Reading
Read Sections 4.1 through 4.4 of our textbook *Compilers.*

Written Assignment 3

1. Exercise 4.2.1 (page 206).
2. Exercise 4.2.2, part a.
3. Exercise 4.2.3, parts (a), (b), and (c).
4. Exercise 4.3.1 (page 216).

Programming Assignment 3

A moderately simple Scheme (*MSS*) expression is similar to those of our previous language VSS, but supports the Boolean type in addition to the floating-point type. This includes the literals *true* and *false*, in addition to Boolean and relational operators. There is also a conditional *if* expression. Here is the grammar:

prog	→	*expr*+
expr	→	DOUBLE
	|	BOOLEAN
	|	ID
	|	'(' RATOR *expr*∗ ')'
	|	'(' 'def' ID *expr* ')'
	|	'(' 'if' $expr_1$ $expr_2$ $expr_3$ ')'
BOOLEAN	→	'true' | 'false'
RATOR	→	ARITHMETIC | RELATIONAL | BOOLEAN
ARITHMETIC	→	'+' | '–' | '*' | '/'
RELATIONAL	→	'=' | '>' | '<'
BOOLEAN	→	'&' | '|' | '!'

The conditional expression first evaluates its first expression $expr_1$ to obtain *testVal*. If *testVal* is true then the conditional expression evaluates to the value of $expr_2$, and if *testVal* is false then it evaluates to the value of $expr_3$ (it is a semantic error if $expr_1$ is not of Boolean type).

You may wish to distinguish between Boolean and floating-point values when evaluating expressions to ensure semantic correctness, but I do not require such runtime error checking. It is also not required that you distinguish between Boolean and floating-point expressions in your grammar (e.g., your grammar need not enforce that the conditional's test expression is of type Boolean). Also, it's not required that you implement the Boolean operators (*and, or,* and *not*), but you should implement the relational operators (*equal, greater-than,* and *less-than*).

I suggest that you revise the representation of values in your interpreter. One way is to define a *Val* class capable of wrapping a Double or a Boolean value. Your symbol table, which stores variable bindings, would bind identifiers to *Val* objects, literals and variables would evaluate to *Val* objects, and operators would input a list of *Val* arguments and output a *Val* object.

As in the previous assignment (VSS language), your interpreter evaluates the top-level expressions in order and prints the value of the last expression. For programs that contain more than one top-level expression, all but the last one are generally there for side-effect. Here are some sample runs:

```
> java run
(def a (if (< 5 7) 2 4))
(* a 6)
^Z
12.0

> java run
(def flag (= 6 (+ 3 5)))
(if flag (+ 1 2) (* 3 4))
^Z
12.0
```

This illustrates semantics involving the Boolean type:

```
true             => true
(! true)         => false
(&)              => true    // identity for and
(& true)         => true
(& true false)   => false
(|)              => false   // identity for or
(| false)        => false
(> 7)            => true    // each element is > than its successor
(> 7 5)          => true
(> 7 5 2)        => true
(> 7 8 5 3)      => false
(< 4 6)          => true
(= 4 (+ 2 2))    => true
(= 3 3 7)        => false
(! (< (+ 2 2) (* 2 3)))     => false
(if true 3 4)               => 3.0
(if (< 6 3) 5 (* 6 3))      => 18.0
(if (= 3 (+ 2 1))
    (if true 4 5)
    (if false 6 7))         => 4.0
```