Spring 2021
Programming Languages
Homework 1

- Due Wednesday, February 17, 2021 at 11:55 PM Eastern Standard Time.
- The homework must be submitted entirely through NYU Classes—do not send by email. Submissions using NYU Classes are a multiple step process including a confirmation step. Make sure you complete the *entire* submission process in NYU Classes before leaving the page. I do not recommend waiting until the last minute to submit, in case you encounter difficulties.
- Late submissions will not be accepted. **No exceptions**.
- This assignment consists of programming tasks in Flex/Bison and also "pencil and paper" questions. Submit programs according to the instructions in question 4. Submit all other written responses in a single PDF document.
- Your Flex and Bison assignments must be tested to execute properly on the CIMS computers using Flex 2.6 and Bison 3.7. Use the commands "module load bison-3.7" and "module load flex 2.6" on the CIMS computers to use these versions. You can use any system you want to perform the development work, but in the end it must run on the CIMS machines. All students registered in this course who do not already have accounts on CIMS should have received an email with instructions on how to request an account.
- 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. However, you may collaborate and utilize reference material for understanding the general topics covered by the homework. For example, discussing the topic of regular expressions or the C/C++ programming languages with a classmate is permitted, as long as the discussion does not involve homework questions or solutions.

1. (15 points) **Language Standards**

   A programming language's standard serves as an authoritative source of information concerning that language. Someone who claims expertise in a particular programming language should be thoroughly familiar with the language standard governing that language.

   In the exercises below, you will look into the language standards of several well-known languages, including C++, Java, and C# to find the answers to questions. You should use the standards documents on the course web page, which contain the most up-to-date publicly available documents[1]

   In your answers to **every question below**, you must cite the specific sections and passages in the relevant standard(s) serving as the basis for your answer. No prior knowledge of either language is assumed nor expected, but you cannot simply provide the answer with no evidential support or you will lose credit.

   Some questions may be easier to answer after attending Lecture 2. You are encouraged, but not required, to wait until that lecture before answering the questions. Those questions are so noted below.

   1. Conventionally, Java packages are stored in a hierarchically way that mirrors the structure of the underlying file system. But does the Java language *require* this? Where else might packages be stored?
   2. The Java language has a keyword **extends** to signify inheritance. Does JLS 15 allow you to extend any class you want? What about `class Enum`?
   3. (Lecture 2) We know that, generally, variables in an outer scope can be *hidden* by variables in an inner scope. What about class and instance variables in Java? Can those be hidden? Cite the location(s) in JLS 15 where this question is answered.
   4. (Lecture 2) In C++, one may have two distinct entities with the same name, in the same scope, such that one name hides the other. How is that possible? (Normally, name hiding only occurs when the names are in *different* inner and outer scopes.) What does section 6.3.10 of the C++ standard have to say about this?
   5. (Lecture 2) What does it mean in C++ when an identifier is *visible*?
   6. (Lecture 2) What does it mean in C when an identifier is *visible*?
   7. (Bonus for Extra Credit) Create a small source code example, based only on your answers to the two questions immediately above, and point to a specific location your example where a particular identifier would be "visible" in C++, but not in C. Give a brief explanation of your source code example.
   8. (Lecture 2) In C++, the *accessibility* of a name (e.g. identifier) refers to whether or not a name can be accessed (i.e., used) from a particular location and is a notion that is specific to classes. Accessibility is an extra layer of "security" on top of visibility, which may reduce access to something that would otherwise be accessible.

      Most programmers are generally familiar with the keywords **private**, **public**, **protected**, etc., as these words show up in many object-oriented languages. These are referred to as access specifiers in C++.

      What is the default level of accessibility of a name in C++ if no access-specifier is provided by the programmer? Cite to the location in the standard where the answer is provided, even if you know the answer already.
   9. This builds upon our understanding of accessibility above. In C++, the most common type of inheritance in an object-oriented program is *public inheritance*, denoted by the notation:

      ```
      class A : public B  {...}
      ```

[1] The C++20 standard is a copyrighted document that must be purchased, but the earlier 2017 working draft is free. We use the 2017 working draft for the purposes of this assignment and course.

Here, B is the *base class* and A is the *subclass*. Public inheritance means that `public` members of class B are accessible as public members of class A and `protected` members of class B are accessible as `protected` members of class A.

However there is also *protected inheritance*. For this we write:

```
class A : protected B  {...}
```

According to the C++ standard, what does this mean?

10. C# has a special control structure called a **foreach** loop which, interestingly enough, was the inspiration for one of C++'s two **for** loop variations. Unlike the more traditional **for** loop in which the programmer can iterate over any condition they choose, the **foreach** loop imposes several restrictions on what the loop can do. Explain two ways in which **foreach** is different from the more general **for** loop.
11. Those who have studied C++ may already know that the compiler provides default constructor if one is not explicitly defined. It turns out that C++ may provide a default implementation for up to *six* (6) different methods if they are not explicitly defined by the program. What are the methods?

2. (15 points) **Grammars and Parse Trees**

1. For each of the following grammars, describe (in English) the language is described by the grammar. Describe the *language*, not the grammar. Example answer: "The language whose strings all contain one or more a's followed by zero or more b's followed optionally by c." Each grammar has only one non-terminal, S, serving as the start symbol. Terminal symbols are enclosed in quotes.

   a) S → 0 S 1 | 0 1
   b) S → '+' S S | '-' S S | a
   c) S → S '(' S ')' S | ϵ
   d) S → a S b S | b S a S | ϵ
   e) S → a | S '+' S | S S | S '*' | '(' S ')'

2. Consider the following grammar, where $\langle S \rangle$ is the start symbol:

   $\langle S \rangle$ ::= $\langle V \rangle$ '`=`' $\langle E \rangle$
   | $\langle S \rangle$ '`;`' $\langle S \rangle$
   | '`if`' $\langle B \rangle$ '`then`' $\langle S \rangle$
   | '`if`' $\langle B \rangle$ '`then`' $\langle S \rangle$ '`else`' $\langle S \rangle$

   $\langle B \rangle$ ::= $\langle E \rangle$ '`===`' $\langle E \rangle$

   $\langle V \rangle$ ::= '`x`' | '`y`' | '`z`'

   $\langle E \rangle$ ::= $\langle V \rangle$ | '`0`' | '`1`' | '`2`' | '`3`'

   Consider the input "if x === 1 then if y === z then z = 2 else z = 3."

   a) Demonstrate the ambiguity of the grammar by drawing as many parse trees as you can (at least 2 required).
   b) Write a new unambiguous grammar that generates the same language.
   c) Redraw the parse tree using the new grammar for the same string as above (there should now only be one) .

3. Consider the following grammar:

   $\langle G \rangle$ ::= $\langle S \rangle$ $$

   $\langle S \rangle$ ::= $\langle A \rangle$ $\langle M \rangle$

   $\langle M \rangle$ ::= $\langle S \rangle$
   | *epsilon*

   $\langle A \rangle$ ::= a $\langle E \rangle$
   | b $\langle A \rangle$ $\langle A \rangle$

   $\langle E \rangle$ ::= a $\langle B \rangle$
   | b $\langle A \rangle$
   | *epsilon*

   $\langle B \rangle$ ::= b $\langle E \rangle$
   | a $\langle B \rangle$ $\langle B \rangle$

   a) Describe in English the language that the grammar generates.
   b) Show a parse tree for the string `a b a a`.

4. Write a context-free grammar for each of the following languages:

   a) Strings of 0's and 1's with an unequal number of 0's and 1's (any order).
   b) Strings of 0's and 1's that do not contain the substring 011.

c) Strings of 0's and 1's whose value is strictly greater than 5 when interpreted as a binary number.

5. Consider the following grammar:

$\langle A \rangle$ ::= ABd
| Aa
| a

$\langle B \rangle$ ::= Be
| b

a) First, describe the language generated by this grammar.

b) The above grammar cannot be parsed by an LL parser. Explain why. Refer to the lecture slides if you have difficulty.

c) Rewrite the grammar so that it can be parsed by an LL parser. You may use ϵ transitions if you wish.

3. (10 points) **Regular expressions**

   1. For each of the following, write a regular expression using only the constructs shown in class. Do not use non-regular features such as forward reference and back reference. If you want to use a regex shortcut, such as "\d," you should specify the intended meaning of that shortcut to be absolutely clear about your intent. Assume that all expressions will be interpreted using "lazy" semantics.
      (a) Write a regular expression recognizing strings that contain *at least* three consecutive 5's.
      (b) Write a regular expression recognizing strings that contain exactly three 5's anywhere within the string.
      (c) Write a regular expression recognizing strings whose length is a multiple of 5.
      (d) Write a regular expression recognizing even integers.
      (e) The *parity* of a number is the sum of the digits of the number. For example, the parity of 212 is $2 + 1 + 2 = 5$ (odd) and the parity of 507 is $5 + 0 + 7 = 12$ (even). Define a regular expression that only matches numbers with *even* parity. You may accept numbers with leading zeroes.
      (f) Write a regular expression to accept dates of the form MM-DD-YYYY. The accepted dates must be valid. For example, 02-30-2018 is not valid because the date doesn't exist. 2-1-2018 is also not valid due to the single digit day and month.
   2. Outside of this class, you may have encountered a regular expression operator $\{x\}$ and also $\{x, y\}$ where x, y are integers. This refers to a specific number of occurrences or a specific range of occurrences of a regular expression. For example $x\{3\}$ specifies that regular expression x occur exactly 3 times. As another example, $x\{3, 5\}$ specifies that regular expression x occur between 3 and 5 times. Explain why these operators are closed under regular expressions (i.e., using these operators in a regular expression produces a regular language). This can be done by offering a construction for converting expressions of this form to equivalent expressions which are restricted to only the operators discussed on the lecture slides. You do not have to offer a formal proof, but your explanation must be able to convince somebody that it can be done.

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

   (a) Problem Statement

   The Bison parser generator[2] supports BNF, but not full EBNF. As we know from class, EBNF offers additional syntax to make grammars more concise without adding expressive power to the language described by the grammar. Let us recall the EBNF notation. Given valid EBNF expressions `A` and `B`, the following also yield EBNF expressions:

   - **Repetition**: `{A}` and `A*` both mean "zero or more occurrences of A," whereas `A+` means "one or more occurrences of A." For this assignment, use `<A>` instead of `{A}`, as curly braces are already reserved for other uses in Bison.
   - **Option**: `[A]` means "zero or one occurrences of A."
   - **Grouping**: `(A)` means an EBNF expression appearing as a group. For example, `(A||B)+`. Note that the group may contain alternation within it. When placing the "or" operator in a group, use the syntax `||` instead of `|` so that your parser may differentiate between Bison's native "or" operator and the new group-based "or" that appears in the middle of a Bison rule.

   We shall refer to input files that also contain the extra EBNF notation as an "EBNF-enhanced" input file—a name we are inventing for the purpose of this assignment. Any or all of the features above may be used on the right-hand-side of any rule in an EBNF-enhanced Bison input file.

   Your ultimate task is to write a translator which, given a well-formed EBNF-enhanced Bison file as input, will output a well-formed grammar acceptable to Bison (which utilizes BNF but not EBNF notation). I recommend you conduct this exercise using the following approach:

   1. **Write the Scanner** : The scanner for the EBNF-enhanced input will be generated by Flex. For your convenience, a skeleton Flex input file has been supplied with this homework. It contains the basic Flex file structure and some suggested tokens, but you need to fill in some of the regular expressions and actions. Some tokens and actions have already been supplied. Note that the file will not compile as currently written, and you are not required to make use of it.
   2. **Write the Parser** : Although this is the second step, you will need to have a Bison file containing at least the token names when you complete the first step above, because Flex requires the token names to be defined in Bison in order to return those token identifiers from the scanner. In this step, you will add the grammar rules to the Bison file. A skeleton Bison input file has been provided that contains these tokens. As before, this is provided for convenience—there is no requirement to use this file, or even these exact tokens. A couple grammar rules have been supplied to get you started. Although you may approach this any way you like, I recommend writing the grammar so that it covers the BNF subset first, and then later expand it to cover the EBNF expressions once the BNF subset has been tested to work properly. You can use the calculator examples for this testing before moving on to recognize the EBNF syntax.
   3. **Write the Translator** : The translator should accept EBNF-enhanced input and write Bison-acceptable BNF to the standard output. The first and third sections of the output file could be identical to the input file, depending on your implementation. Only the second section—the grammar rules—*must* contain modifications to the input, since Bison cannot recognize EBNF. For your convenience, a skeleton Bison file has been provided to help get you started.

      Although the Bison-generated parser constructs a tree internally, this tree is unfortunately not available for our use. To get around this, I recommend constructing your own parse tree which mirrors the parse tree constructed by Bison. The starter C++ code provided with this assignment has C++ source code illustrating how to do this. The C++ code consists of `treenode` base class and subclasses corresponding to the name of each Bison grammar rule (e.g. `prog`, `statement`, `terminal`, etc.) An instance of one of these subclasses will be the nodes of your new tree.

      A few things to note in the sample Bison file:

[2] Your solution must work using Bison version 3.5.

- The number of arguments passed to each `treenode`-derived constructor mirrors the size of the Bison production.
- The union in the first section has a single member of type `treenode` named "val." The `%token` directives below, stating that all of the terminals and non-terminals use this "val" data member to store their semantic information.
- The first step of every Bison action (the part enclosed in curly braces) is to create an instance of the class whose name is the same as the left-hand-side of the grammar rule, and assign it to the semantic value of the current production using the pseudo-variable "$$."

Following this recommended approach will produce a full tree rooted with the `prog` node. I recommend first getting your translator to display the original input EBNF grammar to the standard output, based on your tree. That is, print out exactly what was recognized without translating anything. This will validate that the tree input to your translation algorithm is correct. Once that works, the final (and arguably, the hardest) step will be to modify the tree to execute the EBNF to BNF conversion. How you modify the tree is up to you, and it does not even have to look like a parse tree when you are done. It is only used as one piece of the translator.

Every effort has been made to ensure that the details of this question are well-articulated and as complete and accurate as possible. We will not attempt to cover every possible detail in this document. However, in the event of minor uncertainties, let common sense prevail. If your uncertainty blocks you from making progress, be sure to ask questions in that case. If your question does not involve sharing solutions, please post it to the NYU Classes forum so that everyone in the class may benefit from both the question and answer.

In order to get you into the right frame of mind for writing the translator, let us consider one of the EBNF constructs ("option") and how it might be translated. You do not have to follow these suggestions. You must ultimately determine how to approach the remaining translations. Consider this EBNF-enhanced input file fragment:

```
number :  digit [ number ]
          ;
```

Your solution might translate this EBNF grammar above to the following BNF form:

```
number :  digit numopt
        ;

numopt :  /* empty */
        | number
        ;
```

Elaborating on the example, given the EBNF-enhanced Bison fragment:

```
number :  digit [ '.' digit+ ]
          ;
```

Your translator might produce[3]:

```
number :  digit digopt
          ;
digopt :  /* empty */
        | '.' digplus
        ;

digplus : digit digplus
        | digit
        ;
```

[3]These are just suggestions. You may implement your translator any way you wish, provided it is correct.

The Bison grammar you use to ultimately generate your translator must be free of reduce/reduce warnings, but may have shift/reduce warnings, provided you have thoroughly tested the program and determined that these warnings do not interfere with the proper acceptance or rejection of input strings. On the other hand, the code that your translator *generates* may contain either shift/reduce or reduce/reduce warnings, since the input EBNF that leads to the generation is not always within the direct control of the translator.

In the event of the EBNF-enhanced Bison input file is syntactically incorrect, your translator should send indicate a "syntax error" by sending the same to the standard error channel. The translator need not explain what the issue is, nor does it need to recover from the error and attempt to continue parsing. However, Bison can print the line number on which a syntax error occurs. Read the documentation for further details.

Submit the following files:

1. Flex file for your translator: <netid >.trans.l
2. Bison file for your translator <netid>.trans.y
3. A shell script for building the translator <netid>.trans.sh
4. Tiny Basic EBNF file `<netid>.tinybasic.in`, which when run through your translator, will produce `<netid>.tinybasic.y`.
5. Tiny Basic flex file `<netid>.tinybasic.l`.
6. A shell script `<netid>.tinybasic.sh` for executing the translator on the Tiny Basic input program (requireed), and then executing Flex and Bison on the translated file (optional, but recommended).
7. `README` file (optional) : see below.

If there is anything that the graders need to know about your submission, you may include that detail in the 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.

These files should be inline attachments to your NYU Classes submission. Do not turn in any artifacts generated by Flex or Bison, such as the generated C files object files, or the executables. The graders will generate these themselves by running your scripts. The graders will test your Flex/Bison-generated parser on at least the sample Tiny Basic program you provide.

In the "Resources/Course Documents" folder of NYU Classes, you will find a zip file containing the following files:

- `trans.starter.l` : a skeleton Flex file for the translator.
- `trans.starter.y` : a skeleton Bison file for the translator.
- `treenode.starter.cpp` : a starting point implementation file for a tree data structure.
- `treenode.starter.h` : a starting point header file for a tree data structure.
- `maketrans.sh` : a shell file for building the translator.

This question may be difficult for some students, so opportunities to receive a substantial amount of credit exist even if your submission is not perfect or does not fully work. The graders will consider the following points while reviewing your homework submission:

- Is the scanner fully completed and working?
- Does the parser portion of the translator properly recognize EBNF?
- How many EBNF operators does your translator successfully support?
- Does the parser at least compile and run on at least *some* input?
- How much apparent effort went into writing the solution and did the approach used make sense?

(b) Testing

The grammar below may be used to assist with your testing of your translator. It is inspired by a 1976 article written in Dr. Dobb's Journal[4]. 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+
```

You will need to take the grammar from its format above and convert it into legal EBNF format for your translator. This should be a straightforward exercise. Then run your translator from the previous exercise to validate that it successfully translates the EBNF input to BNF, in accordance with the instructions.

Applicability to real life: There are a few scenarios where you may use this knowledge from this question in the industry:

- It is common for software companies to develop products that accept competing file formats as input. For example, Google Docs accepts Microsoft Word files. This is done to offer incentive for users of the competing products to move over to your product instead. If the competing product is proprietary (i.e., not open source), you will most likely not have access to the competing product's grammar, in which case you will have to use sample input files from the competing product in order to deduce that product's grammar.
- Although this exercise happens to involve a programming language, the knowledge can be transferred to writing parsers that accept *any* formal language, such as SQL, XML, HTML5, a new format, etc. For example, you can write an XML parser that reads a corpus and feeds a machine learning algorithm with the data it just parsed.
- The financial industry, military, and certain other industries are well-known for maintaining their own version of compilers for languages like C and C++, mostly for validation purposes. You may

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

therefore find yourself working with a parser for a major language like C++, even if you don't work for a compiler company or work on compilers as your main job duty.

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 "alternative" precedence table, shown with highest precedence on top to lowest precedence on the 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, but using the new rules of precedence and associativity shown above. Illustrate how you arrived at each answer by writing the derivation (you may use parentheses.) Some answers may be fractional, in which case you can write the solution in either decimal or fractional notation.

1. $2 * 8 \; / \; 2 + 6 - 6$

2. $2 * 2 \; / \; 2 + 5 - 1 * 6$

3. $5 - 3 * 5 \; \% \; 3 + 2$

4. $10 - 2 + 6 * 10 - 1 \; / \; 3$

5. $2 + 5 \; / \; 2 * 4 - 5 \; \% \; 2$

6. $12 - 4 * 9 - 4 \; / \; 10 \; / \; 6$

6. (5 points) **Short-Circuit Evaluation**

Consider the following code below. Assume that the language in question supports short-circuit evaluation, evaluates in left-to-right order and that logical "and" (`&&`) has higher precedence than logical "or" (`||`):

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

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

bool g()
{
  cout << "World! ";
  return ____________;
}

bool h()
{
  cout << "There ";
  return ____________;
}

bool i()
{
  cout << "Darling! " << endl;
  return ____________;
}
```

1. Fill in the blanks above with `true`, `false` or `either` as necessary so the program prints, "Hello There Darling! What lovely weather!" You should write `either` if the function executes but the return value doesn't affect the output, or in the event the function never executes.
2. Are C++ compilers required to implement short-circuit evaluation, according to the standard posted on the course page? Cite the specific section where you can find the answer. 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? Cite the specific section where you can find the answer.

7. (10 points) **Bindings and Nested Subprograms** This topic will be covered during Lecture 2.

Consider the following program:

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

  procedure sub1;
    var c, a : integer;
    begin {sub1}
    ...
    end; {sub1}

  procedure sub2;
    var d, c, b: integer;

    procedure sub3;
      var a, b, e : 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
main	a b	main main
sub1	a b c	
sub2	a b c d	
sub3	a b c d e	