CS 450

Lecture 1: Introduction to Scheme—Syntax, Special Forms, Applicative and Normal Order Evaluation

Carl D. Offner

1 Introduction

Scheme is a dialect of Lisp ("lots of idiotic silly parentheses"). It is the second oldest programming language, after Fortran. Amazingly enough, however, both are still used:

Fortran: used for scientific applications, number crunching, massive amounts of data. Compilers for Fortran can generate extremely efficient code.

Lisp: used for academic research into programming language design, computational models, prototyping, AI research, natural language studies, ...

Languages such as Fortran, Pascal, C, Algol, are examples of *imperative* programming languages.

Lisp has some imperative constructs, but its soul is *functional.* In particular, it developed originally from a computing model called the *lambda calculus*, which was invented in 1936 by Alonzo Church to help in investigating what functions could be regarded as computable.

Church-Turing Hypothesis: Those functions which are expressible in lambda calculus (which turn out to be the same as all those functions which are computable by Turing machines) constitute exactly the class of functions which we naturally regard as *computable.*

To program in a functional language such as Scheme, you really have to forget everything you think you know about programming in Java, or C, or Fortran, and start from the beginning. As we get deeper into Scheme, we will see how it can model different kinds of languages for us.

2 Some first things to know about Lisp

1. Lisp is typically interpreted rather than compiled.
2. Parentheses always have meaning in Lisp, and it is *not* the meaning you are used to.
3. Identifiers (e.g., symbols) are lexically more diverse than in most programming languages. For instance

- `*a` is a symbol.
- `* a` is two symbols (`*` followed by `a`).
- `+` is pre-defined to be the addition operator. But it could be redefined to be something else.
- On the other hand, signed numbers are a special case:

$$+3 \text{ is the number } 3$$
$$-3 \text{ is the number } -3$$

4. Lisp is an expression language. In its pure form, there are no imperative statements. Instead, the interpreter

   reads an expression,

   evaluates it, and

   prints the resulting value (which might be another expression).

3 Some examples

To bring up the UMB Scheme interpreter, simply type "scheme" at the Unix prompt. The UMB Scheme prompt is `==>`.

```
==> 27

27
==>
```

Here is a more substantive example, illustrating a procedure application (which is what most of the rest of us call a "function call"):

```
    Expression                    Evaluates to

    (+ 7 4)                       11
```

How is this read?

- The parentheses indicate a function call (in this case; they can also indicate a special form, which we will discuss later).
- The first element of the list inside the parentheses is the procedure—in this case, it is the procedure indicated by `+`, which is addition.
- The remaining elements of the list inside the parentheses are the arguments of the function. Note that in effect Scheme implements prefix notation for all functions, even for functions that we would normally write as binary infix operators, like `+`.

Here are some more examples:

```
Expression                  Evaluates to

(- 7 4)                     3
(-7 4)                      ??? (error)
(* 2 3)                     6
(/ 10 2)                    5
(/ 10 3)                    3.33333 or 10/3
```

These are called *compound expressions*. Something like (function arg1 arg2 ...) is called a *combination*.

`(+ (* 3 5) (- 10 6))` evaluates to 19

and so on, to even more complicated expressions.

3.1 Variables

```
Expression                  Evaluates to

(define size 2)             size
```

creates a variable[1] whose value is 2. Actually, normally the value never changes, so in some sense this is not a "variable". Technically, this expression creates a *symbol* (`size`) which is *bound* to the value 2.

Also note that **define** is not a function. This is also a **compound expression**, but it is not a **combination**. Instead, it is an example of a **special form**.

```
Expression                  Evaluates to

size                        2
(+ size 3)                  5
```

This is bad practice, but possible:

```
(define + 3)                +
(- 7 +)                     4
```

Of course, in such a case, we could no longer add anything!

4 Scheme syntax

This may seem very strange to you: There are no statements in Scheme! Everything is an expression. Here are (pretty much) all the different kinds of expressions:

[1]And note that the "define" expression evaluates to the name of the variable being defined. This is just a peculiarity of UMB Scheme. In fact, "define" is really not a function. As we will see below, it is a "special form", and it has no useful return value. For instance, in Dr. Scheme, you can't display the return value of a define expression.

expression

- **simple expression**
  - **constant** (e.g., `17`, `-6`, `+2.11`, `"hello"`, `#t`, `#f`)
  - **variable name** (e.g., `abc`, `count`, ...)
- **compound expression** ("compound" in this context just means "not simple") A **compound expression** is a sequence of expressions (each of which may be a simple or compound expression) enclosed in parentheses: `(expr1 expr2 ...)`
  - **procedure call** (e.g., `(+ 5 -11)`, `(square 6)`)
  - **special form** (e.g., `(define x 2)`)

Procedures are either

- **primitive** (i.e., pre-defined in the language, like `+`), or
- **user-defined** (e.g., `square`)

4.1 User-defined procedures

User-defined procedures are created by an extension of **define**:

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

We know we are defining a procedure and not a variable because a parenthesis comes after the **define**. This expression is scanned as follows:

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

            name of     formal           body of
            procedure  parameter        procedure
```

Of course, there can be more than one formal parameter. Also, the body of the procedure can consist of more than one expression. We will see examples of this later. In such a case, all the expressions in the body are evaluated, in the order written. The value of the final one is the value of the procedure application.

This expression can even easily be translated into English, as follows:

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

  To      square   something,   multiply  it by itself
```

Now with this definition, we can do the following:

```
     Expression                  Evaluates to

     (square 10)                 100
     (square (+ 2 5))            49
     (square (square 3))         81
```

Now we can make a further definition:

```
==> (define (sum-of-squares x y) (+ (square x) (square y)))

sum-of-squares
==> (sum-of-squares 3 4)

25
==>
```

(Note that we are using hyphens in the middle of symbol names. This is OK in Scheme. It wouldn't work in C or Java.)

Let's go farther:

```
==> (define (F A) (sum-of-squares (+ A 1)(* A 2)))

f
==>
```

Exactly how does the Scheme interpreter evaluate something like `(F 5)`? We'll get to that shortly. First, lets handle conditional expressions:

5 Conditionals

Booleans: `#t` `#f`

True is anything except `#f`. (So any number represents true, even 0.)

```
     Expression                  Evaluates to

     (< 2 3)                     #t
     (= +1 1)                    #t
     (< 1 -1)                    #f
```

Two special forms:

```
(if (< 2 3)
    5
    7)
```

Only evaluate the expression selected—that's why it's a special form. For instance,

```
(if (< 2 3)
    5
    (/ 10 0))
```

This does not generate an error. But if **if** were a function (rather than a special form name), it *would* generate an error.

```
(define (abs x)
   (cond ((> x 0) x)  ;;; last x could have been a sequence of expressions
         ((= x 0) 0)
         ((< x 0) (- x)) )) ;;; note unary minus
```

Evaluate only the sequence selected.

if and **cond** are expressions. For instance,

```
(define a 3)
(define b (+ a 1))
(+ 2 (if (< a b) b a))
((if (< a b) + -) a b)
```

6 Evaluating a procedure application

Here is the algorithm that the Scheme interpreter uses to evaluate procedure calls. It is called *applicative-order evaluation.* It is equivalent to *call-by-value.*

1. **Evaluate** (in any order) all the expressions in the list.
2. **Apply** the procedure (which is the evaluated first expression) to the arguments (which are the rest of the evaluated expressions).

The reason that `if`, `cond`, and `define` are called *special forms* is that even though they look like functions (in that they are the first elements of lists inside parentheses), they don't evaluate all their arguments like functions do.

Here is an example of applicative-order evaluation:

Recall some of our user-defined procedures:

```
(define (square x) (* x x))
(define (sum-of-squares x y) (+ (square x) (square y)))
(define (F A) (sum-of-squares (+ A 1)(* A 2)))
```

To evaluate `(F 5)`, we proceed like this:

```
(F 5)
(sum-of-squares (+ 5 1)(* 5 2))
(sum-of-squares 6 10)
```

```
(+ (square 6) (square 10))
(+ (* 6 6) (* 10 10))
(+ 36 100)
136
```

7 The lambda special form

Well, what we did was actually not quite correct, because it is not quite what we said we would do. We really should do something like this: Evaluate `F` and `5` to transform `(F 5)` into

```
("function having one parameter -- call it A -- and whose value
  is (sum-of-squares (+ A 1) (* A 2))"    5)
```

which then becomes (when we apply the function)

```
(sum-of-squares (+ 5 1)(* 5 2))
```

and then we proceed as before.

There is a way of doing this: we use another special form. We could have defined

```
(define F
   (lambda (A) (sum-of-squares (+ A 1)(* A 2))))
```

Thus, `(lambda (x) (<stuff>))` is an unnamed function of 1 parameter whose body is `(<stuff>)`.

In fact, `(define (f x y) (<expression in x and y>))` is really turned by the interpreter internally into

```
(define f (lambda (x y) (<expression in x and y>)))
```

8 Some examples of the use of lambda

```
==> ((lambda (x) (+ x 3)) 4)

7
==>
```

The trouble with this, of course, is that since this function has no name, it can only be used once. So the idea is that we can use **define** to give it a name:

```
==> (define f (lambda (x) (+ x 3)) )

f
==> (f 4)
```

```
7
==> (f -3)

0
==>
```

and so on. If we entered this expression

```
==> (define (f x) (+ x 3))

f
==>
```

then the interpreter actually turns it internally into the previous one:

```
(define f (lambda (x) (+ x 3)) )
```

Here is how `(F 5)` is really evaluated internally by the Scheme interpreter:

```
(F 5)                                            ;; Now eval F
((lambda (A) (sum-of-squares (+ A 1)(* A 2))) 5) ;; Now apply the function
(sum-of-squares (+ 5 1) (* 5 2))                 ;; Now eval every element
((lambda (x y) (+ (square x)(square y))) 6 10)   ;; Now apply the function
(+ (square 6) (square 10))                       ;; Now eval every element
(+ ((lambda (x) (* x x)) 6) ((lambda (x) (* x x)) 10)) ;; Now apply lambdas
(+ (* 6 6) (* 10 10))                            ;; Now eval every element
(+ 36 10)                                        ;; Now apply the function
136
```

Some more examples:

```
==> (define four 4)

four
==> (define (five) 5)

five
==> four

4
==> five

(lambda () 5)
==> (four)

*** error
==> (five)
```

```
5
==> (procedure? four)

#f
==> (procedure? five)

#t
```

Note: `procedure?` is a primitive procedure. Note that the question mark is just another character in the name. The convention in Scheme is that a procedure that evaluates to a Boolean ends in a question mark.

9 Normal-order evaluation

This is a different method of evaluating expressions. It is not the method used in Scheme, but it is very important, and we will see examples of this later on. If we changed Scheme so that it used normal-order evaluation, we would evaluate procedure applications like this:

1. **Evaluate** the leftmost subexpression of the list (i.e., the operator).
2. (a) If the procedure that is the value of that expression is primitive, then
      - **evaluate** the other subexpressions (i.e., the arguments, and
      - **apply** the procedure to the resulting evaluated arguments (as usual).

   (b) Otherwise (i.e., if the procedure is a user-defined procedure),
      - **apply** the procedure to the *unevaluated* argument expressions.

Normal-order evaluation corresponds to Algol's *call-by-name.*

Let's evaluate `(F 5)` using normal-order evaluation:

```
(F 5)                                          ;; Now eval F
((lambda (A) (sum-of-squares (+ A 1)(* A 2))) 5) ;; Now apply the function
(sum-of-squares (+ 5 1) (* 5 2))               ;; Now eval sum-of-squares
((lambda (x y) (+ (square x)(square y))) (+ 5 1)(* 5 2)) ;; Now apply fn.
(+ (square (+ 5 1)) (square (* 5 2)))          ;; Now eval arguments
(+ ((lambda (x) (* x x)) (+ 5 1)) ((lambda (x) (* x x)) (* 5 2)))
(+ (* (+ 5 1) (+ 5 1)) (* (* 5 2) (* 5 2)))
(+ (* 6 6) (* 10 10))
(+ 36 100)
136
```

You can see why normal-order evaluation is also called *lazy evaluation* or *delayed evaluation*—we don't actually evaluate anything until we absolutely need to.

Basic Fact: Whenever applicative-order evaluation yields a result, normal-order evaluation yields the same result. But there are cases when normal-order evaluation is more powerful.

For example:

```
(define f (lambda (x y) x))
```

```
(f 4 (/ 2 0))
```

The reason that Scheme uses applicative-order evaluation is:

- It is easier to implement. This is a minor reason.
- It leads to much more efficient code. This is the main reason.