            (function (v rest-v) (+ v rest-v))
            100)
```

The loop terminates when 100 is reached (first argument), returning the value of 100 in the last step (last argument); alternatively, we could have terminated at 101 and returned 0. The initial value is 1 (second argument). The next value of the loop control is obtained by incrementing the current one (third argument). Finally, we combine the current loop variable with the result of (summing) the rest by adding the two together. Evaluating the above would yield 5050.

We see that recursion is powerful enough to express all looping constructs and suffices for our language. Note that recursion itself is really a function calling itself. However, it

is a sufficiently important and powerful paradigm that it warrants a special mention.

Maps and Filters

We shall now present some examples that utilize first-class functions to abstract over computational patterns. In the process we will derive two extremely useful functions that are used extensively in Scheme programming.

Say we have a list of numbers representing grades on a test. We shall henceforth refer to it by the name grades. Assume the test was graded out of 80, and we now wish to scale the scores to 100. We could do it as follows:

```
(define scale
  (function (list-of-grades)
    (if (empty? list-of-grades)
        empty
        (join (/ (* (first list-of-grades) 100) 80)
              (scale (rest list-of-grades))))))
(scale grades)
```

(The function empty? determines whether a list is empty.) This function considers, in turn, each element in the list, scales it appropriately, and joins it onto the list formed by scaling the remaining grades.

To make the program structure a little cleaner, we can abstract out the function that performs the scaling. This yields:

```
(define scale\n
  (function (list-of-grades)\n
    (let ((new-grade (function (old-grade)\n
                       (/ (* old-grade 80) 100))))\n
      (if (empty? list-of-grades)\n
          empty\n
          (join (new-grade (first list-of-grades))\n
                (scale (rest list-of-grades)))))))\n
(scale grades)\n
```

But perhaps tests have different weights. This function only works on tests that scale from 80 to 100. So we abstract the new-grade function out of the loop entirely,

passing in an appropriate scaling function. Thus we get:

```
(define scale
  (function (list-of-grades new-grade)
    (if (empty? list-of-grades)
        empty
        (join (new-grade (first list-of-grades))
              (scale (rest list-of-grades))))))
(scale grades (function (old-grade) (/ (* old-grade 100) 80)))
```

At this point, however, notice that the function bound to `scale` has nothing to do with grades. It is, quite simply, a function that takes a list and an action function as arguments, and applies the action to the list element-by-element. Since it ``maps'' the old list onto a new one, with the action being the mapping, this function is traditionally known as `map`. If passed the appropriate function, it can print address labels just as well as it could scale up scores.

Returning to our grades example, say we wanted to know all the grades that were above a certain threshold. Our program might look like:

```
(define partition
  (function (list-of-grades)
    (if (empty? list-of-grades)
        empty
        (let ((first-grade (first list-of-grades)))
          (if (> first-grade 50)
              (join first-grade (partition (rest list-of-grades)))
              (partition (rest list-of-grades)))))))
(partition grades)
```

If the first grade passes some threshold, it is joined onto the result of culling the remainder. If it does not, we return the result of culling the remainder.

We can repeat our experiences from the `map` example and obtain a function that is independent of grades and honor rolls. Instead, it takes a list and a question and asks the question of each element in the list. Only if the answer is affirmative do we retain the element. Quite naturally, this paradigm is called a ``filter.'' We could express it as:

```
    (define filter
      (function (list-of-grades predicate?)
        (if (empty? list-of-grades)
            empty
            (let ((first-grade (first list-of-grades)))
              (if (predicate? first-grade)
                  (join first-grade (partition (rest list-of-grades)))
                  (partition (rest list-of-grades)))))))
    (filter grades (function (grade) (> grade 50)))
```

Maps and filters are extremely useful in structuring a system. Rather than create functions that do tasks that are similar, we are better off abstracting over the task, and passing in appropriate functions to handle the specifics. Since the list is ubiquitous in Scheme, functions that handle common recursion patterns over them find frequent use.

Handling Control

Consider again the evaluation of a simple arithmetic expression such as

```
    (+ 5 (* 3 (- (+ 2 1) 7)))
                 -------
==> (+ 5 (* 3 (- 3 7)))
               -------
==> (+ 5 (* 3 -4))
         --------
==> (+ 5 -12)
    ---------
==> -7
```

At each stage, the underlined part represents the ``thing to be evaluated next,'' and its evaluation leads to one more reduction. For brevity, we have avoided underlining the evaluation of the operators and the numbers.

The underlined portion is called a ``redux'' (i.e., the thing to be reduced). Certainly we can rewrite each stage in the reduction by replacing the redux by a variable and applying it to the redux itself, like so:

```
    (+ 5 (* 3 (- (+ 2 1) 7)))
==> ((function (v) (+ 5 (* 3 (- v 7))))  (+ 2 1))
==> ((function (v) (+ 5 (* 3 v)))        (- 3 7))
==> ((function (v) (+ 5 v))              (* 3 -4))
==> ((function (v) v)                    (+ 5 -12))
==> -7
```

The part of an expression surrounding a redux is called the ``context'' of that redux.

Now suppose we are in the midst of a computation and discover we are about to perform an operation that would lead to an error (such as division by zero). We would prefer to terminate gracefully, maybe returning some value to the user. We add a new form to Scheme which we call `abort`. When `abort` is passed some value, it halts program execution and returns that value directly to the user. Thus,

```
    (abort 10)
==> 10
    (* 2 (+ 1 (/ 5 (abort 0))))
==> 0
```

Notice from the second example above that what `abort` does is to merely ignore its context. Also, notice that *every* context implicitly has an `abort` at its head; this will be relevant soon.

While we may concede that `abort` is certainly powerful, and its effect would have been very cumbersome to achieve without it being built-in, it appears that `abort` is also too powerful for some uses. Consider a function that finds the product of the elements of a list of numbers:

```
    (define pi
      (lambda (lon)
        (if (empty? lon)
            1
            (if (zero? (first lon))
                0
                (* (first lon) (pi (rest lon)))))))
```

When pi is applied to a list that contains a zero, we don't traverse the list any further. However, the pending multiplications (for elements before the zero) must still be done:

```
    (pi (join 1 (join 0 (join 2 empty))))
==> (* 1 (pi (join 0 (join 2 empty))))
==> (* 1 0)
==> 0
```

For a long list, this could be wasteful; we would instead like a way of return the value 0 directly. Can we use abort? Consider:

```
    (define pi
      (function (lon)
        (if (empty? lon)
            1
            (if (zero? (first lon))
                (abort 0)
                (* (first lon) (pi (rest lon)))))))
```

But now suppose we have:

```
    (+ 1 (pi (join 1 (join 0 (join 2 empty)))))
==> (+ 1 (* 1 (pi (join 0 (join 2 empty)))))
==> (+ 1 (* 1 (abort 0)))
==> 0
```

which is probably not what we wanted: the abort ignores *all* context, including the pending addition waiting outside the call to pi. We would have preferred the reduction to be of the form

```
    (+ 1 K (* 1 (jump K 0)))
==> (+ 1 0)
==> 1
```

where K was some magically inserted marker, and jump sent the value of its second argument to the mark labelled by the first.

How do we get access to the context surrounding the second argument to the addition? For this, we introduce another operation, which we call `let-cc` (which is like a `let`, except it binds the current context). It has the form

```
    (let-cc context body)
```

and works as follows: it converts the context surrounding the `let-cc` body into a function object. This is then bound to `context`. The function takes one argument; invoking it with an argument passes that argument to the context that was current at the time the `let-cc` was encountered.

Consider the following examples:

```
    (let-cc cc (cc 3))
```

This reduces as follows:

```
==> ((function (v) (abort v)) 3)
==> (abort 3)
==> 3
```

Next,

```
    (+ 1 (let-cc cc (+ 2 (cc 3))))
==> (+ 1 (+ 2 (cc 3)))
```

At this point, cc is bound to `(function (v) (abort (+ 1 v)))`:

```
==> (+ 1 (+ 2 ((function (v) (abort (+ 1 v))) 3)))
==> (+ 1 (+ 2 (abort (+ 1 3))))
==> (+ 1 (+ 2 (abort 4)))
==> 4
```

Note that the addition of 1 takes place only once, though it appears twice in the

context.

```
    (+ 1 (bind-cc cc (* 1 (cc (* 10 (cc 0))))))
```

The context is the same as above:

```
==> (+ 1 (* 1 (cc (* 10 (cc 0)))))
==> (+ 1 (* 1 ((function (v) (abort (+ 1 v)))
                (* 10 (cc 0)))))
==> (+ 1 (* 1 ((function  ... ) (* 10
                                   ((function (v) (abort (+ 1 v))) 0)))))
==> (+ 1 (* 1 ((function  ... ) (* 10 (abort (+ 1 0)))))
==> (+ 1 (* 1 ((function  ... ) (* 10 (abort 1)))))
==> 1
```

Hence, when there are multiple calls to a context procedure, the one whose argument first reduces to a value is the one that returns: the other pending ones, like all other pending operations, are ignored.

Also, it is important to note that, whereas previously we could be sure that when we invoked a function, if it terminated, we would return to the point of invocation, this is no longer true for ``functions'' that are really contexts, due to the implicit `abort`.

Thus, we see that `let-cc` is what we really want to write Pi: (Editor's Note: This file has been garbled since it was first published in December 1994, and we are unable to recover about 1 paragraph or so of material at this location in the article. We apologize for any confusion this causes.)

Finally, we have alluded in various places to Scheme's minimalism. Most parts of the language serve a specific and significant purpose, and usually cover it in sufficient generality so that most common uses of it are special cases. For instance, `function` produces procedural abstraction; `let` binds names; recursion obviates the need for various iterative constructs; closures implement lexical scope; and `let-cc` abstracts several patterns of control flow. These are all fundamental issues for a language to address; special cases can be (and usually are) obtained by syntactic transformations (or ``macros'').

There is, naturally, much more to Scheme than we have let in on above. In particular, we have almost exclusively studied what is sometimes known as ``Core Scheme.'' Scheme, however, is an imperative language with rich data structure facilities in which a wide variety of applications can be written. But most Scheme programs will fundamentally depend upon the ideas we have introduced above.

Syntax

We finally note that the syntax we have adopted above is slightly different than that of standard Scheme. To wit,

Write	As
empty?	null?
function	lambda
first	car
rest	cdr
join	cons
empty	'()

Replacing `let-cc` is a little more difficult. If your Scheme has a form named `let-cc`, `let/cc` or `letcc`, use that. Otherwise, write

```
(let-cc name body)
```

as

```
(call-with-current-continuation (lambda (name) body))
```

(Some Scheme systems helpfully truncate the aforementioned function to name `call/cc`. Check with your local system.)

With these substitutions, any of the programs in this article can be entered and evaluated in a Scheme system.

Bibliography

There are several fine books and articles that describe or use Scheme. We list below a few that we have found particularly inspiring, challenging or useful.

Abelson, H., and Sussman, G.J., with Sussman, J. *Structure and Interpretation of Computer Programs.* MIT, Cambridge, Mass., 1985. (Here is a link to the **second edition, 1996**)

Dybvig, R.K. *The Scheme Programming Language.* Prentice-Hall, Englewood Cliffs, NJ, 1987. (Here's a link to closely related book since the one listed above is out of print. The link here is for **The Scheme Programming Language: ANSI Scheme; 1996**)

Friedman, D.P., and Felleisen, M. ***The Little Lisper.*** MIT, Cambridge, Mass., 1987.

Friedman, D.P., Wand, M., and Haynes, C.T. ***Essentials of Programming Languages.*** MIT, Cambridge, Mass., 1992.

Springer, G., and Friedman, D.P. ***Scheme and the Art of Programming.*** MIT, Cambridge, Mass., 1989.

Steele, G.L., Jr., and Gabriel, R.P. The Evolution of Lisp. In *The Second ACM SIGPLAN History of Programming Languages Conference* (April 1993, Cambridge, Mass.) ACM/ SIGPLAN, New York, 1993, pp. 231-270.

Steele, G.L., Jr., and Sussman, G.J. *The revised report on Scheme, a dialect of Lisp.* Memo 452, MIT Artificial Intelligence Laboratory, Cambridge, Mass., 1975.

Shriram is a graduate student at Rice University. He welcomes electronic mail at *shriram@cs.rice.edu*.