

# Section Notes 1

CS51—Spring 2009

Week of February 2, 2009

## Outline

1. Intro to Scheme syntax
2. Evaluating expressions
3. Conditionals and special forms
4. Useful definitions
5. Lists
6. Recursion

At the end of this section, you should be able to understand how Scheme evaluates expressions, predict what simple functions will evaluate to given certain arguments, and write functions that include control flow.

## 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 an identifier, 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; they can contain almost any character, like `!`, `?`, etc. You can give identifiers a definition with `define`. For instance,

```
> (define x 2)
```

defines the identifier `x` to be 2. We can define functions as well:

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

Revel in the elegance of this syntax: we always tell `define` which identifier we want to set, and then 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.

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". In computer science, as with other fields, the most sophisticated systems are built on the simplest of primitives. We will see examples of this throughout this class.

## 2 Evaluation

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:

1. If the expression is a constant (e.g. a number), return its value.
2. If the expression is an identifier, return its definition (or error if none exists).
3. 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.

For instance, to evaluate

```
> (square 2)
```

Scheme first sees that the entire expression is a list, and applies rule 3, which evaluates each item in the list. The first item in the list is the identifier `square`. Hence by rule 2, Scheme returns the definition and applies it to the rest of the list, which consists of the number 2, yielding a final result of 4.

Make sure you understand these evaluation rules. The `square` function is intuitively obvious, but what do you think would happen if we try to evaluate a list where the first element is not a function, as in trying to evaluate `(1 2 3)`?

## 3 Conditionals and special forms

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. Consider the conditional expression

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

If we treat `if` as a normal identifier and try to evaluate this expression, rule 3 says that we must first evaluate everything in the list. Why is this behavior problematic here?

`if` does not always evaluate all of its arguments, and because of this, is called a *special form*. We call `zero?` a *predicate*, and Scheme provides other predicates for checking other conditions, like if a value is a number (`number?`) or an integer (`integer?`). A predicate is just a function that returns true or false, so you can easily write your own predicates – can you give an example?

Actually, `if` is defined for all values, not just true and false. Values are considered to be true if they aren't false. For instance,

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

prints yes for *every* value except for false. That is, it prints yes when applied to `"hi"`.

`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 the arguments evaluate to true" and "until an argument evaluates to true".

## 4 Useful Definitions

### 4.1 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?` 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) (+ 2 1))
#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 `=`?

### 4.2 `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. For instance,

```
(begin (display 'foo') 2)
```

prints "foo" and then returns 2.

## 5 Lists

Scheme provides the `list` function for constructing lists from arguments. For example:

```
> (define my-courses (list "CS51" "EC10" "Expos" "Life Sciences 1a"))
```

This statement sets `my-courses` to a list containing the four strings. The functions `car` and `cdr` allow us to access parts of a list. When we apply `car` to a list, we get back the first element of that list. When we apply `cdr`, we get the rest of the list. For example, using `my-courses` from above:

```
> (car my-courses)
"CS51"
> (cdr my-courses)
(list "EC10" "Expos" "Life Sciences 1a")
```

A very important function in Scheme is `cons`. You will learn much more about `cons` later in the course, but for now, think of `cons` as a function that adds an element to the front of a list. For example, again using `my-courses` from above:

```
> (cons "Justice" my-courses)
(list "Justice" "CS51" "EC10" "Expos" "Life Sciences 1a")
```

We can represent a list with no elements in two ways: `(list )` or `empty`. Scheme also provides predicates to test if a list is empty and to test if something is a list:

```
> (empty? (list ) )
true
> (empty? (list 1 2))
false
> (list? "hi")
false
> (list?(list 1))
true
```

We can store anything in a list, including other lists, as in:

```
(list "hi" (list 1 2))
```

Make sure you understand that this list has only two elements, "hi" and the list consisting of 1 and 2.

Lists are simple and elegant, especially when combined with recursion. Suppose we want to create a list with  $n$  llamas. Our base case is a list with 0 llamas - the empty list. The recursive case adds a llama to a list of  $n - 1$  llamas. In Scheme:

```
> (define (list-of-llamas n) (if (= n 0) empty (cons "llama" (list-of-llamas (- n 1)))))
> (list-of-llamas 0)
empty
> (list-of-llamas 1)
(list "llama")
> (list-of-llamas 4)
(list "llama" "llama" "llama" "llama")
```

## 6 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.

For instance, we can define `odd` as

```
> (define (odd? x)
      (cond ((= x 0) #f)
            ((= x 1) #t)
            (else (odd? (- x 2)))))
```

Actually, there's a mistake in this definition. It will evaluate forever when given a negative input (can you see why?). Insufficient testing of inputs is actually one of the most common bugs, so we will encourage and require thorough testing for "corner cases."

Besides =, what functions have we seen might be useful for detecting a base case?

## 6.1 The Towers of Hanoi

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 an easier version of the game, where we just want to know 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 many moves it takes 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's input is the number of disks to move.

```
> (define (towers n)
      (if (<= n 1)
          1
          (+ (towers (- n 1)) (towers 1) (towers (- n 1)))))
```

Points to observe and discuss:

- Pegs are represented simply as integers. If the simple solution works, go for it.
- The conditional test, true, and false branches of the `if` are written on separate lines for readability.
- Why did we use `<=` instead of `=`?
- What numerical function does `towers` compute? Is this a good way to compute that function?

How would this solution change if we wanted to print out a human readable description of the moves required, rather than just compute how many steps the solution takes? Hint: we will need to add additional arguments.

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.