Assignment 4: Expression Evaluator for Rationals Using Trees

COL106: Data Structures and Algorithms, Semester-I 2023–2024

Submission Deadline: 30th September 5:00 PM

Honor Code

Following is the Honor Code for this assignment.

- Copying programming code in whole or in part from another or allowing your own code (in whole or in part) to be used by another in an assignment meant to be done individually is a serious offence.
- Using libraries apart from those included in the Startercode or explicitly permitted on forums such as Piazza is strictly forbidden.
- The use of publicly available codes on the internet is strictly not allowed for this assignment.
- The use of ChatGPT and other AI tools is strictly forbidden for this assignment. If any student has been found guilty, it will result into disciplinary action.
- Collaboration with any other person or team would be construed as academic misconduct.
- Outsourcing the assignment to another person or "service" is a very serious academic offence and could result in disciplinary action.
- Sharing of passwords and login ids is explicitly disallowed in this Institute and any instance of it is academic misconduct. Such sharing only compromises your own privacy and the security of our Dept./Institute network.
- Please ensure that your assignment directories and files are well-protected against copying by others. Deliberate or inadvertent copying of code will result in penalty for all parties who have "highly similar" code. Note that all the files of an assignment will be screened manually or by a program for similarity before being evaluated. In case such similarities are found, the origin as well as destination of the code will be held equally responsible (as the software cannot distinguish between the two).

Introduction

An *expression evaluator* is a software component or program that interprets and computes the result of mathematical, logical, or symbolic expressions provided as input. It takes an expression as input, processes it according to predefined rules or algorithms, and returns the result of the evaluation. Your task in this assignment is to create an expression evaluator for programs written in the language E++, for a special kind of input data which is described below. **Assignment 5 will build upon Assignment 4**. So it is highly encouraged to do this assignment carefully and well. The startercode for this assignment can be found here. You should download the startercode again if you have not been following the updates on Piazza.

For evaluation, some test cases will be made available to you to help you debug your program but all the "evaluation test cases" will not be made available to you. You need to ensure that your program works correctly **on all the inputs**, not just on the test case inputs.

1 Getting Started

1.1 Expressions

Expressions in E++ are **fully parenthesized**, well-formed arithmetic infix expressions, using the following operators: `+, -, *, /`. The operands appearing in an expression may be numerical, or variables (such as `x`).

1.2 Examples

1.2.1 Examples of valid E++ expressions:

- $((a + b))$
- $((x * y) - z)$
- $((5 + x)/(a + b))$
- $((a + 2) * 3)$
- $(((x + y) * (a - b)) - (c/d))$
- $(((2 * a)/3) + (b - (c - d)))$

1.2.2 Examples of invalid E++ expressions:

- $a + b$ (missing enclosing brackets)
- (ab) (missing operator)
- $((x + y) * (a + b) + c)$ (missing operators between sub-expressions)
- $()$ (empty expression)
- $(\&)$ (invalid operator)
- $(*ab)$ (not an infix expression)
- $((a/))$ (operator at the end)
- $((x + y)(a - b))$ (missing operator between sub-expressions)
- $(a + b))$(unbalanced parentheses)

1.3 Syntax

The syntax of E++ is very simple. There are two types of statements:

1. **Variable Assignment:** `v := E` where the LHS is a variable `v` and the RHS is a well formed expression `E` as defined above.
2. **Returning a Value:** `return E` where `E` is a well formed expression. (You will use this statement in Assignment 5. Programs you encounter in this assignment will not have this construct.)

Statements in E++ need to be separated with a newline. You need to also note the following semantics:

- **Declaration Before Use:** A variable x can appear on the RHS of an expression only if there exists a statement before it of the form `x := E`.
- **Immutability:** A variable x can only be assigned to once, i.e. there should not exist two distinct statements S_1 and S_2 such that S_1 : `x := E1` and S_2 : `x := E2`.

Note: You are not required to implement error detection or correction mechanisms for the provided E++ programs in the assignment. The programs you receive for testing purposes will be error-free and conform to the language's syntax and semantics.

1.4 UnlimitedInt

In this subsection, we will discuss the details of the `UnlimitedInt` class. This class will provide you with an opportunity to practice implementing fundamental integer operations and working with dynamic memory allocation. For some examples of languages using unlimited ints, see Appendix 7.

The `UnlimitedInt` class is a custom integer data type designed to handle integers of arbitrary size without any limitations on their magnitude. Unlike standard integer types in programming languages, such as `int` or `long`, which have fixed ranges, `UnlimitedInt` can represent extremely large or precise integer values. This class is a fundamental building block for working with integers in situations where traditional integer types fall short due to their size limitations.

You are provided with the header file for the `UnlimitedInt` class, which includes function prototypes, also described below. Your task is to implement these functions in the corresponding source file. Be sure to carefully manage dynamic memory allocation and deallocation.

1.4.1 Class Description

The `UnlimitedInt` class has the following attributes and methods:

- `size`: An integer that represents the size of the `UnlimitedInt` object.
- `capacity`: An integer that represents the capacity of the `UnlimitedInt` object.
- `sign`: An integer that indicates the sign of the `UnlimitedInt` object. It is set to 1 for positive numbers and -1 for negative numbers. For the UnlimitedInteger representing 0, you can set this to either 0 or 1 or -1
- `unlimited_int`: An integer pointer that points to an array storing the unlimited integer.

- `get_capacity()`: Returns the capacity of the `UnlimitedInt` object.
- `get_size()`: Returns the size of the `UnlimitedInt` object.
- `get_array()`: Returns a pointer to the array storing the `UnlimitedInt` digits.
- `get_sign()`: Returns the sign of the `UnlimitedInt` object (1 for positive, -1 for negative).
- `to_string()`: Convert a `UnlimitedInt` object to its string representation. For eg. `"5"` for `5` and `"-5"` for `-5`
- Arithmetic operations such as addition, subtraction, multiplication, division, modulus for `UnlimitedInt` objects.

1.5 Rationals with UnlimitedInt

In a related context, we introduce a class named `UnlimitedRational`, which will be used for evaluating expressions in one of the subparts of the assignment. A rational number is typically represented in the form p/q, where p and q are integers ($\mathbb{Z}$), p and q are coprime (meaning they have no common divisors other than 1), and $q \neq 0$.

The `UnlimitedRational` class extends the concept of rational numbers by using the previously introduced `UnlimitedInt` class for its numerator and denominator. This allows you to accommodate even the most extensive calculations in scenarios where standard floating-point representations might suffer from precision loss.

The `UnlimitedRational` class, as provided in the starter code, includes two attributes: `p` and `q`, each of type `UnlimitedInt`, representing the numerator and denominator of the rational number, respectively. It is important to note that for certain special cases, you may encounter situations where $q = 0$. These special cases will be discussed in more detail as you progress through the assignment.

You will be required to implement various arithmetic operations for `UnlimitedRational` objects, which will build upon the capabilities of the `UnlimitedInt` class to handle rational numbers of unlimited precision.

1.5.1 UnlimitedRational Class

The `UnlimitedRational` class includes the following attributes and methods:

- `p` and `q`: Pointers to `UnlimitedInt` objects representing the numerator and denominator, respectively. (Remember, p and q **must** be coprime, or your implementation will not pass our tests.) Note if `p` is 0, any non-zero value of `q` will be acceptable for the denominator.
- `get_p()` and `get_q()`: Methods to access the numerator and denominator as `UnlimitedInt` objects.
- `get_p_str()` and `get_q_str()`: Methods to retrieve the numerator and denominator as string representations.
- `get_frac_str()`: Method to return the rational number as a string in the form "p/q". If p/q is positive, then both "p/q" and "-p/-q" would be acceptable as answers. Similarly if p/q is negative, then the "-"sign can be either in the numerator or the denominator.
- Arithmetic operations such as addition, subtraction, multiplication, division for `UnlimitedRational` objects.

Note the following error conditions:

1. Incase of division by 0, the answer will be 0/0
2. Any operation with 0/0 as one of the operands must return 0/0 as the correct answer.

You will be tasked with implementing these operations for `UnlimitedRational` objects, leveraging the capabilities of the `UnlimitedInt` class for precise and unlimited-precision calculations. Appropriate header files have been given in the starter code for these.

2 Parser for E++ (30 Marks)

A parser converts each statement of the program into a tree structure (called a *Parse Tree*) which is expressive of all the components of that statement. It allows us to capture the structure of the program, as well as perform syntactic checks. Parse tree for the statement $x_2 := (2 + (7 * x_0))$ is given below

In this assignment, the parse tree for a given statement will always be unique. We do this by imposing the following conditions on the parse tree

- For a statement $v := E$, node associated with the variable v will be in left subtree of the node associated with := assignment operator, and all of the nodes in the tree of expression E will be in the right subtree of the node associated with := assignment operator
- Within an expression, all of the variables and constants needed to evaluate the left operand of the expression must be in the left subtree of operator node, all of the variables and constants needed to evaluate the right operand of the expression must be in the right subtree of operator node

Hence the parse tree below for the statement $x_2 := (2 + (7 * x_0))$ would be incorrect since for the + operand we require 2 to be in the left subtree of the operator node and 7 and x_0 to be in right subtree of the operator node.

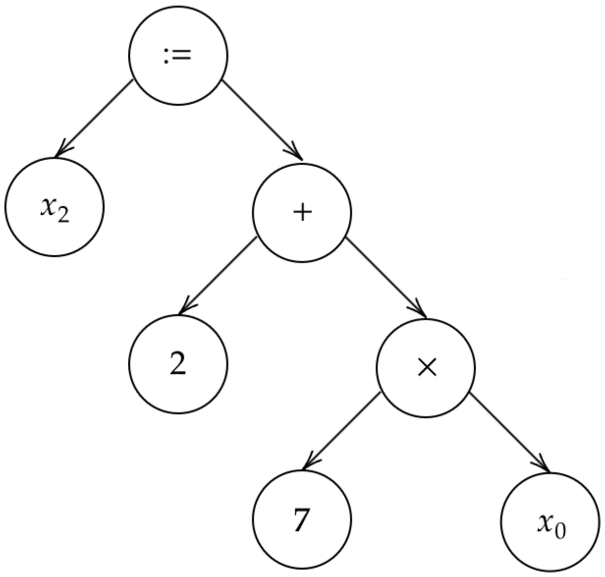

Figure 1: Example of the Parse Tree for the statement $x_2 := (2 + (7 * x_0))$

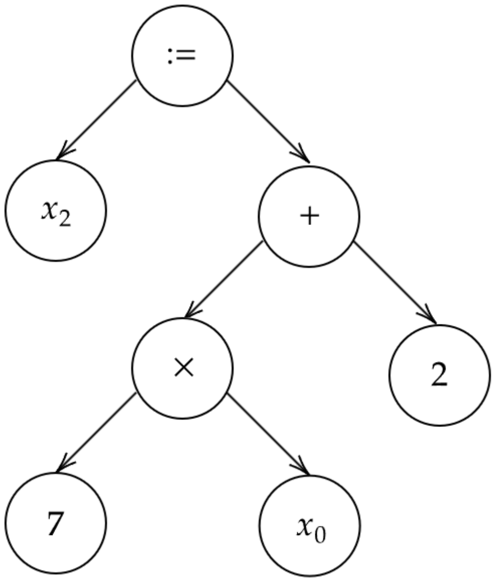

Figure 2: Example of an Incorrect Parse Tree for the statement $x_2 := (2 + (7 * x_0))$

3 Expression Evaluator (40 Marks)

Each well-formed expression evaluates to a value. Therefore, each node in the parse tree will have a value associated with it, which can be calculated explicitly by recursively evaluating the subtree rooted at that Node. **Note:** For an assignment `v := E`, we will not check the evaluated values at the nodes corresponding to `v` and `:=`. Feel free to assign a garbage value. Some more details are present in the following sections, particularly Section 5.2.

4 Symbol Table in Parsing and Expression Evaluation (30 Marks)

In this assignment, you will also be implementing a Symbol Table using a (unbalanced) Binary Search Tree (BST) to manage variables in the E++ expression evaluator. This task is essential for both parsing and expression evaluation. Let's explore the concept and its relevance:

4.1 Understanding Symbol Tables

A **symbol table** is a critical data structure used in our evaluator to store information about variables, functions, or other identifiers within an E++ program. It serves as a mapping, associating unique identifiers (in this case, variable names) with their corresponding values or attributes.

4.2 Relevance to Parsing

Symbol tables play a crucial role in syntax checking. They ensure that variables are declared before they are used, preventing undefined variable errors and other syntax-related issues.

However, in this assignment this would *not* be relevant, since the test cases ensure syntax and semantics are followed correctly.

4.3 Relevance to Expression Evaluation

For expression evaluation:

- **Variable Resolution**: In E++, expressions often include variables. To evaluate these expressions, you must resolve the variables to their actual values. The symbol table provides a mechanism to map variable names to their corresponding values.
- **Efficient Evaluation**: A symbol table enhances the efficiency of expression evaluation. Without a symbol table, you would need to traverse the entire code repeatedly to find variable values, leading to performance issues.

5 Implementation of Symbol Table, Parser, and Expression Evaluator

In this section, we describe the implementation for the Symbol Table, Parser, and Expression Evaluator for the programming language E++. We outline the expected functionality and relationships between these components.

5.1 Symbol Table

This needs to be implemented using a BST, not necessarily balanced. You should use the `<` operator (lexicographic ordering) for comparison amongst keys (strings).

5.1.1 Functions to Implement

You are required to implement the following functions in the `SymbolTable` class defined in `symtable.h`:

- `insert(string k, UnlimitedRational* v)`: This function should insert a key-value pair (variable name and its value) into the Symbol Table.
- `remove(string k)`: Implement a function to remove a key-value pair based on the variable name.
- `search(string k)`: Write a function to retrieve the value associated with a variable name from the Symbol Table.

- `get_size()`: Implement a function to get the size (number of entries) in the Symbol Table.
- `get_root()`: Provide a function to get the root node of the Symbol Table (useful for internal implementation).

5.2 Expression Tree Node

The `ExprTreeNode` class represents nodes in the expression parse tree. Each node has a `type`, which can be "ADD," "SUB," "MUL," "DIV," "VAL," or "VAR" and it may store a `val` (in case of a value node) and an `evaluated_value` (used during expression evaluation) and a `string_id` in case of a variable node. Note the root node of the parse tree must be the `":="` node and the type of this node can be set to any arbitrary value. **Also no nodes should be made for the brackets.**

5.3 Evaluator

This is the top-level class which will read individual instructions, parse them and evaluate them.

5.3.1 Functions to Implement

You are required to implement the following functions in the `Evaluator` class defined in `evaluator.h`:

- `void parse(vector<string> code)`: This function should read an input vector of tokens (strings), parse it, and convert it to a parse tree. The root of the tree should be pushed into the `expr_trees` vector. Ensure that the `type` and `val` properties are correctly set during parsing. An example is:
  Statement:
  `v := (13 + (2 / 51))`

  Tokens (Vector of strings):
  `["v", ":=", "(", "13", "+", "(", "2", "+", "(", "2", "/", "51", ")", ")"]`

  Note that the tokens corresponding to numerical values given as input to this function will always be **integral** (not rational). Only the intermediate and final values during calculations have to be rationals.
- `void eval()`: This function should evaluate the last element of the `expr_trees` vector. It will be called immediately after a call to `parse`, and it should also populate the symbol tables. Ensure that the `evaluated_value` property is correctly set during evaluating.

5.4 Usage Guidelines

You should use the Symbol Table to store and retrieve variable values during expression evaluation within the `evaluate_subtree()` function. Make sure to follow the conditions for parse tree construction outlined in the assignment instructions.

Your implementation should handle various expression types (e.g., addition, subtraction, multiplication and division) correctly.

Remember that this section outlines what needs to be implemented, but the specific implementation details and logic are up to you to develop. Your code should ensure correctness, efficiency, and adherence to the principles of parsing and expression evaluation.

6 Submission Instructions

Please read these instructions **very carefully**. You are required to submit the following **6 files** on Gradescope: `ulimitedint.cpp, ulimitedrational.cpp, exprtreenode.cpp, entry.cpp, symtable.cpp, evaluator.cpp`.

Do **not** submit the header files. We may replace them with our own files during evaluation testing.

We don't want you to stay up late at night as it is detrimental to your health, so please note the submission deadline is **5 PM**, and not midnight.

7 Points to Note

1. For each part, read the instructions carefully and use only the data structure specified.
2. Follow good memory management practices. We will stress test your submissions. There should be no memory leak, otherwise your code may fail on the evaluation test cases.

Appendix

Usage of Unlimited Integers in Other Languages

Several programming languages offer similar concepts for handling unlimited-precision integers:

- **Python**: Python's built-in `int` type can automatically switch to arbitrary precision when required. For example, you can calculate factorials of large numbers without worrying about integer overflow.
- **Golang**: Go has the `math/big` package, which offers the `Int` type for arbitrary-precision integers. Developers can use it for calculations involving large integers.
- **Java**: Java provides the `BigInteger` class in its standard library, which allows you to work with arbitrarily large integers. It offers methods for arithmetic operations and comparisons.

These languages employ similar concepts to `UnlimitedInt` to handle integers of unlimited precision, enabling a wide range of applications in scientific computing, cryptography, and more.