Spring 2023
Programming Languages
Homework 1 - Solutions

- Due Sunday, February 12, 2023 at 11:59 PM Eastern Time.
- Late submissions will not be accepted. **No exceptions**.
- 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.
- 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 program 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.
- The "util.h" file required in question 4 (Extended Calculator) and the source file for this pdf can be downloaded from the Assignments tab in NYU Brightspace.

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. 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?

      `Section 4.12.4 - Page 99. Three kinds of variable are implicitly declared final: a field of an interface (§9.3), a local variable declared as a resource of a try-with-resources statement (§14.20.3), and an exception parameter of a multi-catch clause (§14.20). An exception parameter of a uni-catch clause is never implicitly declared final, but may be effectively final.`

   2. 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.

      `Section 8.1.1.3 - Page 224. The effect of the strictfp modifier is to make all float or double expressions within the class declaration (including within variable initializers, instance initializers, static initializers, and constructors) be explicitly FP-strict (§15.4).`

   3. 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.

      `Section 10.9 - Page 374. In the Java programming language, unlike C, an array of char is not a String, and neither a String nor an array of char is terminated by '\u0000' (the NUL character). A String object is immutable, that is, its contents never change, while an array of char has mutable elements.`

   4. 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.

      `See Example 8.3.1.1-2 Hiding of Class Variables`
      `See Example 8.3.1.1-3 Hiding of Instance Variables`

   5. In C++, a common way to access objects created on the heap is using pointers. However, use of pointers sometimes creates memory management issues when programmers have many pointers pointing to the same object. This can result in the creation of zombie pointers. In an effort to mitigate this issue, the concept of smart_pointers was introduced in C++. Two of the most commonly used smart_pointers are unique_ptr and shared_ptr. What is the difference between these two? When should one be used over the other?

[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.

Section 23.11.1 and Section 23.11.2.2. A unique pointer is an object that owns another object and manages that other object through a pointer. The shared_ptr class template stores a pointer, usually obtained via new. shared_ptr implements semantics of shared ownership. A shared_ptr is to be used over a unique_ptr when there we need to use multiple pointers to point to the same object or when we need the objects to be CopyConstructible or CopyAssignable like cases where multiple different process/modify the data but using different pointers. A unique_ptr is used over a shared pointer when we want the data to be only modified through one pointer and this is not shared by any other pointer and it is automatically deleted when the pointer is destroyed.

6. When an exception is thrown in C++, all the existing objects within the scope need to be destroyed. What is this object destruction process called? Explain how objects are destroyed in this process. Hint: Read about constructors and destructors in the C++ language standard document.

   Section 18.2 - Page 427. As control passes from the point where an exception is thrown to a handler, destructors are invoked by a process, called stack unwinding. The destructor is invoked for each automatic object of class type constructed, but not yet destroyed, since the try block was entered. If an exception is thrown during the destruction of temporaries or local variables for a return statement, the destructor for the returned object (if any) is also invoked. The objects are destroyed in the reverse order of the completion of their construction.

7. 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?

   If a class or enumeration name and a variable, data member, function, or enumerator are declared in the same scope (in any order) with the same name, the class or enumeration name is hidden wherever the variable, data member, function, or enumerator name is visible.

8. What does it mean in C++ when an identifier is *visible*? What does it mean in C when an identifier is *visible*?

   C++:
   An identifier is visible if it is:
   1) In scope
   2) Not hidden

   Section 6.3.10
   If a name is in scope and is not hidden it is said to be visible.

   C:
   Section 6.2.1 says that an identifier is visible if it "can be used."

9. 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:

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

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

`Section 14.2 of the C++ standard states: If a class is declared to be a base class for another class using the protected access specifier, the public and protected members of the base class are accessible as protected members of the derived class.`

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.

    `See section 13.9.5 for foreach. Must iterate over a collection. Element cannot be modified while iterating. Compile time processing determines the collection type, enumerator type, and element type.`

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

      `Postfix mathematical notation with + and - and with a and b symbols`

   b) S → a S a a | B
      B → b B | ϵ

      `All strings in the language L : {a^n b^m a^(2n) | n, m >= 0}`

   c) S → A a | M S | S M A
      A → A a | ϵ
      M → ϵ | M M | b M a | a M b

      `All strings with more a's than b's`

   d) S → a S a | b S b | a | b | ϵ

      `All palindromes (a palindrome is a string that reads the same forwards and backward)`

   e) S → ϵ | S S | [S]

      `The language of strings of properly balanced left and right brackets: every left bracket can be paired with a unique subsequent right bracket, and every right bracket can be paired with a unique preceding left bracket. Moreover, the string between any such pair has the same property.`

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 for the same string.

      `For this we can generate the string 011 with two different derivations (both replacing leftmost variable first): S → AS → 0A1S → 01S → 01AS → 01A1S → 011S → 011 or S → AS → 0A1S → 0A11S → 011S → 011`

   b) Write a new unambiguous grammar that generates the same language.

      ```
      S → AS | 1S | epsilon
      A → 01 | 0A1
      ```

   c) Redraw the parse tree using the new grammar for the same string as above (there should now only be one).

3. 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, $\delta 3$} where $N3 = N1 \cup N2 \cup \{S3\}$, where $S3 \notin N1 \cup N2$, and $\delta 3 = \delta 1 \cup \delta 2 \cup \{S3 \to S1, S3 \to S2\}$. G3 generates the language L3. Prove that G3 generates the language $L1 \cup L2$. In other words, prove that $L3 = L1 \cup L2$.

We need to show that L(G3) = L3. To do this, we need to prove that $L(G3) \subseteq L3$ and L3 is a subset of L(G3).

To show that L(G3) $\subseteq$ L3, first consider any string w ϵ L(G3).
Since, w ϵ L(G3), we have S3 $\overset{*}{\to}$ w. Since the only rules in G3 with S3 on the left side are S3 $\to$ S1 and S3 $\to$ S2, we must have
S3 $\to$ S1 $\overset{*}{\to}$ w or S3 $\to$ S2 $\overset{*}{\to}$ w.
1. Suppose that S3 $\to$ S1 $\overset{*}{\to}$ w. Since, S1 ϵ N1 along with the condition N1 $\cap$ N2 = ϕ, the derivation S1 $\overset{*}{\to}$ w must only use non-terminals in N1 and rules in $\delta 1$, this implies that w ϵ L1.
2. Similarly, if S3 $\to$ S2 $\overset{*}{\to}$ w, then we must have that w ϵ A2.
Thus, w ϵ L3 = L1 $\cup$ L2, so L(G3) $\subseteq$ L3.

To show that L3 $\subseteq$ L(G3), first suppose that w ϵ L3. This implies w ϵ L1 or w ϵ A2.
1. If w ϵ L1, then S1 $\overset{*}{\to}$ w. But then S3 $\to$ S1 $\overset{*}{\to}$ w, so w ϵ L(G3).
2. Similarly, if w ϵ L2, then S2 $\overset{*}{\to}$ w. But then S3 $\to$ S2 $\overset{*}{\to}$ w, so w ϵ L(G3).
Thus, L3 $\subseteq$ L(G3).

Since we previously showed that L(G3) $\subseteq$ L3, it follows that L(G3) = L3.

4. Give a context-free grammar (CFG) for each of the following languages over the alphabet $\sum$ = {a, b}:

   a) All strings in the language L :
   $$\{a^n b^m a^{2n} | \; n, m >= 0\}$$

   S $\to$ aSaa | B
   B $\to$ bB | ϵ

   b) All nonempty strings that start and end with the same symbol.

   S $\to$ aXa | bXb | a | b
   X $\to$ aX | bX | ϵ

   c) All strings with more a's than b's.

   S $\to$ Aa | MS | SMA
   A $\to$ Aa | ϵ
   M $\to$ ϵ | MM | bMa | aMb

   d) All palindromes (a palindrome is a string that reads the same forwards and backwards).

   S $\to$ aSa | bSb | a | b | ϵ

5. Consider the following grammar:
   S $\to$ S + X | X
   X $\to$ X * Y | Y
   Y $\to$ 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.

```
S → X S'
S' → + X S' | ε
X → Y X'
X' → * Y X' | ε
Y → a
```

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.

      `[bc]*a[bc]*`

      (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).

      `[bc]*((aaa)[bc]*)*`

      (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).

      `((a|b|c){5})*`

      (d) Write a regular expression recognizing even integers.

      `0|([-+]?[1-9][0-9]*(0|2|4|6|8))|([-+]?(2|4|6|8))`

      (e) Write a regular expression recognizing all strings of 0's and 1's not containing the substring `101`.

      `0*(1*000*)*1*0*`

   2. Consider the statement: For any regular expression R, it is always true that:

      `((R* R) | R)* = R*`

      where, $*$ and $|$ represent their usual regular expression meanings. Do you agree with this statement? If yes, give a formal proof for the same. If no, give at least one counterexample.

      ```
      We have:
      R is a subset of (R* R), so ((R* R) | R) = R* R = R+.
      And note that by definition (R+)* = R* , since both these two sets contain exactly
      all strings that are concatenations of a few elements of R.
      ```

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 | 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 mi_to_km
=16.09
$ (10 mi_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  /* For this question, we assume that -0.00 is equal to 0.00 or 0,
        i.e., don't worry if this is not an exact match with your calculator.
        Floating point computations don't need to be an exact match
        to the reference calculator*/
$ 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 "util.h" 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>.calc.l
2. Bison file for your calculator: <netid>.calc.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 [<netid>_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.

Some `FAQs`:

1. Question: *It seems like having the grammar expr = function and function = SQRT expr causes shift reduce errors. How do I fix this while still abiding by the given grammar rules?*
Answer: You can leave the shift/reduce warning as it is. However, make sure there are no reduce/reduce conflicts. The grammar given to you is such that it has no reduce/reduce error - even though it has shift/reduce errors. I would recommend using the grammar in the same form as given in the question. Having said that, you can modify the grammar as long as it doesn't change the language generated by it. If you abide by the grammar given to you in the question, you'll see some shift/reduce warnings - but you can ignore those.
2. Question: *The assignment mentions that NUMBER can be a decimal. Does this mean we should allow users to input decimals as well?*
Answer: Yes. Decimal numbers are allowed as input. Please refer to the reference calculator program.

3. Question: *What kind of behavior is expected if a user inputs a variable that has not been declared yet?*
   Answer: As long as the syntax is correct but the variable(s) used are undefined, you have two options:
   i. print something like "ERROR: UNDEFINED VARIABLE"
   ii. print 0.00
   If there is a syntax error, you need to output: "ERROR: syntax error".
   If there is a symbol that is not present in the grammar, output: "ERROR: Undefined symbol"

4. Question: *Is the exponent operation should be left-associative?*
   Answer: If you use the grammar as given in the question - this operation would be left-associative by default - there is no associativity defined (yes the grammar is ambiguous). Hence the reference calculator behaves as it does. However, both implementations - left associativity as well as right associativity would be considered as the correct answer (as long as the associativity is consistent, i.e., if you define POW to be left-associative - it should always be left-associative and vice versa). Same goes for other mathematical operations as well (wherever applicable).

   You can always refer to the reference calculator for a more detailed behavior. Your calculator does not need to adhere to the outputs of the reference calculator as long as whatever you output makes sense.

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. $5 * 2 - 6 + 7 \; / \; 7$

```
We have:
5 * 2 - 6 + 7 / 7
5 * ((2 - (6 + 7)) / 7)
5 * ((2 - 13) / 7)
5 * (-11 / 7)
-55/7
```

2. $8 * 8 \; / \; 4 + 4 \; / \; 2 * 2$

```
We have:
8 * 8 / 4 + 4 / 2 * 2
8 * 8 / (4 + 4) / 2 * 2
8 * (8 / (8 / (2 * 2)))
8 * (8 / (8 / 4))
8 * (8 / 2)
8 * 4
32
```

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

```
We have:
5 - 3 * 5 % 3 + 2
(5 - 3) * (5 % (3 + 2))
2 * (5 % 5)
2 * 0
0
```

4. $10 - 2 + 6 * 10 - 2/6 + 4$

```
We have:
10 - 2 + 6 * 10 - 2 / 6 + 4
(10 - (2 + 6)) * ((10 - 2) / (6 + 4))
(10 - 8) * (8 / 10)
2 * (8 / 10)
```

```
2 * (4 / 5)
8 / 5
```

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

```
We have:
2 + 5 / 2 * 5 - 4 % 2
(2 + 5) / (2 * ((5 - 4) % 2))
7 / (2 * (1 % 2))
7 / (2 * 1)
7 / 2
```

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

```
We have:
12 - 4 * 9 - 4 / 10 / 6
(12 - 4) * ((9 - 4) / (10 / 6))
8 * (5 / (5 / 3))
8 * (15 / 5)
8 * 3
24
```

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.

      ```
      f: true
      g: false
      h: false
      i: false
      ```

   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 `||`.

      ```
      C++ - Section 8.15
      The || operator groups left-to-right. The operands are both contextually converted
      to bool (Clause 4). It
      returns true if either of its operands is true, and false otherwise. Unlike |, ||
      guarantees left-to-right evaluation; moreover, the second operand is not evaluated
      if the first operand evaluates to true.
      ```

3. Give a situation (i.e., provide an example) 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.

   Short Circuit Evaluation: This method is the one in which the second argument is evaluated only if the first argument does not determine the value of the expression.

   Strict Evaluation: It is an evaluation strategy in which the expression is evaluated as soon as it is bound to a variable.

   Example:

```
int a = 0;
if (a != 0 && myfunc(b))
{
    function();
}
```

   In this example, short circuit evaluation guarantees that myfunc(b) is never called. This is because a != 0 evaluates to false. Strict evaluation can make it complex.

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	main
	b	main
sub1	a	sub1
	b	main
sub2	a	sub2
	b	main
	c	sub2
sub3	a	sub2
	b	sub3
	c	sub2
	d	sub3