Fall 2021
Programming Languages
Homework 1

- Due Monday, September 27, 2021 at 11:59 PM Eastern Standard Time.
- The homework must be submitted entirely through NYU Brightspace—do not send by email. Make sure you complete the *entire* submission process in Brightspace 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 recent 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. Do not cite to web pages, books, textbooks, Stack Overflow, or any other source besides the *language standard*.

   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. In Java, what does the *same compile-time type* of two reference types mean?
   3. When using *parametrized classes* in Java, what are the different kind of type arguments that can be used? Which one is used when type parameter is a generic one rather than a specific one?
   4. When a variable is declared **final** in Java, it can be assigned a value just once (when it is unassigned). To make a variable **final**, one needs to use the **final** keyword. However, there are variables which are implicitly **final**. What are the different kind of these variables?
   5. As far as floating pointing calculations in Java are concerned, the JVM implementation is free to use extra precision where available thus giving different precision across different platforms. In order to prevent this behavior, Java allows the use of a modifier keyword. What is the this keyword? Give an example usage of this modifier.
   6. In C++, we can change the contents of a string variable (i.e., string is *mutable*). Is the same allowed in Java? If yes, give an example. If no, what is an alternative to string data type which can be used for this purpose? Cite the specific section in Java Language Specification.
   7. 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.
   8. 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?
   9. In C++, a friend of a class is a function or class that is given permission to use the private and protected member names from the class. Is friendship inherited?
   10. What does it mean in C++ when an identifier is *visible*? What does it mean in C when an identifier is *visible*?
   11. In C++, the most common type of inheritance in an object-oriented program is *public inheritance*, denoted by the notation:

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

       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:

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

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

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

12. 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.

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."
   Non-terminal Symbols: S, B, A, M
   Terminal Symbols: -, +, [,], a, b
   a) S → S S - | S S + | a | b
   b) S → a S a a | B
      B → b B | ϵ
   c) S → A a | M S | S M A
      A → A a | ϵ
      M → ϵ | M M | b M a | a M b
   d) S → a S a | b S b | a | b | ϵ
   e) S → ϵ | S S | [S]
2. Let G = (Σ, N, S, δ) be the following grammar:
   Σ (set of terminal symbols): {0, 1}
   N(set of non-terminal symbols): {S, A}
   S: root symbol
   δ (set of rules/productions):
   S → A S | ϵ
   A → A 1 | 0 A 1 | ϵ

   a) Demonstrate the ambiguity of the grammar by drawing two parse trees of the same string.
   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. a) Give a context free grammar that accepts all the strings generated by the regular expression: `(ab)+aa(ab)*a` and rejects all other strings. All the symbols (*, +, ()) have the same semantic meaning as discussed in the lecture.
   b) Show a parse tree for the string `abababaaba` using the grammar described in (a).
4. Write a context-free grammar for each of the following languages:
   a) Strings of 0's and 1's which have at least three 1's anywhere in the string(not necessarily consecutive).
   b) Strings of 0's and 1's whose length is odd and the middle symbol is 0.
   c) Strings of a's, b's and c's of the form
   $$\{a^i b^j c^k |\ i, j, k >= 0 \ and \ i = j \ or \ i = k\}$$
   d) Strings of a's, b's and c's of the form
   $$\{a^i b^j c^k |\ i, j, k >= 0 \ and \ i + j = k\}$$
5. Consider the following grammar:
   S → S + X | X
   X → X * Y | Y
   Y → a

   A grammar is said to be *left recursive* if there is a nonterminal A such that $A \Longrightarrow^+ A\ \alpha$ for some α. The grammar above is left-recursive. (Note that `Type` and `Identifier` rewrite to terminals.)
   (a) Explain in your own words why this sample grammar above is problematic for a top-down parser. (Hint: try drawing a parse tree).
   (b) Rewrite the grammar to eliminate the left recursion.

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 over the alphabet $\{a, b, c\}$ which contain exactly one 'a' along with any number of b's and c's.
      (b) Write a regular expression recognizing strings over the alphabet $\{a, b, c\}$ where all occurrences of 'a' appear in groups of three. (Please keep in mind that zero is a multiple of three).
      (c) Write a regular expression recognizing strings over the alphabet $\{a, b, c\}$ whose length is a multiple of 5 (Please keep in mind that zero is a multiple of 5).
      (d) Write a regular expression recognizing even integers.
      (e) Write a regular expression recognizing all strings of 0's and 1's not containing the substring `101`.
      (f) Write a regular expression to accept dates of the form MM-DD-YYYY. The accepted dates must be valid, except that you do not need to handle leap years (i.e., years where February has 29 days). For example, 03-32-2018 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). You can do this by offering a high level construction for converting expressions of this form to equivalent expressions that are restricted to only the regular expression operators discussed on the lecture slides. You do not have to offer a formal proof, but your explanation must be convincing.

4. (35 points) **Extended Calculator**

This question is inspired by the the calculator example previously discussed during recitation. The example presented to you had basic calculator functionality: addition, subtraction, multiplication and division. Here, you will extend this to include additional operations, including the following:

- Simple arithmetic following BODMAS rules. Example: 4 * (3 + 2) = 20
- Standard functions (ceil, modulo, floor, abs)
- Trigonometric functions (sin, cos, tan)
- Logarithmic functions ($\log_2$, $\log_{10}$)
- Unit conversions (currency, temperature, distance)
- Memory based variable stores (create and use your own variables)
- Calculator reads input from command line

The grammar of the calculator is as follows:

```
program_input ::= \*epsilon*\ | program_input line

line ::= EOL | calculation EOL

calculation ::= expr | function | assignment

constant ::= PI

expr ::= SUB expr
        | NUMBER
        | VARIABLE
        | constant
        | function
        | expr (DIV || MUL || ADD || SUB || POW || MOD) expr
        | L_BRACKET expr R_BRACKET

function ::= conversion
            | log_function
            | trig_function
            | expr FACTORIAL
            | (SQRT || ABS || FLOOR || CEIL) expr

trig_function ::= (COS || SIN || TAN) expr

log_function ::= (LOG2 || LOG10) expr

conversion ::=  temp_conversion
                | dist_conversion
                | expr (GBP_TO_USD ||  USD_TO_GBP || GBP_TO_EURO)
                | expr (EURO_TO_GBP || USD_TO_EURO || EURO_TO_USD)

temp_conversion ::= expr (CEL_TO_FAH || FAH_TO_CEL)

dist_conversion ::= expr (MI_TO_KM || KM_TO_MI)

assignment: VAR_KEYWORD VARIABLE EQUALS calculation
```

Notes:

- | denotes EBNF notation. That is, "a shorthand for the list of productions with the same left side", i.e., same notation as discussed in lecture.
- || denotes the same semantics as "or" in a regular expression. Example: A(y || z)C can be broken down into AyC | AzC.
- program_input is the start symbol of the grammar
- non-terminals are represented by small letter words (program_input, line, ...)
- terminals are denoted by CAPITAL letters (NUMBER, VARIBALE, ADD, ...)

Definition and representation of symbols:

```
EOL: one or more newlines. Unix and Linux based systems represent a newline as '\n'.
     However, windows systems use '\r\n' for the same.
PI: a constant value of 3.14
NUMBER: can be an integer as well as a decimal number.
SUB: subtraction operator (can act as unary as well as binary operator) (-)
ADD: addition operator (binary operator) (+)
MUL: multiplication operator (binary operator) (*)
DIV: division operator (binary operator) (/)
POW: power operator (binary operator) (^) [example: 2^3 = 8, 3^2 = 9]
MOD: modulo operator (binary operator) (%) [example: 2%3 = 2, 5%2 = 1].
     You don't need to worry about modulo behavior with negative numbers or x%0(undefined).
     It should work fine with non-negative integers.
L_BRACKET: (
R_BRACKET: )
SQRT: square root functionality ("sqrt" or "SQRT")
FACTORIAL: refer to the definition of factorial in mathematics (!)
ABS: absolute value ("abs" or "ABS")
FLOOR: floor function in mathematics ("floor" or "FLOOR")
CEIL: ceiling function in mathematics ("ceil" or "CEIL")
COS: "cos" or "COS" (acts on radians - use PI for this - refer to the example shown)
SIN: "sin" or "SIN" (acts on radians - use PI for this - refer to the example shown)
TAN: "tan" or "TAN" (acts on radians - use PI for this - refer to the example shown)
LOG2: log to the base 2 ("log2" or "LOG2")
LOG10: log to the base 10 ("log10" or "LOG10")
GBP_TO_USD: "gbp_to_usd" or "GBP_TO_USD"
USD_TO_GBP: "usd_to_gbp" or "USD_TO_GBP"
GBP_TO_EURO: "gbp_to_euro" or "GBP_TO_EURO"
EURO_TO_GBP: "euro_to_gbp" or "EURO_TO_GBP"
USD_TO_EURO: "usd_to_euro" or "USD_TO_EURO"
EURO_TO_USD: "euro_to_usd" or "EURO_TO_USD"
CEL_TO_FAH: celcius to fahrenheit ("cel_to_fah" or "CEL_To_FAH")
FAH_TO_CEL: fahrenheit to celcius ("fah_to_cel" or "FAH_TO_CEL")
MI_TO_KM: miles to kilometers ("mi_to_km" or "MI_TO_KM")
KM_TO_MI: kilometers to miles ("km_to_mi" or "KM_TO_MI")
VAR_KEYWORD: "var" or "VAR"
VARIABLE: a combination of small and capital alphabets, and numbers.
          Must start with an alphabet
EQUALS: assignment operator ("=")
```

Write a Bison grammar that will accept the above calculator language along with a Flex file which will handle the regular expressions. There are two cases while parsing a given expression:

- *expression is well formed (parsing is a success)* : Return the output of the expression in the next line(the output format should be same as shown in the examples below). After printing the output to the console, wait for the user to enter a new expression. This goes on until an illegal expression is entered or user terminates the application.
- *expression is not well formed (parsing is a failure)* : Output the phrase "ERROR: Undefined symbol" and exit the application.

Example:

```
1+2*3
=7.00
1 * 4 + 2
=6.00
1 * (4/2) - 5
=-3.00
10 m_to_km
=16.09
(10 m_to_km)*10 + 5
=165.93
log10 67
=1.83
abs -10
=10.00
abs 45
=45.00
5 mod 7
=5.00
5 mod 2
=1.00
var x = 6
=6.00
var y= 3
=3.00
var z=4
=4.00
(x+y)*z
=36.00
PI
=3.14
pi * 6 + 1
=19.85
sin PI/2
=1.00
sin PI
=-0.00
sin -PI
=0.00
2@3
ERROR: Undefined symbol
```

If you haven't already, notice how the extra spaces were ignored.

Your Flex and Bison output programs should generate an executable that parses the above language

and gives an output. Your parser is therefore expected to perform actual computation beyond accepting or rejecting an input expression. You will notice that the grammar above is not in BNF. Because Bison only accepts BNF grammars, you'll need to rewrite it as a 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 calculator language. We have provided a C language utility file to help you with the conversions and factorial functions. Use these conversion functions so that there is the consistency in the rates used for conversion.

Submit the following files:

1. Flex file for your calculator: <netid>.tinybasic.l
2. Bison file for your calculator: <netid>.tinybasic.y
3. A Makefile for building the calculator: Makefile
4. A sample calculator program, similar to the example above, which substantially exercises the grammar, suitable for demonstrating your working calculator when fed into the executable parser: <netid >.input.txt
5. `README` file (optional) : see below.

Submit the above files as attachments to your submission or within a single Zip file. The Makefile should generate and compile the Flex and Bison-generated C program, thereby outputting an executable parser [named <calculator>]. 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 Makefile. To be clear, your parser must be fully buildable on the command line by typing "`make`". You can assume that the `make` utility is installed.

The graders will test your Flex/Bison-generated parser on the sample calculator program you provide (see above) and also on other hidden inputs as well. 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 (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.

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	Right

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. $5 * 2 - 6 + 7 \; / \; 7$
2. $8 * 8 \; / \; 4 + 4 \; / \; 2 * 2$
3. $5 - 3 * 5 \; \% \; 3 + 2$
4. $10 - 2 + 6 * 10 - 2/6 + 4$
5. $2 + 5 \; / \; 2 * 5 - 4 \; \% \; 2$
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() && h() && i() || g() || f() && i() )
{
  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 World! Hello Darling! " 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 a, b : integer;
    begin {sub1}
    ...
    end; {sub1}

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

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