Fall 2023
Programming Languages
Homework 1

- Due Thursday, September 28, 2023 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++, 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. Java prohibits multiple inheritance of classes. There can nevertheless exist multiply inherited fields. How might this be possible?
   3. In Java, must a **final** variable be initialized at the same place it is declared?
   4. What does it mean when a variable is *definitely assigned* in Java?
   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 is the alphabet (i.e., character set) utilized by the Java grammar? The character set in question may evolve over time. Where does one go to find out about newer variations/versions of the character set?

[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 $\rightarrow$ {a+S[b] | bSa+} | ϵ
   b) S $\rightarrow$ 0S0 | 0S1 | 1S0 | 1S1 | 0
   c) S $\rightarrow$ a S d | B
      B $\rightarrow$ b B c | ϵ
   d) S $\rightarrow$ a S a a | B
      B $\rightarrow$ b B | ϵ
   e) S $\rightarrow$ 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}
   S: root symbol
   δ (set of rules/productions):
   S $\rightarrow$ A 1 B
   A $\rightarrow$ 0 A | A 1 B | ϵ
   B $\rightarrow$ 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 identify 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. 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. Assume two languages:
   - Language L1: has context-free grammar G1 = {$\sum$, N1, S1, δ1}
   - Language L2: has context-free grammar G2 = {$\sum$, N2, S2, δ2}

   Here, N1 and N2 are the set of non-terminal symbols. δ1 and δ2 are set of rewrite rules (productions). S1 and S2 are the root symbols. Both the grammars have same set of terminal symbols $\sum$. Assume that $N1 \cap N2 = \phi$.
   Now, we define a new context-free grammar G3 = {$\sum$, N3, S3, δ3} where $N3 = N1 \cup N2 \cup \{S3\}$, where $S3 \notin N1 \cup N2$, and $\delta 3 = \delta 1 \cup \delta 2 \cup \{S3 \rightarrow S1, S3 \rightarrow S2\}$. G3 generates the language L3.
   Prove that G3 generates the language $L1 \cup L2$. In other words, prove that $L3 = L1 \cup L2$.

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 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 subquestion for an example.
   2. 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'.)
   3. Write a regular expression recognizing strings over the alphabet $\{a, b, c\}$ where all occurrences of 'a' appear in groups of three. Keep in mind that zero is a multiple of three.
   4. Write a regular expression recognizing strings over the alphabet $\{a, b, c\}$ whose *length* is a multiple of 3. Zero is a multiple of 3.
   5. Write a regular expression recognizing even integers.
   6. Write a regular expression recognizing all strings of 0's and 1's *not* containing the substring `010`.
   7. Write a regular expression recognizing all strings over [a-zA-Z<>/] not containing the substring </.

4. (35 points) **HTML and CSS Parser**

The task at hand is to create one parser that will recognize either a subset of the Hypertext Markup Language (HTML) language and a subset of the Cascading Style Sheets (CSS) language. The user will supply the parser with either HTML or CSS input and the parser will detect which type of input it was given and parse it accordingly. The parser is to be created using the Flex and Bison tools discussed in class and recitation.

Attached to this homework are exemplary HTML and CSS files. You may assume that these input files exercise all of the features of HTML and CSS features your parser is expected to recognize. In other words, somebody handed you these files and asked you to write a parser to accept the input appearing in each. Note that the real life HTML and CSS languages are large and complex, and contain many more syntactic features than shown in the input documents. **You are not expected to write a parser that accepts full HTML or CSS input!**

HTML centers around the concept of an *element*, which consists of opening and closing tags, like so:

```
<html>  </html>
```

A handful of elements, like **link** and **br** do not require a separate opening and closing tags, but can rather be opened and closed simultaneously using this notation:

```
<br/>
```

Also, the DOCTYPE element is a special element that does not have a closing tag:

```
<!DOCTYPE html>
```

Elements can have zero or more *attributes*, which appear in the opening tag only. Here is an example featuring the attributes "rel," "type," and "href":

```
<link rel="stylesheet" type="text/css" href="homepage.css"/>
```

Here are some additional requirements:

- The parser must be capable of recognizing a document as either HTML or CSS. The document cannot contain a mixture of both. Once it has been determined that the document is HTML, for example, only HTML is permitted in the rest of the document. Similarly for CSS.
- The parser will perform no semantic analysis of either the HTML or CSS. It will be checking well-formedness only and yield a success or failure message in accordance with whether it was able to parse the input.
- Additionally, the parser will report to the standard output the depth of the deepest nested element it found during the parsing of the HTML document (for HTML input only) expressed as a number. The number 1 refers to the outermost "html" element.
- By way of example, consider: `<html><header></header><body></body></html>`, the "header" and "body" elements are both the most deeply nested ones, both appearing at depth 2. Thus, the parser will report the deepest nested element at level 2, in addition to the success/failure message. For CSS input, just the success/failure message is required.
- Although the HTML standard specifies a definitive list of legal element types (e.g., html, body, header, div, br, etc.) and attribute keys (e.g., id, href, class, etc.), you need not recognize every element type or key individually. You only need to recognize that which appears in the input files provided.
- The parser need not check if a particular attribute is legal for a given element type. So as far as the HTML portion of the parser is concerned, every element could have any number of attributes and each attribute could have any key.

- The parser should ignore any content between tags that is not an element. By way of example, consider the sequence: `<script>This should be ignored</script>`. We say "ignored" in the sense that the parser can disregard everything up to the closing tag. However, elements appearing within the opening and closing tags of other elements should be recognized.

The CSS portion of the parser recognizes a sequence of blocks of the following form:

```
h1 {
  font-size: 24px;
  font-weight: bold;
  border: 1px solid black;
  color: pink;
}
```

In this example, "h1" is the *selector*, "font-size" is a *property* and "24px" is a *value*. There could be more than one selector, seperated by commas. Selectors may be preceded by a period (.) or a pound sign (#).

For certain selectors, there are *pseudo-elements* denoted by a double-colon and followed by a name, like so:

```
p::first-line {
  color: blue;
  text-transform: uppercase;
}
```

Selectors can also be supplemented to include attribute constraints. Consider:

```
.formfield[font-size='11'] {
  font-size: 26px;
}
```

This will select elements belonging to class "formfield" whose "font-size" attribute is 11.

While implementing the parser, you may discover some non-trivial issues that need to be resolved during the design of your parser. One issue is the occurrence of '<' characters that appear between tags and the problem of distinguishing between these standalone characters and the start of an end tag (i.e,. </xxxx>). One (abbreviated) example from the input file we provided:

```
<script>
   document.write('<script src="https://js.sentry-cdn.com/7bc8bccf5c254286a99b11c68f6bf4ce.min.js"
 crossorigin="anonymous">' + '<' + '/script>');
</script>
```

Be on the lookout for this and potentially other blockers. If you have any issues, feel free to ask. (Be sure not to share the details of your implementation with other students on the forum.)

Submission

Write a Flex scanner and Bison grammar that will collectively accept the combined HTML and CSS language described above.

Submit the following files:

1. Flex file for your parser: <netid>.csshtml.l
2. Bison file for your parser: <netid>.csshtml.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 <htmlcss-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 grader will generate these himself 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 grader 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 grader needs 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()) && (h() || i() && 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 Darling! 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	