Reading assignment
Read Chapter 3 of our textbook *Compilers.* You can choose to skip Section 3.5 since we are using ANTLR for lexical analysis instead of Lex. You can also choose to skip Section 3.9.

Written Assignment 2

1. Exercise 3.3.2 (page 125). Describe each language informally using a sentence or two in English.

2. Write regular expressions for the following languages over the alphabet {**a**, **b**}:
   (a) The set of strings with at least three **a**'s.
   (b) The set of strings with three consecutive **a**'s.
   (c) The set of strings with an odd number of **a**'s.
   (d) The set of strings that do not contain the substring **bba**.

3. Design an NFA for each of the four languages of the previous exercise.

4. Design a DFA for each of the four languages of exercise 3. You may want to apply subset construction (Section 3.7) to convert the NFAs of the previous exercise to equivalent DFAs. If any of the NFAs your wrote for the previous exercise are DFAs, it's enough to note that fact. Alternatively, you might choose to write a DFA for each of the four languages 2(a-d) directly, and so answer problems 3 and 4 simultaneously.

Programming Assignment 2

A *very simple Scheme* (*VSS*) expression takes one of several forms: a floating-point literal whose value is itself; or an application that applies the function named by an operator to zero or more argument values to which its operands evaluate; or a *define* expression; or a reference to a variable. A *define* expression, introduced by the keyword *def,* takes an identifier *v* and an expression *expr;* when evaluated, it evaluates *expr* and assigns its value *val* to the identifier *v;* the value of the expression is *val.* References to *v* that appear in subsequent expressions are replaced by *v*'s value *val,* the value assigned. The most recent value assigned to *v* is its value, and references to unassigned variables have the value zero. A VSS program is a sequence of one or more expressions. Here is the syntax, with nonterminals in italics and terminals in uppercase or enclosed in single-quotes:

prog	→	*expr*+
expr	→	DOUBLE
	|	'(' RATOR *expr*∗ ')'
	|	'(' 'def' ID *expr* ')'
	|	ID
RATOR	→	'+' | '−' | '*' | '/'

A floating-point literal (DOUBLE) is represented by one or more digits, optionally preceded by a minus sign, and optionally followed by a radix point (.) and zero or more digits. An identifier ID is a letter followed by zero or more letters and digits and underscores (_). Semantically, dentifers

are case-sensitive (e.g., *apple* and *Apple* are different identifiers). Basically, this program is similar to the previous one except that (a) applications can take a variable number of operands, (b) global variables can be defined and initialized, and (c) variables can be referenced for their value.

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 (e.g., assignment to a variable). Here are some sample runs:

```
> java run
(+ 3 4 5)
^Z
12.0

> java run
(* 2 3)          // discarded
(+ 2 4 (/ 8 (+ 1 (* 2 1.5))))
^Z
8.0

> java run
(def a (+ 2 3))
(* a a a)
^Z
125.0

> java run
(def hi_there (* 7 7))
(def xyz (+ 1 2 3))
(+ hi_there xyz yup)
^Z
55.0

> java run
(def a (+ 2 2))
(def b (* a a a))
(+ a b)
^Z
68.0
```

This illustrates the semantics of the arithmetic operators:

```
(+)              => 0.0     // identity for addition
(+ 2 3)          => 5.0
(+ 3 4 5)        => 12.0
(- 5)            => -5.0    // 0-5 = -5
(- 5 4)          => 1.0
(- 5 4 3)        => -2.0    // 5-4-3
(-)              => illegal
```

```
(*)              => 1.0     // identity for multiplication
(* 2 (+ 3 4) 5)  => 70.0
(/ 4)            => 0.25    // 1/4
(/ 6 4)          => 1.5
(/ 8 4 (- 3 1)) => 1.0     // 8/4/2
(/)              => illegal
```