Spring 2024
Programming Languages
Homework 1

- Due Wednesday, February 14, 2024 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**. I highly recommend that you submit well in advance of the deadline to ensure that your submission is successfully transmitted.
- 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. It is okay if you scan in a handwritten document for some questions as long as everything is together in a single PDF document.
- The Flex and Bison part of this assignment requires some minor coding in either C or C++, in order to capture tokens and also specify the semantic actions whenever a rewrite rule is invoked. It is highly recommended that you get started right away on writing the scanner and grammar. This will require some time to complete properly.
- 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, C++, Java, and C# to find the answers to some basic 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 from the standard or you will lose credit. Do not cite to web pages, books, textbooks, Stack Overflow, or any other source other than the *language standard*.

   1. In Java, what is the difference between a **final** class and a **sealed** class?
   2. In Java, must a **final** variable be initialized at the same place it is declared?
   3. What are the three kinds of linkages defined in the context of identifiers in the C programming language? Explain how each kind of linkage affects the visibility and scope of identifiers within a program.
   4. What is a virtual function in C++, and how does it relate to the concept of polymorphism and overriding in object-oriented programming?
   5. 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.
   6. In C#, if a value type is declared with no initializer, such as below:

      ```
      int x;
      bool y;
      ```

      What does the C# standard say about the value of uninitialized variables? Is there a default value or is the variable undefined? If there is a default value, where can one find out what the default value is?
   7. In C#, what does it mean when a variable is declared using the keyword **dynamic**? Specifically, when is error checking performed on a dynamic variable and in what form will errors be raised?
   8. In Java, the output of the compilation process is typically a set of so-called "class files"—files in a binary format containing intermediate code, such as Bytecode. However, *must* the output of compilation be class files, or does the standard allow other formats too?
   9. What considerations and flexibilities does the host system have regarding the creation, storage, and observability of modules, packages, and compilation units in Java?

[1] For example, the C++23 standard is a copyrighted document that must be purchased at a high cost, but the earlier 2020 working draft is free. We therefore use the 2020 working draft for the purposes of this assignment and course.

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."
   Assume S is the start symbol.
   EBNF: +, [,], {, }, *, |

   a) S → {a+S[b] | bSa+} | ϵ
   b) S → 0S0 | 0S1 | 1S0 | 1S1 | 0
   c) S → A B
      A → a a A | a
      B → b b B | ϵ
   d) S → a S B c c | ϵ
      B → b B | ϵ
   e) S → a S a | b S b | a | b | ϵ

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, B}
   S: root symbol
   δ (set of rules/productions):
   S → S 1 S | A
   A → 0 A | A 1 B | ϵ
   B → 0 B | 1 B | ϵ

   a) Write a regular expression describing this language.
   b) Demonstrate the ambiguity of the grammar by drawing two parse trees for the same string. You must first identify the string. The leaves of both parse trees must match the string.
   c) Write a new unambiguous grammar that generates the same language.
   d) Redraw the parse tree using the new grammar for the same string as above (there should now only be one).

3. Consider the language consisting of strings with zero or more occurrences of "p" followed by one or more occurrences of "q" followed optionally by "r". Write two grammars describing this language such that one grammar is left-recursive and the other is right-recursive. (Thought exercise, not to turn in: can you come up with even more grammars, not the same as the first two, which also describe this same language?)

4. 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).

5. a) The following BNF grammar does not use any EBNF-specific features. Rewrite the grammar to use EBNF-specific features in order to make the grammar more concise and readable. Your rewritten grammar must use EBNF symbology to the greatest extent possible (i.e., there should not exist any opportunities to make it more concise than the version you provide).
   E → E + T | E - T | T
   T → T * F | T / F | F
   F → E ** F | E
   E → D | D E
   D → 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

b) Convert the following EBNF grammar to BNF. That is, rewrite it so that all EBNF-specific symbology is removed. You will likely end up with more rules than the original. After your conversion, be sure that both grammars must describe the same language.

stmts → stmt {stmts}

stmt → val == exp

exp → val {(PLUS | MINUS | MUL) val}

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

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 in a legend to be absolutely clear about your intent. Assume that all expressions will be interpreted using "lazy" semantics. You should assume the broadest possible interpretation unless otherwise stated—see the first sub-question for an example.

1. Write a regular expression recognizing strings over the alphabet $\{a, b, c\}$ which contain exactly one 'b' along with any number of a's and c's. (For clarity, the broadest possible interpretation is that terminals a, b, and c can appear in any order, repeat, be of any length, etc., as long as there is only one 'b'.)
2. Write a regular expression recognizing the patterns of date in mm-dd-yyyy or yyyy-mm-dd format, where the months range from 01-12, the day ranges from 0-31 and the year is any 4 digit number whose first digit is 1 or 2. (You do not need special handling for months consisting of fewer than 31 days.)
3. Write a regular expression recognizing strings over the alphabet $\{a, b, c\}$ where the number of occurrences of 'a' is divisible by two. Keep in mind that zero is divisible by two.
4. Write a regular expression recognizing all strings containing alphanumeric and special characters, where each string must begin with an alphanumeric character and "." and "-" are the only special characters permitted. Special characters may occur multiple times but cannot be consecutive (i.e., each must be separated by at least one alphanumeric character.)
5. Write a regular expression recognizing all strings of 0's and 1's *not* containing the substring `010`.
6. Write a regular expression recognizing all strings over [a-zA-Z<>/] not containing the substring </.

4. (35 points) **Elementary Math Homework Checker**

Using Flex and Bison, design and implement a simple elementary math "homework checker" capable of checking the correctness of a list of equations and inequalities. Your Flex/Bison program should output a program that, when executed, accepts a list of simple equalities or inequalities and checks whether each one is correct. This exercise will involve not only parsing the input to ensure it is well-formed, but will also perform some rudimentary semantic analysis for the purpose of checking the correctness of the inputs.

Requirements:

1. Each line must begin with numbering of the form 1:, 2:, ..., and contain at least one equation or inequality. Assume that all input test cases will be sequentially numbered (i.e., 1:, 2:, ...).
2. The program will produce a similarly numbered output yielding either a "Yes" or "No" answer for each equation or inequality. Both equation and inequality are not permitted on the same line ($1 < 1 + 1 = 2$ is not permitted). The presence of both should result in a syntax error and exit.
3. Mathematical operators are limited to addition ($+$), subtraction ($-$), multiplication ($*$), and division ($/$).
4. The precedence of multiplication and division (i.e., multiplicative operators) is higher than that of addition and subtraction (i.e., additive operators).
5. Both the multiplicative and additive operators are left-associative.
6. Comparison operators are limited to less than ($<$), greater than ($>$), and equality ($=$), and have lower precedence than all math operators.
7. Expressions can only involve integers–variables are not permitted (we are not doing full algebra). The presence of variables should result in a syntax error and exit.
8. Expressions enclosed in parenthesis should have the highest precedence of evaluation.
9. Division should result in rounding down to the nearest integer ($1.33 = 1$, $1.5 = 1$, $1.75 = 1$).
10. Division by zero should result in an error message: "Error: Division By Zero." The processing of inputs should continue after printing this error message.
11. Generalized algebraic expressions are not permitted. (e.g., (1+2)(3+5) will not be considered). For the avoidance of doubt, the mathematical expressions on each side of the equality/inequality operator should be the same language as the calculator examples.

Example Input/Output:

Input 1:

```
1: 1=1
2: 1 < 2 < -1
3: 1+(2*3)+4=13
4: 1+1 < 3 > 1-2
5: 1+2-3*5=-12
6: 1/0 = 1
7: 98/9 = 10
8: 1*2-3+(1+2+3)=5
```

Output 1:

```
1: Yes
2: No
3: No
4: Yes
5: Yes
```

```
6: Error: Division by Zero
7: Yes
8: Yes
```

Input 2:

```
1: 7=2+3+4
2: 1/1=1
3: 1+1=2*1
4: 2x=6
5: 1 < 2 * 5
6: 10/5=2=4/2
```

Output 2:

```
1: No
2: Yes
3: Yes
Error: syntax error
```

If you have any issues, feel free to ask. The discussion forum on Brightspace is a suitable place for public questions. Be sure not to share the details of your implementation with other students on the forum, however.

Submission

Write a Flex scanner and Bison grammar to implement this homework checker. Ensure to handle all edge cases and error conditions as described. Submit the following files:

1. Flex file for your parser: <netid>.hwchecker.l
2. Bison file for your parser: <netid>.hwchecker.y
3. A Makefile for building the calculator: Makefile
4. `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 <hw-checker>]. 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 same inputs as provided with this homework assignment, or with other inputs that do not exceed the expressiveness of the inputs provided with this assignment. Therefore, make sure you thoroughly test your program. Also be sure that the build process operates successfully on the CIMS computers.

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 containing some strange operations. Consider the following precedence table, shown with highest precedence on top to lowest precedence on the bottom:

Operations
REV (highest)
ODD
SWP & CAT
ADD & SUB (lowest)

Consider also the following associativity rules:

Operations	Associativity
REV	Right
ODD	Left
SWP & CAT	Right
ADD & SUB	Right

Each operation is explained below:

ADD adds two numbers (e.g., 31 ADD 5 = 36)
SUB subtracts two numbers (e.g., 31 SUB 5 = 26)
CAT concatenates two numbers (e.g., 31 CAT 5 = 315)
ODD a unary operator which removes all odd digits from the number (e.g., ODD 123574567 = 246)
REV a unary operator which reverses the number (e.g., REV 123 = 321)
SWP swaps two numbers [the operands must be numbers] (e.g., 123 SWP 456 = 456123)

Evaluate each of the expressions below according to the definitions as well as the precedence and associativity rules supplied above. Illustrate how you arrived at each answer by reducing the original expression, step by step. Note the semantic restriction that all of the operands expect numbers as input:

1. ODD 32 ADD 3 SWP REV 24

2. REV 421 SUB 1 ADD 4 SWP ODD 32

3. ODD 725 SWP REV 78 SWP ODD 242 ADD 2

4. 71 CAT 6 SWP 12 SUB ODD REV 423

5. ODD (10 SWP ODD REV 921 SWP 12)

To illustrate the expected answer, here is an example question and answer using a different mathematical language which features standard mathematical operations with the usual meanings, but with precedence and associativity that are different:

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

Consider also the following associativity rules:

Category	Associativity
Additive	Right
Multiplicative	Right

The reduction for the mathematical expression $5 * 20 - 6 + 7 \ / \ 7$ is shown below. The parentheses indicate which operation will occur in the next step:
$5 * 20 - 6 + 7 \ / \ 7$
$5 * 20 - (6 + 7) \ / \ 7$
$5 * (20 - 13) \ / \ 7$
$5 * (7 \ / \ 7)$
$(5 * 1)$
5

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

Consider the following code below. Assume that the language in question supports short-circuit evaluation but with an unusual right-to-left evaluation order. Note the added parentheses, which change how the evaluation occurs. Despite the C-like syntax below, this is not C code and therefore testing your solution on a C compiler will not yield the correct results.

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

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

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

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

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

1. Fill in the blanks above with `true`, `false` or `either` as necessary so the program prints, "Hello There! Hello World! " 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. Give an example situation (other than the lecture slides) where short-circuit evaluation is very helpful and the result may differ from strict evaluation. Write a few lines of pseudo-code to show your example.

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

Consider the following program:

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

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

  procedure sub2;
    var a, c: integer;

    procedure sub3;
      var b, 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	