Mattox's Quick Introduction to Scheme

or, What Do People See in This Language, Anyway?

©November, 2000
Mattox Beckman

1 Introduction

The purpose of this document is to give the reader an introduction to both the Scheme programming language and some of the techniques and reasoning used in functional programming. The goal is that the reader will be comfortable using Scheme and be able to write programs in functional style. The reader is assumed to be a CS321 student: someone who has written some large programs in C++ or Java, and is at any rate familiar with imperative or object oriented languages; but who is not familiar with functional programming. You will get a lot more out of this if you have a Dr. Scheme window open, try out the examples, and make up some of your own as you go along.

The outline of this paper is as follows: Section 2 briefly discusses four major language paradigms. Section 3 introduces lists and list operations.

2 Language Paradigms

There are four major language paradigms in use today, which have different definitions as to what constitutes a program.

- Imperative: Languages like C, Fortran are called *imperative*, meaning "command", because a program is thought of as being a list of commands for the computer to execute.
- Functional: Languages like SML, Haskell, Lisp, and Scheme are common examples. These language involve "programming with functions." A program is considered an expression to be evaluated.
- Object Oriented: Languages like Smalltalk, in which a program is a collection of objects which communicate with each other. C++, Java, and Haskell are also considered Object Oriented, though it may be more accurate to refer to them as hybrid languages.
- Logic: The most famous example is Prolog, in which you program using predicates, which are a functions which return a true or false value.

3 Types

The first thing to know about Scheme, and most other functional programming languages, is that the notion of "base type" is very different than in most imperative langauges. In C++, your base types are integers, various kinds of floating point representations, and pointers. You are allowed to compose these to make bigger types, i.e., classes and structs. By *base type*, we mean to say that these can be passed as arguments to functions, returned as the result of a function, and operated upon to make other values.

In a functional language, you have the same base types as the imperative languages, but we add two more: lists[1] and functions. Scheme also adds *symbols*: just about anything can be made into a symbol if you put a single-quote in front of it. In C++ you have to define a class if you want a list data-structure. In Scheme you start out with a list structure—in fact, the language itself is written using lists. If you want to get anything done using this language, you need to become an expert at manipulating lists.

[1] Actually, you don't have to add lists. It just happens that most functional languages do.

A list is a recursive data-structure. In Scheme, a list is either `null`, or else it is an object (itself possibly a list) together with another list. A list is represented by an open parenthesis, followed by zero or more elements, followed by a close parenthesis. If you type a list like this into the Scheme interpreter, it will assume that the first element of the list is a command, and that the following elements are arguments to the command.

```
> (+ 3 5 10)
18
> (write "Hello, world!")
"Hello, world!"
```

In the above example are two lists. The first, `(+ 3 5 10)`, has four elements. The first element `+` is taken as a command, and the remaining elements are taken as arguments. This does the expected addition. The second list, `(write "Hello, world!")`, has two elements. The command `write` prints the remaining elements onto the screen.

One special command you need to know is the `define` command. If you wanted to save the result of the previous addition, you could do it by using

```
> (define result (+ 3 5 10))
> (write result)
18
```

Finally, note that lists can be *nested*: an element of a list can be a list itself. Consider:

```
> (+ (* 2 3) (* 4 5))
15
```

The main list has three elements, and the final two elements are themselves three-element lists.

3.1 List Operations

3.1.1 Creating Lists

This is all very nice, but what if you don't want the first element to be taken as a command? You need some way to construct your own lists.

The first way is to use the quote character.

```
> (define alist '(+ 3 5 10))
> (write alist)
(+ 3 5 10)
```

The quote has kept this list from being evaluated.

Another way is to use the `list` operator. (The quote in front of `+` is to make it into a symbol, instead of a function. Try this example without and see what happens.)

```
> (define blist (list '+ 3 5 10))
> (write blist)
(+ 3 5 10)
```

They do very similar things. The quote operator is more compact, but the list operator allows computation to occur inside the list first.

```
> (write '(+ (* 2 3) (* 4 5)))
(+ (* 2 3) (* 4 5))
> (write (list '+ (* 2 3) (* 4 5)))
(+ 6 20)
>
```

The final way is to use `cons`, which means *construct*. The first argument is an element for the list, and the second argument is the tail of the list. The word `null` represents an empty list.

```
> (define clist (cons 12 blist))
> (write clist)
(12 + 3 5 10)
> (define dlist (cons 1 (cons 2 (cons 3 null))))
> (write dlist)
(1 2 3)
```

Actually, there is no rule that says that the second argument to `cons` has to be a list. You can say `(cons 3 4)` if you want. What you get is a *pair*, and not a list. Another notation for this is `(3 . 4)`.

3.1.2 Car, Cdr, and friends

To accesses the parts of a list we have the functions `car` and `cdr`. The function `car` gives you the first element of a list. The function `cdr` gives you the remainder of the list.[2]

```
> (car dlist)
1
> (cdr dlist)
(2 3)
```

Note, it is an error to take the `car` or `cdr` of an empty list.

When you have a nested list, it is often the programmer's intent to take the `car` or `cdr` of the result.

```
> (define nestlist '( (10 20) (a b c) x))
> (car nestlist)
(10 20)
> (car (car nestlist))
10
> (cdr nestlist)
((a b c) x)
> (car (cdr nestlist))
(a b c)
> (car (car (cdr nestlist)))
a
```

This can get tedious quickly, so we have a shorthand notation: instead of `(car (car l))`, you can say `(caar l)`. To take the `car` of the `cdr` of the `car` of a list, you could just take the `cadar` instead.

[2] The names of these function are historical. The `car` function is pronounced "car," and is short for "contents of address register." The function `cdr` is pronounced "could-er," and is short for "contents of decrement register." These were registers used to store the head and tail of a list in the original implementation of Lisp, the precursor to Scheme.

```
> (define nestlist '( (10 20) (a b c) x))
> (car nestlist)
(10 20)
> (caar nestlist)
10
> (cdr nestlist)
((a b c) x)
> (cadr nestlist)
(a b c)
> (caadr nestlist)
a
```

3.1.3 Length and null?

Two other things you will need to do to make use of lists is to find out if the list is null, and find out the length of a list.

```
> (define anemptylist '())
> (null? anemptylist)
#t
> (null? alist)  ; remember alist is '(+ 3 5 10)
#f
> (length anemptylist)
0
> (length alist)
4
```

Another function you may want to use is `list?`, which checks whether something is a list.

4 Techniques I — Values and Recursion

4.1 Immutability

In Scheme, variables are *mutable*: as in C++, you can change them if you want, by using `set!` (pronounced "set-bang"). *Don't do this!* In functional programming, "destructive update" has the same negative connotations as the `GO TO` statement (which most of you are probably too young to remember) in BASIC, and multiple inheritance in C++: to use it indicates that you are either doing something very specialized, or else you don't understand what you are doing very well at all.

"Okay then," you might object, "if I'm not supposed to change anything, how do I get anything done?" The answer is this: instead of destroying old data, you make a copy of it, and modify the copy as you create it.

Here's an example, which will also show you how to define a function and use the `if` expression.

```
> (define (inclist L)
    (if (null? L)
        '()
        (cons (+ (car L) 1)
              (inclist (cdr L))
        )
    )
  )
```

```
9  > (inclist '(4 5 7 8))
10 (5 6 8 9)
```

In line 1, you see the syntax for defining a function:

`(define (`*name arguments...*`)  ( `*function body*` ) )`

The name of the function is `inclist`, the single argument is `L`, and the body of the function is the stuff between lines 2 and 7, inclusive.

Line 2 shows an `if` statement. The syntax for if is:

`(if (`*boolean-expression*`) `*then-expression else-expression*`)`

You must have a *then-expression*, but the else part can be omitted. Note well that `if` is an *expression* in this language:

```
1 > (if (> 4 3) 20 30)
2 20
3 > (+ 15 (if (> 4 3) 20 30))
4 35
```

Now you have enough information to understand the `inclist` function. Line 2 checks to see if the list `L` is null or not. If it is, it returns a null list in line 3. Otherwise, lines 4 and 5 take the first element of the list, add 1 to it, and `cons` it to the result of calling `inclist` on the rest of `L`.

Too fast? Open up a Dr. Scheme window, and type the `inclist` example into the top window, including the `(inclist '(4 5 7 8))` part. Then select the "intermediate student" language option from the language menu. Finally, hit the "step" button to watch the function call happen in slow-motion.

To test yourself, try making a version that takes two arguments: the list and the number to add. Make another version that skips every other element in the list. Experiment!

Another test: write the function `nth`, which takes the n-th element of a list, like this:

```
1 > (nth 3 '(a b c d e f g))
2 'c
```

You will have to use the `=` symbol to test for equality. Assume that the input will always make sense: i.e., no empty lists will be given to it. The answer is on the next page. No peeking!

```
> (define (nth n L)
    (if (= n 1)
        (car L)
      (nth (- n 1) (cdr L))
    )
  )
```

4.2 Let

One more trick that you may find useful: the `let` statement. This is how you define "local variables" in your functions. The syntax is

`(let ( (`*variable1 expression1*`) (`*variable2 expression2*`) ...)  `*body*` )`

You can have one or more pairs of the form `(`*variable expression*`)`; it indicates that the variable is to be assigned the value of the expression. It actually creates a new variable by that name... any old variables using the same name are not affected.

Here's the `inclist` function using `let`:

```
> (define (inclist n L)
    (let ( (head (+ n (car L)))
           (tail (inclist n (cdr L))) )
      (cons head tail))
)
```

Here's a function called `lettest`. Try to predict what it does, and then type it in to see if you are right.

```
> (define (lettest n)
   (write n)
   (let ( (n (+ n 5) ) )
      (write n)
   )
   (write n)
)
```

One thing about `let`: all the variable assignments take effect only during the body of the let. In other words, the following attempt at `inclist` will not work:

```
1 > (define (inclist n L)
2     (let ( (head (car L))
3            (newhead (+ n head))
4            (tail (cdr L))
5            (newtail (inclist tail)) )
6       (cons newhead newtail)
7     )
8   )
```

This is because line 3 tries to access `head`, which is defined in line 2. That definition will not be visible until line 6 though.

If you really want to use this style, you can use `let*`. This tells it that each variable is visible to the ones defined after it in the variable list.

```
> (define (inclist n L)
    (let* ( (head (car L))
            (newhead (+ n head))
            (tail (cdr L))
            (newtail (inclist tail)) )
      (cons newhead newtail)
    )
  )
```

4.3 Accumulator Parameters

You are now ready for one of the important techniques of functional programming: accumulator parameters. Consider the following example, which has two versions of `reverse`. Try to understand how these work before reading the explanations.

```
(define (append L1 L2)  ; append two lists
   (if (null? L1)
       L2
      (cons (car L1)
            (append (cdr L1) L2))
   )
)

(define (badrev L)      ; an inefficient version of reverse
   (if (null? L)
      '()
     (append (badrev (cdr L))
             (list (car L)))
   )
)

(define (goodrev L)     ; an efficient version of reverse
   (revaux L '()))

(define (revaux L acc)
   (if (null? L)
       acc
     (revaux (cdr L) (cons (car L) acc))
   )
)
```

Both `badrev` and `goodrev` return the same result. But `badrev` does so in a terribly inefficient way: it reverses the tail of a list, and then appends the head of the list to the result. The problem is that `append` is $\mathcal{O}(n)$, because it has to find the end of the first list. Since `append` is called for each of the n elements in the list `L`, that makes this reversal function have a total running time of $\mathcal{O}(n^2)$.

The function `goodrev` shows a commonly used technique to get around this. It uses a helper function `revaux`, which takes an *accumulator parameter* called `acc`. The reversed list is built by modifying the `acc` value as we make recursive calls. Once the main list `L` is empty, the parameter `acc` will contain the answer we want, and so we just return it.

At this point, you probably have enough information to complete MP4. The things in the next section will probably turn out to be useful in MP5, and will explain why computer scientists like Scheme and other functional programming languages so much.

5 Higher Order Functions or Why Does Anybody Like This Language Anyway?

The last section discussed lists; this section discusses functions, and shows the full power of Scheme-like languages.

5.1 Functions as Data

In Scheme, you are allowed to manipulate functions the same way you can manipulate integers in other languages. Consider the following definition:

```
> (define (inc x) (+ x 1))
> (inc 3)
4
```

Pretty simple: it just takes a number and adds one to it.

Now consider this code (make sure you are in "full scheme" language mode before trying this! It won't work in the student modes.):

```
> (define (twice f x) (f (f x)))
```

What does it do? It takes two parameters: a function `f` and a value `x` that's meant to be passed to `f`. Then, it applies `f` to `x` *twice*.

You can combine it with `inc` like this:

```
> (twice inc 3)
5
```

Or, you can use it to define a new function:

```
> (define (plustwo x) (twice inc x))
> (plustwo 4)
6
```

5.2 Anonymous Functions

You can also define functions that return other functions as values. To do that you need to know how to define an *anonymous function*. The syntax for an anonymous function is:

`(lambda (`*variable*`) (`*body*`))`

Here's an example:

```
> (lambda (x) (+ x 1))
#<procedure>
> ( (lambda (x) (+ x 1)) 2)
3
```

Here, the function takes a variable `x` and returns `x` + 1. Notice that the function is not very useful on its own.

Using this, you can make an "increment function generator":

```
> (define (mkInc n)
    (lambda (x) (+ x n))
  )
> (define plusfive (mkInc 5))
> (plusfive 23)
28
```

Here, the function `mkInc` has created a new function and returned it. We saved the result to a variable `plusfive`, which we used later. This ability to compose functions is where functional programming languages get their name.

5.3 Functions and Lists

The ability to compose functions and lists together, however, is where functional programming languages get their power. Consider the following definition, and try to figure out what it's doing before looking at the explanation.

```
> (define (map f L)
    (if (null? L)
        L
      (cons (f (car L))
            (map f (cdr L)))
    )
  )
```

What will this do?

```
> (map (mkInc 3) '(2 4 6 8))
```

Answer: It will return `(5 7 9 11)`.

The `map` function is so common it is built in to most versions of scheme. What it does is take a function `f` and a list `L`, and apply `f` to each of the elements of `L`, and return the result.

There's a lot more to be said about higher order functions: this is only the beginning of what can be done with them. But I'll say more later. Until then, here's another example function to look at:

```
> (define (zip f L1 L2)
    (if (or (null? L1) (null? L2))
       '()
     (cons (f (car L1) (car L2))
           (zip f (cdr L1) (cdr L2)))
    )
  )
> (zip + '(1 2 3) '(4 5 6))
(5 7 9)
> (zip cons '(1 2 3) '(4 5 6))
((1 . 4) (2 . 5) (3 . 6))
```