

# Section Notes 1

CS51—Spring 2008

Week of February 3, 2008

## 1 Welcome to Scheme

The programming language you use affects the way you think about programming. Consequently, learning different kinds of programming languages is a good way to deepen your understanding. More practically, programming languages are tools, and like any good craftsman a good programmer must have a well-developed set of tools from which to choose.

The most frequently used programming languages these days have complex syntax, taking many pages or even whole books to describe. Not so with Scheme. In Scheme, programs consist of *expressions*. And a Scheme expression is one of two things:

- An *atom*, which is a simple thing like a symbol, string, or number.
- A parenthesized list of Scheme expressions

Notice that this definition is recursive, allowing lists to contain other lists as well as atoms. Also, we should note that Scheme is a lot more lenient than most languages in terms of which identifier names are valid; symbols can contain almost any character.

Don't be fooled by this simplicity: Scheme is not a toy language (although it can be fun). It is incredibly powerful and has been used to write very cool "real programs".

We run a Scheme program by *evaluating* an expression using a Scheme interpreter. We've said that the Scheme interpreter is conceptually a simple program, since all it does is apply the same simple rule to expressions, over and over again for each expression you give it. This rule is called the *evaluation rule*, and can be summarized as follows:

- If the expression is a constant (e.g. a number), return its value.
- If the expression is a symbol, return its definition (or error if none).
- If the expression is a list, first evaluate everything in the list. Then *apply* the function represented by the first element in the list to its arguments, which are the rest of the list.

What do you think would happen if the first element of a list were not a function?

### 1.1 Exceptions to the Evaluation Rule

Like any truly great rule, the Scheme evaluation rule has exceptions. Before stating what these are, let's try to see why we might need them.

Think about the second condition in the evaluation rule above: How does a symbol get its definition? Suppose I tell you we have a function called `define` that gives values to symbols:

```
> (define number-of-jeffs-toes 12)
```

Try applying the evaluation rule to this expression. What happens?

Conditional expressions pose a slightly different problem. The evaluation rule says we always evaluate everything in a list, but what if we want to evaluate an expression only under certain conditions that *we* specify? The canonical example:

```
> (if (zero? denom) 0 (/ num denom))
```

Getting ahead of ourselves just a little, let's consider a recursive factorial function:

```
(define (factorial n)
```

What would be the consequence of applying the evaluation rule when using this function?

`if` and other similar constructs that don't always evaluate all of their arguments are called "special forms" (short-circuiting `and` and `or` are also members of this club, to name a few).

## 2 Useful functions

### 2.1 Conditions and predicates

Functions are often structured around checking conditions. You also need to be able to check conditions in order to detect and handle errors such as bad input. Scheme provides a variety of *predicates* for this purpose: `number?`, `integer?`, `zero?`, and others.

Every value in Scheme is a truth value, i.e. can be tested directly:

```
(if x (display "yes") (display "no"))
```

In fact, every value in Scheme is true except for `#f` (false).

`and` and `or` are also available, and like their counterparts in most programming languages, perform "short circuit" evaluation, only evaluating arguments as far as necessary to determine the truth value of the expression. For this reason, `and` and `or` are frequently used for control flow as well, meaning, respectively, "while operations succeed" and "until an operation succeeds".

### 2.2 Equality

Scheme has several different notions of equality, which can take a little while to get used to. For now we will focus on `equal?` and `=`.

- `equal?`

`equal?` is able to compare any two values. It is true if and only if its arguments are equal in the sense that they would look the same when printed.

```
> (equal? "hello" "hello")
#t
> (equal? (+ 1 2) (+ 1 2))
#t
> (equal? 2 "two")
#f
```

Note that `equal?` does not care whether it *makes sense* to compare its two arguments; every pair of values is either equal or not.

- =

`=` is for numerical equality. How is it different from `equal?`? If you only compare numbers, you would not notice a difference. However, `=` requires that its arguments be numbers:

```
> (= 2 "hi")
=: expects type <number> as 2nd argument, given: "hi"; other arguments were: 2
```

What are the implications of this behavior? When should you use `=`?

## 2.3 cond

`cond` is a useful conditional form that is more general than `if`, and it should be used whenever you need to check more than one condition explicitly:

```
(cond (condition1 expr1 expr2 ... exprN)
      (condition2 expr1 expr2 ... exprN)
      ...
      (else expr1 expr2 ... exprN))
```

`cond` evaluates conditions successively. As soon as a condition is found to be true, the associated expressions are evaluated and the result of the last expression is returned. If none of the conditions are true, the expressions after `else` are evaluated instead. The `else` part is optional, but we recommend that you use it, since otherwise you will end up with an expression that sometimes evaluates to “nothing”.

## 2.4 display and begin

The function `display` prints its arguments. You should notice that something feels different about this function: we use it not for the value it returns, but for something it *does*. Such extra actions are called “side effects”.

When using a function like `display`, you often need to explicitly sequence evaluations, for example to print something then return some other value. Scheme provides `begin` for this purpose. `begin` evaluates its argument expressions in order and returns the value of the last one.

## 2.5 Defining functions

Functions may be defined using the same syntax as giving a value to a symbol:

```
> (define (square x) (* x x))
> (square 4)
16
```

Revel in the elegance of this syntax: we always tell `define` 1) what thing’s value we want to set, and 2) what we want to set it to. In this case it’s like we’re saying that, in general, the expression `(square x)` should have the value `(* x x)`, which is just what we mean.

### 3 Recursion

Make no mistake about it, you will be writing a lot of recursive functions in this course. It is always worth remembering that a proper recursive function must have:

- A base case that deals with the simplest (smallest) input the function can take.
- A recursive case that breaks the problem down, calls the function on the subproblems, and combines the results.

What functions that we've seen might be useful for detecting a base case?

The classic Towers of Hanoi puzzle lends itself nicely to a recursive solution. In this puzzle, you are given a stack of different-sized disks (piled in size order with the largest on the bottom) on one of three pegs. The object of the game is to move the disks from one peg to another, observing the rules that 1) you may only move one disk at a time, and 2) a larger disk may never be on top of a smaller disk.

It turns out that this puzzle can be solved for any number of disks, though doing so may require a prohibitive number of moves! Let's write a Scheme function that simulates the game, telling us how many moves are required for  $n$  disks.

First of all, what is the base case?

Since we know we can handle the base case, we're allowed to assume that we know how to move  $n - 1$  disks. The recursive solution to Towers of Hanoi breaks the problem down by moving the top  $n - 1$  disks first, then moving the bottom disk, then moving the top  $n - 1$  disks onto the bottom disk in its new location.

The Scheme function `towers` is listed below. It takes four arguments: the number of disks to move, the number of the peg to move them from, the number of the peg to move them to, and the number of the auxiliary peg to use in intermediate steps.

```
(define (towers n from to aux)
  (if (<= n 1) 1
      (+ (towers (- n 1) from aux to)
         (towers 1 from to aux)
         (towers (- n 1) aux to from))))
```

Points to observe and discuss:

- Pegs are represented simply as integers. If the simple solution works, go for it.
- The arguments to `if` and `+` are lined up, making the code easy to read. An exception is the number 1, which is too short to need its own line.
- Why did we use `<=` instead of `=`?
- What numerical function does `towers` compute? Is this a good way to compute that function?

Notice that we're not yet using the peg numbers (`from`, `to`, `aux`) for anything. However, including them makes the function a fairly complete description of how to solve the problem.

A common pastime among hobbyist programmers is writing programs to print out instructions for solving puzzles. How could we modify the function above to print out a human-readable description of the moves to make to solve Towers of Hanoi?

Next, imagine we want to display the solution graphically. Will our data representation need to change? Review top-down design by thinking about what functions you would write to draw the puzzle solution process.