Fall 2017
Programming Languages
Homework 1

- Due Monday, September 25, 2017 at 5:00 PM Eastern Standard Time.
- The homework must be submitted entirely through NYU Classes—do not send by email. Submissions using NYU Classes are a multi-step process. Make sure you complete the *entire* submission process in NYU Classes before leaving the page. It is recommended that you not wait until the last minute to submit, in case you encounter difficulties.
- Late submissions are highly discouraged. Nonetheless, the following provision exists for late submissions. A late penalty of 15 points per day applies for submissions received during the first 24 hour period after the deadline. An additional 15 point penalty (total 30 points) applies for the subsequent 24 hour period. Submissions are not accepted for credit more than 48 hours after the deadline. **No exceptions will be made.**
- This assignment consists of programming tasks and "pencil and paper" questions. Submit programs according to the instructions in each question. Submit all written responses in a single PDF document.
- While you are working on this assignment, you may consult the lecture material, class notes, textbook, discussion forums, and/or recitation materials for general information concerning Flex, Bison, grammars, or any topics related to the assignment. You may **under no circumstance** collaborate with other students or use materials from an outside source (i.e., people, books, Internet, etc.) in answering specific homework questions. You may collaborate and utilize reference material for understanding the general topics covered by the homework. For example, discussing the topic of regular expressions with a classmate is permitted, as long as the discussion does not involve homework questions or solutions.

1. (10 points) **Java Language**

   A programming language's standard serves as an authoritative source of information concerning that language. Consider Java, whose syntax and semantics are partly governed by the Java Language Specification (JLS8). Let us look to the JLS8 in exploring some questions about the Java programming language. (No prior knowledge of Java is assumed for the purposes of the following questions).

   1. Floating point values in Java are stored using type `float` and `double`. These types permit the storage of values with sign and magnitude, positive and negative zeroes, and what else? Consult JLS8 §4.2.
   2. There are eight (8) *kinds* of variables in Java. What are they?
   3. If you initialize an integer variable (i.e., type `int`), can you rely on the value being initialized to zero?
   4. If you cast (i.e., convert) some primitive type to a smaller type, such as casting `int` (32 bits) to `short` (16 bits), this is called a *narrowing primitive conversion.* When specifically casting a signed integer to some other smaller integral type, what does JLS8 say will happen?
   5. Java uses the notion of *packages* and *compilation units* to modularize programs. Packages and compilation units are typically stored in files. However, *must* they be stored in files? See §7.2 of JLS8 and explain.
   6. What is the definition of an *inner class* according to JLS8?
   7. According to JLS8, exception classes derived from class `Error` have what characteristic in common? What about exception classes derived from class `RuntimeException`?

2. (15 points) **Grammars and Regular Expressions**

   1. Write a context-free grammar over the alphabet $\{a, b, c\}$ which recognizes the language "n occurrences of a followed by 2 or more occurrences of b, followed optionally by [0 or more occurrences of c], always followed by n occurrences of b."

   2. Write a context-free grammar for $\{a^i b^j c^k |\ i \neq j \vee j \neq k\}$. (The shorthand expression x^y in this context means "y occurrences of the symbol x", for some non-negative number y. For example, a^3 represents the string aaa.)

   3. Write a regular expression or grammar over the specified alphabet below, which describes a language that accepts an email address. The language described should adhere to the following requirements:

      (a) The alphabet is the set of uppercase and lowercase characters, digits from 0-9, dot ".", plus "+", hyphen "-" and at "@". By restricting the alphabet, your expression doesn't have to deal with characters beyond the alphabet.

      (b) There is a *localpart* on the left of an @ and a *domain* on the right.

      (c) Neither the localpart nor the domain may be empty.

      (d) The localpart can consist of *labels* separated by dots but there can not be two successive dots, nor can it start or end with a dot.

      (e) The *leftmost* label must not start with a digit.

      (f) No labels may start with a hyphen, end with a hyphen, nor contain two successive hyphens.

      (g) The domain may be a name (e.g. "bob.com", "bob.jones.com") which follows the same rules as localparts above, or an IP address consisting of four octets (e.g. "192.168.2.4"). Although octets in real life must fall in the range 0-255, it suffices here to ensure that the no octet exceeds the value 299. (On your own, think about how you might enforce 0-255 if asked to do so.)

      (h) The rightmost label in the domain (if using a domain name instead of octets) must follow the stricter rule that it be all alphabetic. For simplicity, you may assume the domain suffix (e.g. .com, .net, .org) is *any* three letters. So the suffix "kqx" would be considered acceptable.

      This is the complete list of restrictions for the purposes of this question. If a restriction is not present for a particular input pattern, that means the pattern is acceptable, even if it wouldn't be acceptable in "real life."

      If you choose to write a regular expression, you may use common regular expression shortcuts, such as character classes (e.g. [a-z] "any lowercase letter" and [ˆabc] "not a, b, or c").

3. (10 points) **Flex Scanner**

   I recommend that you wait until the September 15 recitation before attempting this question.

   Generate a scanner using `flex` which recognizes the following tokens:

   1. Keywords: `PRINT`, `IF`, `THEN`, `GOTO`, `INPUT`, `LET`, `GOSUB`, `RETURN`, `CLEAR`, `LIST`, `RUN`, `END`.
   2. CR : newline/carriage return (\n\r on Windows, \r on Mac and Unix)
   3. RELOP : any one of $< > =$
   4. DIGIT : any digit from 0 to 9
   5. MULDIV : * or /
   6. PLUMIN : + or -
   7. VAR : capital letter A through Z. (Just a single letter)
   8. COMMA : ,
   9. STRING : any sequence of letters, numbers, spaces, and underscores, enclosed within double quotes. Example: `"_13bl h_6"`
   10. LPAREN : (
   11. RPAREN :)

   Compile and test your scanner independently before proceeding to the next question to confirm proper recognition. These tests are for your own benefit. You do not have to submit a working program as part of this question, but you will submit your Flex ".l" input file as part of your solution to the next question.

4. (35 points) **Bison Parser**

   I recommend that you wait until the September 15 recitation before attempting this question.

   This question is inspired by a 1976 article written in Dr. Dobb's Journal[1]. The article in question presents a small language called "Tiny Basic," which is a stripped-down version of the BASIC programming language. The following is a slightly modified version of the grammar presented in the above cited article:

```
program ::= (line)+

line ::= number statement CR | statement CR

statement ::= PRINT expr-list
              IF expression RELOP expression THEN statement
              GOTO expression
              INPUT var-list
              LET VAR = expression
              GOSUB expression
              RETURN
              CLEAR
              LIST
              RUN
              END

expr-list ::= (STRING|expression) (COMMA (STRING|expression) )*

var-list ::= VAR (COMMA VAR)*

expression ::= PLUMIN term (PLUMIN term)*

term ::= factor (MULDIV factor)*

factor ::= VAR | number | LPAREN expression RPAREN

number ::= DIGIT+
```

   Write a Bison grammar that will accept the Tiny Basic language. The capitalized words are the tokens that your scanner from the previous question will accept. The lowercase words are non-terminal symbols. You'll notice that the grammar above is written in BNF. Because Bison does not accept BNF grammars, you will need to rewrite it as a non-BNF grammar. You may make whatever adjustments to the grammar that you deem necessary in doing so as long as the resulting parser properly accepts the Tiny Basic language.

   Using your Bison grammar and Flex scanner from the previous question, generate a C program that correctly parses the above language and outputs whether the parsing succeeded or failed. Your parser is only checking for well-formedness—it should not concern itself with semantics, and is therefore not expected to perform any actual computation beyond accepting or rejecting an input program.

   Bison doesn't support BNF grammars. Therefore in formulating your solution, you will need to convert the grammar from BNF form to non-BNF. For example, a BNF production:

   x ::= y*

   might be expressed in Bison as:

[1]Dr. Dobb's Journal of Computer Calisthenics & Orthodontia, Volume 1, Number 1, 1976, p. 9.

```
x : /* epsilon */
  | y x
  ;
```

Note: the absence of anything on the right hand side denotes an ϵ-transition. The comments are disregarded by Bison. Try to use ϵ transitions sparingly as they tend to give rise to reduce/reduce errors.

Submit the following files:

1. Flex file ending in .l
2. Bison file ending in .y
3. A shell script for building the scanner/parser
4. A sample Tiny Basic program which is accepted by your parser.
5. README file (optional. See below.)

Submit the above files as attachments to your NYU Classes submission. The shell script should generate and compile the Flex and Bison-generated C program, thereby outputting an executable parser. You should, of course, generate the scanner and parser yourself to confirm the proper execution of your program. However, do not turn in any artifacts generated by Flex or Bison, such as C files object files, or the executable. The graders will generate these themselves by running your build script. If the idea of running a shell script to build an executable makes you cringe, you may instead use a Makefile in lieu of a shell script. In this case, your parser must be buildable on the command line by typing "make." You can assume that the `make` utility is installed.

The graders will also test your Flex/Bison-generated parser on the sample Tiny Basic program you provide (see #4 above) and also on other inputs that are known to either accept or reject the language described above. Therefore, make sure you thoroughly test your program.

If there is anything that the graders need to know about your submission, you may include that detail in a separate fourth (optional) README file. Example: if you are unable to get your program to run properly (or at all), turn in whatever files you can and use the README file to explain which parts work and what problems you encountered. If you want to direct the grader's attention to any particular aspect of your submission or provide further explanation, you may use this README file to do so.

As noted above, your task for this assignment is only to parse, but not to execute a Tiny Basic program. You may find, however, that the extra steps necessary to make the program properly interpret and execute the input program are not very far from reach if you are familiar with basic data structures and string manipulation routines made available by the C++ standard library. As a test of your skills, try implementing the full Tiny Basic semantics on your own. You may hand in this full implementation if you wish, but it is not required and no additional credit will be given.

5. (10 points) **Associativity and Precedence**

Consider a bizarre new mathematical calculator language whose rules of operator precedence and associativity defy the laws of mathematics. Consider the following precedence table, shown with highest precedence on top to lowest precedence on bottom:

Category	Operations
Additive	$+\ -$
Multiplicative	$*\ /\ \%$

Consider also the following associativity rules:

Category	Associativity
Additive	Right
Multiplicative	Left

Evaluate each expression below, assuming the usual meaning of the mathematical operators above, but using the new rules of precedence and associativity shown above. Illustrate how you arrived at the answer by writing the derivation.

1. $2 + 9 - 5 + 3 - 8$

2. $3 + 2 * 12 \ \% \ 2 + 3$

3. $8 + 6 \ / \ 2 * 3 + 4$

4. $6 * 5 \ / \ 4 + 5 - 4 * 3$

6. (10 points) **Short-Circuit Evaluation**

   I recommend that you wait until the September 18 lecture before attempting this question.

   Consider the following code below. Assume that the language in question supports short-circuit evaluation:

```
if ( f() && g() || h() )
{
  cout << "What lovely weather!" << endl;
  return ____________;
}

bool f()
{
  cout << "Hello, ";
  return ____________;
}

bool g()
{
  cout << "Darling. " << endl;
  return _____________;
}

bool h()
{
  cout << "World! " << endl;
  return ____________;
}
```

   1. Fill in the blanks above with `true`, `false` or `either` (if it doesn't matter what the return value is) to get the program to print "Hello, World!"
   2. Are C++ compilers required to implement short-circuit evaluation, according to the standard posted on the course page? You can download the standard from the course page. Note that the operator for logical OR in C++ is `||`.
   3. Are Java compilers required to implement short-circuit evaluation according to the standard? What does Section 15 of the Java Language Standard (JLS) on the course page say about it? Note that the operator for logical OR in Java is the same as C++.

7. (10 points) **Bindings and Nested Subprograms**

I recommend that you wait until the September 18 lecture before attempting this question.

Consider the following program:

```
program main;
  var a, b : integer;

  procedure sub1;
    var x, y : integer;
    begin {sub1}
    ...
    end; {sub1}

  procedure sub2;
    var x, b, t: integer;

    procedure sub3;
      var y, a: integer;
      begin {sub3}
      ...
      end; {sub3}
    begin {sub2}
    ...
    end; {sub2}

begin {main}
  ...
end {main}
```

Complete the following table listing all of the variables, along with the program units where they are declared, that are visible in the bodies of sub1, sub2, and sub3, assuming static scoping is used.

Unit	Var	Where Declared
sub1	x, y a, b	sub1 main
sub2		
sub3		