EGE UNIVERSITY

FACULTY OF ENGINEERING

DEPARTMENT OF COMPUTER ENGINEERING

PROGRAMMING LANGUAGES

PROJECT ASSIGNMENT

A blue and white logo

Description automatically generated

**WRITING AN INTERPRETER FOR THE PLUS++ LANGUAGE**

Prepared by

**Hedera Yağmur AŞKAR – 05230000386**

**Mert AKYILDIZ – 05220000358**

**Zeynep Nur ALANAT - 05230000374**

Course Instructor

**Dr. Öğr. Üyesi Ahmet EGESOY**

İZMİR, 2025

**Contents**

[1.1 Introduction 1](#_Toc200568487)

[1.2 Parser Design 1](#_Toc200568488)

[1.3 Virtual Machine Design 2](#_Toc200568489)

[2.1 Start 3](#_Toc200568490)

[2.2 Big Integer Implementation 4](#_Toc200568491)

[2.3 Variable Management 7](#_Toc200568492)

[2.4 Parser Implementation 8](#_Toc200568493)

[3.1 Main Method 17](#_Toc200568494)

[3.2 Output and Tests 18](#_Toc200568495)

## **1.1** Introduction

This project aims to build an interpreter for the Plus++ language, a simplified high-level programming language. The interpreter reads .ppp files, parses and validates the syntax and semantics, and executes the instructions if the program is correct.

## **1.2** Parser Design

**1.2.1 Purpose of the Parser**

The parser is responsible for analyzing the sequence of tokens produced by the lexer and validating that they follow the syntactic and semantic rules of the Plus++ language. If the input is valid, the parser generates an intermediate representation (not a full parse tree) in the form of an instruction list, which is then executed by the virtual machine.

**1.2.2 How the Parser Works**

The parser follows a recursive descent or manual FSM-based strategy, with a separate function for each type of valid statement in Plus++ language such as declarations, assignments, write commands, and repeat blocks. It begins by consuming tokens one at a time from the lexer using getNextToken(), where each token carries its type, lexeme, and source code location.

As it processes the input, the parser uses functions like parseDeclaration(), parseAssignment(), parseRepeatBlock(), and parseWriteStatement() to recognize and validate each construct. It employs a match(expected\_token) function to verify that the syntax is correct, and immediately reports an error if an unexpected token is found.

Instead of building a traditional syntax tree, the parser generates a linear instruction list, a form of intermediate representation, similar to bytecode. For example, the Plus++ code:

x = 10;

repeat 2 {

write x;

x += 1; }

is converted into the instruction sequence:

DECLARE x

ASSIGN x 10

REPEAT\_START 2

WRITE x

ADD x 1

REPEAT\_END

These instructions are stored in an array or linked list and passed to the virtual machine for execution.

The Plus++ parser is implemented as a predictive, LL(1) parser, which processes the input left-to-right with one-token lookahead, constructing a leftmost derivation. The grammar of Plus++ has been designed to be compatible with LL(1) constraints by avoiding ambiguity and left recursion, allowing clear and deterministic parsing decisions.

The parser uses a simple top-down approach where current\_token is used to determine which parsing function to call based on the token's type and content. For example:

If current\_token.type == TOKEN\_KEYWORD and the lexeme is "number", it calls parseDeclaration().

If current\_token.type == TOKEN\_IDENTIFIER, it performs one-token lookahead to determine if the statement is an assignment (:=), increment (+=), or decrement (-=), and dispatches accordingly.

The parser does not implement multi-error recovery; instead, it performs immediate termination on the first syntax or semantic error via the parseError() function. This simplifies the control flow and ensures the program halts on invalid input, which is acceptable for educational-level compilers or interpreters.

Overall, the parser is a clear example of an LL(1) predictive parser, using token-driven logic for statement dispatching and providing straightforward error diagnostics during parsing.

**1.2.3 Error Handling in the Parser**

Syntax errors are caught when expected tokens are missing or out of place such as using int = x;, which would raise: “Syntax Error at line 1: Expected identifier after 'int'.”

Semantic errors involve invalid usage of language rules, such as referencing undeclared variables or using non-integers in repeat loops. For instance, x = 3; without declaring x first will yield: “Semantic Error at line 1: Variable 'x' not declared.”

The parser stops as soon as it encounters a critical error and reports it clearly, specifying the error type, a meaningful description, and the exact line number where the issue occurred.

## **1.3** Virtual Machine Design

**1.3.1 Purpose of the Virtual Machine**

The virtual machine is responsible for executing the list of instructions generated by the parser. It maintains a symbol table to map variable names to their BigInt values, a program counter to track which instruction is currently being executed, and optionally a loop counter stack for managing nested repeat blocks.

**1.3.2 How the Virtual Machine Works**

The virtual machine processes instructions one by one in a sequential manner.

DECLARE x – Adds variable x to the symbol table.

ASSIGN x 10 – Sets the value of x to 10.

REPEAT\_START 2 – Begins a loop with 2 iterations.

WRITE x – Prints the current value of x.

ADD x 1 – Increments x by 1.

REPEAT\_END – Jumps back to the start of the loop if iterations are not complete.

The virtual machine uses a loop counter stack to manage nested repeat blocks. All arithmetic operations are handled using a custom BigInt type, allowing the interpreter to perform calculations on very large numbers without overflow.

**1.3.3 Error Handling in the Virtual Machine**

The virtual machine includes runtime error checks to handle issues such as: Use of undeclared or uninitialized variables, invalid repeat counts, division by zero. When an error occurs, the virtual machine halts execution and provides a clear message showing the instruction and the cause of the error. Example: “Error at instruction 4: Cannot repeat with undeclared variable 'i'.”

**Conclusion**

The parser ensures correct syntax and semantics and produces a clean instruction list. The virtual machine executes these instructions reliably, manages variable states, performs BigInt arithmetic, and catches runtime errors. Together, the parser and virtual machine provide a modular and robust foundation for interpreting Plus++ programs.

## **2.1 Start**

A black rectangle with white text

AI-generated content may be incorrect.

The BigInt struct is designed to store large integers that exceed normal integer limits. It holds the number’s digits as a character array, allowing for arbitrary length numbers, plus one extra space for the null terminator to mark the string’s end. The sign field stores whether the number is positive or negative, while the length field keeps track of how many digits the number currently contains.



The Parser struct encapsulates the parser’s current state. It holds a pointer to the script file being read, the current token extracted from the input (current\_token), and a flag (has\_error) indicating whether an error has occurred during parsing.

A black rectangle with white text

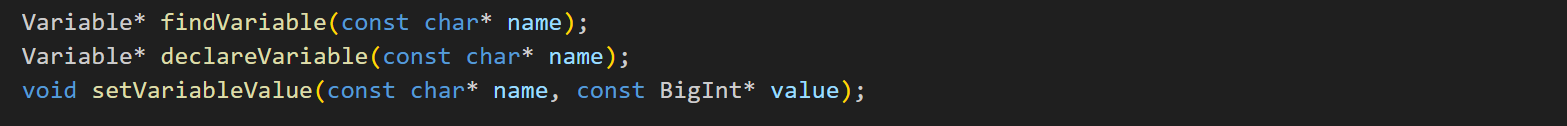
AI-generated content may be incorrect.

Variables in parser are stored as nodes in a singly linked list. Each Variable struct contains the variable’s name, which can hold up to 20 characters plus the null terminator, and the variable’s value represented by a BigInt. The next pointer links to the next variable in the list, enabling dynamic addition and lookup of variables. The global pointer variables points to the head of this linked list, which starts as NULL when no variables are declared yet.

A black screen with text

AI-generated content may be incorrect.

The BigInt operation functions manage initialization, setting values from strings, arithmetic operations like addition and subtraction, comparison, copying, and printing.



The variable management functions support searching for a variable by name, declaring a new variable, and updating a variable’s value.

A black screen with white text

AI-generated content may be incorrect.

The parser functions handle the core parsing tasks: advancing to the next token, reporting errors, parsing the whole program, and breaking down the input into statements and specific commands like declarations, assignments, increments, decrements, output writing, repeat loops, and parsing blocks of code. The function parseIntValue reads and converts integer literals into BigInt values.

## **2.2** Big Integer Implementation

A black rectangle with white text

AI-generated content may be incorrect.

This function initializes a BigInt instance to a default zero value. It fills the entire digits array with '0' characters, ensuring the string is null-terminated. The sign is set to positive, and the length is initialized to 1, representing the single digit zero. This sets a clean state for any new or reset big integer before use.

A computer screen shot of text

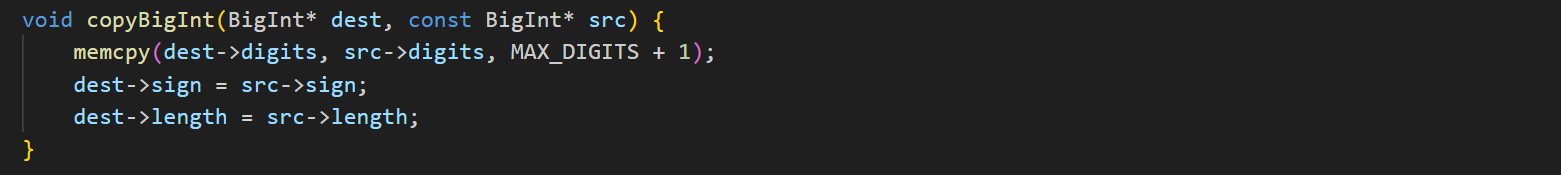
AI-generated content may be incorrect.

This function converts a normal string representation of a number into the BigInt format. First, it initializes the BigInt with zeros. It checks if the string starts with a minus sign to determine negativity. The function validates that the number of digits does not exceed the maximum allowed. It then stores digits in reverse order for easier arithmetic processing. Finally, it removes any leading zeros from the highest digits, adjusting the length.

A black screen with colorful text

AI-generated content may be incorrect.

This function prints a BigInt to standard output in human-readable form. It first prints a minus sign if the number is negative and not zero. Then, it prints the digits in reverse order to represent the number correctly. This allows users or the program to see the actual number stored inside the BigInt.



This function duplicates the content of one BigInt into another. It copies the digits array including the null terminator, copies the sign, and the length. This is useful for creating temporary copies during arithmetic or assignment operations without altering the original.

A computer screen with colorful text

AI-generated content may be incorrect.

This function compares two BigInt values and returns an integer representing their order. It first compares their signs: different signs immediately determine the result. If signs match, it compares the lengths longer numbers are greater if positive, smaller if negative. If lengths match, it compares digit by digit from the most significant to the least significant. The return value is positive, zero, or negative depending on whether the first number is greater, equal, or smaller than the second.

A computer screen shot of a black screen

AI-generated content may be incorrect.

This function adds two BigInt numbers and stores the result. If the signs differ, it calls subtraction with the negated second operand instead. Otherwise, it initializes the result, sets the sign, and performs digit-wise addition with carry handling, looping through the longer number’s digits. The result length is updated accordingly. This implements classical addition for large numbers digit-by-digit.

A screen shot of a computer

AI-generated content may be incorrect.

A computer screen shot of text

AI-generated content may be incorrect.

This function subtracts one BigInt from another. If their signs differ, it redirects to addition with an appropriately negated operand. It then determines which number is larger in absolute value to decide the sign and order of subtraction. It subtracts digit by digit, borrowing as necessary when the digit in the minuend is smaller than the subtrahend’s digit. After completing digit-wise subtraction, it trims any leading zeros for a proper minimal representation.

## **2.3 Variable Management**

A black background with blue and white text

AI-generated content may be incorrect.

This function searches through the linked list of variables to locate a variable by its name. It starts from the global variables pointer, which points to the head of the list, and traverses each node one by one, comparing the target name with the stored variable names using strcmp. If it finds a match, it immediately returns a pointer to that variable. If it reaches the end of the list without finding the variable, it returns NULL, indicating the variable is not declared yet.

A computer screen shot of a black screen

AI-generated content may be incorrect.

This function is responsible for adding a new variable to the list. Before creating a new variable, it calls findVariable to check if the variable name is already declared; if it is, the function returns NULL to avoid duplicate declarations. If the name is new, it allocates memory for a new Variable struct, copies the given name into the struct, initializes its value to zero by calling setBigInt, and then inserts this new variable at the front of the linked list by adjusting the variables head pointer.

A black screen with white text

AI-generated content may be incorrect.

This function updates the value of an existing variable. It first searches for the variable using findVariable. If the variable exists, it copies the new BigInt value into the variable’s value field using the copyBigInt function.

## **2.4** Parser Implementation

A screen shot of a computer code

AI-generated content may be incorrect.

This function advances the parser by reading the next token from the input source using getNextToken. If this token indicates an error, it calls parseError to report the problem and stop execution. The parseError function prints a clear error message with the line number where the issue occurred, sets an error flag within the parser state, and terminates the program. These two functions form the core mechanism for controlled token consumption and immediate error handling, ensuring the parser does not continue when it encounters invalid input.

A screen shot of a computer code

AI-generated content may be incorrect.

This function parses an integer value that can either be a literal constant or a variable identifier. If the current token is an integer constant, it converts the token's lexeme to a BigInt and advances to the next token. If the token is an identifier, it attempts to find the corresponding variable in the symbol table. If the variable is not found, it reports an error. Otherwise, it copies the variable's stored BigInt value. If the token is neither an integer constant nor a valid variable, the function raises a parsing error.

A screenshot of a computer program

AI-generated content may be incorrect.

This part handles the parsing of variable declarations, which start with the keyword number. It verifies that the current token matches the number keyword, then expects an identifier token representing the variable name. The parser then attempts to declare this variable; if it already exists, an error is raised to avoid redeclaration. Finally, it expects a semicolon token to properly terminate the declaration statement. Each step carefully advances the token stream and reports errors when expectations are not met, ensuring declarations follow the correct syntax.

A screen shot of a computer code

AI-generated content may be incorrect.A screen shot of a computer program

AI-generated content may be incorrect.A screen shot of a computer program

AI-generated content may be incorrect.

This section implements parsing for variable assignment and arithmetic modification statements. The parseAssignment function expects a variable identifier followed by the assignment operator “:=”. It validates that the variable has been declared, then reads the integer value to be assigned and updates the variable's value accordingly. It also enforces a semicolon at the end. The increment and decrement functions follow a similar pattern but look for the operators “+=” and “-=”, respectively, and update the variable by adding or subtracting the parsed value.

A screen shot of a computer code

AI-generated content may be incorrect.

A computer screen with text on it

AI-generated content may be incorrect.

This function parses write statements used to output strings, numeric values, variable contents, or newline characters. After ensuring the write keyword is present, it enters a loop that can print multiple items separated by the keyword and. It supports printing string literals, integer constants, variable values, and the special keyword newline to output a line break. Each print item advances the token, and the loop ends when no further and keyword follows. The statement must be terminated by a semicolon.

A screen shot of a computer code

AI-generated content may be incorrect.

A computer screen with text on it

AI-generated content may be incorrect.

A screen shot of a computer code

AI-generated content may be incorrect.

A computer screen with many white text

AI-generated content may be incorrect.

A screen shot of a computer program

AI-generated content may be incorrect.

This function parses repeat loops with syntax such as repeat 10 times { ... } or repeat variable times { ... }. It first confirms the presence of the repeat keyword. Then, the loop count can be given either as a numeric constant or as a variable name whose value is used as the count. The function validates that the loop count is non-negative and not excessively large. After parsing the count, it expects the keyword times. The actual loop body parsing and execution logic (including file position rewinding to re-parse the block multiple times) are not included here to focus on loop count parsing.

A screen shot of a computer program

AI-generated content may be incorrect.

A screen shot of a computer code

AI-generated content may be incorrect.

A screen shot of a computer code

AI-generated content may be incorrect.

The first function is responsible for parsing a block of code that must start with a '{' and end with a '}'. Initially, it checks whether the current token is '{'; if not, it raises a parsing error. After confirming the opening of the block, it advances to the next token and performs an additional check to detect empty blocks. If the very next token is the closing '}', it reports a parsing error stating that empty blocks are not allowed. This ensures that every block contains at least one valid statement. Following this validation, the function enters a loop where it repeatedly calls parseStatement for each statement inside the block, continuing until it encounters the closing '}' or reaches the end of the file. Once the loop ends, it verifies that the current token is indeed the closing brace and, if so, consumes it by advancing the token; otherwise, it reports another parsing error.

The second function acts as a dispatcher that identifies and handles various types of statements. If the current token is a keyword, it checks its specific type and invokes the appropriate parsing function: parseDeclaration for "number", parseWrite for "write", and parseRepeat for "repeat". If the statement starts with an identifier, the function uses a lookahead technique: it saves the current token and file position, then reads the next token to determine the type of operation. Depending on whether the operator is ":=", "+=", or "-=", it resets the file pointer and token state, and calls the respective parsing function—parseAssignment, parseIncrement, or parseDecrement.

A black background with text

AI-generated content may be incorrect.

This is the main driver function for parsing the entire program. It initializes parsing by reading the first token, then enters a loop that continues until the end of the file or an error occurs. Each iteration calls parseStatement to interpret the next statement. By looping through the entire token stream, this function ensures that all input is parsed according to the grammar rules defined in other functions. It serves as the entry point for the recursive descent parser.

## **3.1** Main Method

A computer screen with text on it

AI-generated content may be incorrect.

The main function begins by determining how the user intends to provide the input file. If no filename is passed as an argument, the program prompts the user to enter a filename manually, excluding the .ppp extension. Once the filename is received, it ensures that the .ppp extension is appended if it was not included, standardizing the file format expected by the interpreter.

A computer screen shot of a black background

AI-generated content may be incorrect.

The function attempts to open the specified file for reading. If the file cannot be found or opened, an error message is displayed, and the program exits with a non-zero return code to indicate failure. Upon successful file access, the program initializes the Parser structure by assigning the opened file pointer and setting the has\_error flag to 0, which serves as an indicator for whether any errors occur during parsing.

A black screen with colorful letters

AI-generated content may be incorrect.

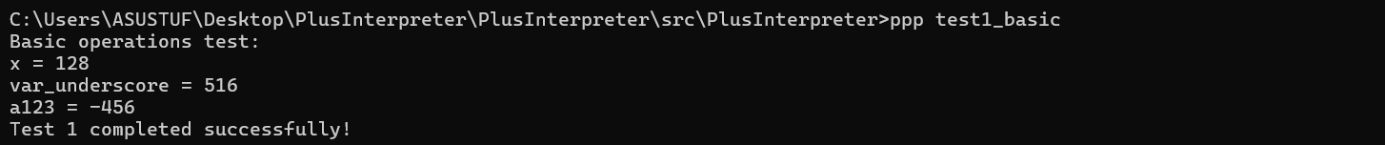
The core logic is triggered through the parseProgram(&parser) call, which begins parsing and executing the contents of the script file. After this call completes, the program checks the has\_error flag. If no parsing errors occurred, it prints a success message indicating that the script executed correctly.

A black rectangle with white text

AI-generated content may be incorrect.

It closes the script file and deallocates memory used for variables created during execution. This is done using a loop that iteratively frees each node in the linked list of variables. The program then returns 0 if no error occurred, or 1 if there was a parsing or execution issue, allowing external scripts or systems to detect whether the interpreter ran successfully.

## **3.2** Output and Tests



This test verifies basic functionality such as variable declaration and assignment. It demonstrates that the interpreter correctly handles variable names with letters, numbers, and underscores, and accurately assigns both positive and negative integer values. The successful output confirms proper execution of fundamental language features.

metin, ekran görüntüsü içeren bir resim

Yapay zeka tarafından oluşturulmuş içerik yanlış olabilir.

This test evaluates various loop functionalities in the interpreter. It includes constant-count loops, variable-based countdowns, nested loops, zero-iteration loops, single-iteration tests, and loops with arithmetic operations during execution. The correct sequencing and expected outputs confirm that the interpreter handles loop logic, control flow, and arithmetic updates inside loops accurately.

metin, ekran görüntüsü, yazı tipi içeren bir resim

Yapay zeka tarafından oluşturulmuş içerik yanlış olabilir.

This test focuses on the BigInt module by performing arithmetic operations on very large numbers, including values with up to 98 digits. It verifies the correctness of addition, subtraction, and combined operations with both positive and negative BigInts. The accurate results confirm that the interpreter handles arbitrary-precision arithmetic reliably, even with extremely large values.

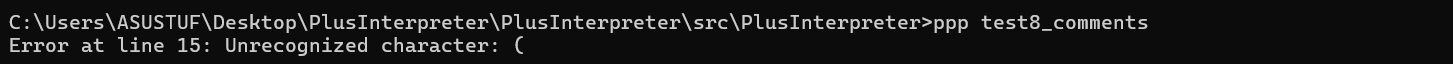
metin, ekran görüntüsü, yazı tipi, siyah içeren bir resim

Yapay zeka tarafından oluşturulmuş içerik yanlış olabilir.

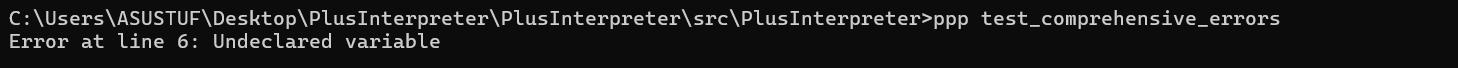
This test evaluates the interpreter’s ability to process various string outputs, including empty strings, strings with spaces, special characters, numeric values, and multi-line text. It also confirms support for long and complex string constants. The successful execution demonstrates that the Plus++ language correctly handles diverse string formats without issues.



This test checks the parser's ability to catch syntactic errors related to empty code blocks. The interpreter correctly identifies the issue and reports an error at the expected line, confirming that it enforces the language rule that disallows empty blocks.



This test ensures that the lexer correctly identifies and rejects unsupported characters in the source code. In this case, the interpreter reports an error upon encountering an unrecognized symbol '(', confirming that lexical analysis accurately filters invalid input according to the Plus++ language specification.



This test validates the interpreter's semantic analysis by attempting to use a variable without declaring it first. The interpreter correctly identifies the issue and reports an error at the appropriate line, confirming that variable declaration is strictly enforced before usage.

metin, ekran görüntüsü içeren bir resim

Yapay zeka tarafından oluşturulmuş içerik yanlış olabilir.

metin, ekran görüntüsü, yazı tipi içeren bir resim

Yapay zeka tarafından oluşturulmuş içerik yanlış olabilir.

This continuation includes edge case evaluations such as zero-iteration loops and complex mixed output with strings and large integers. The final summary confirms that all Plus++ interpreter features—ranging from variable management and arithmetic to loops, strings, and comments—have been thoroughly tested and validated. The message “PLUS++ INTERPRETER VALIDATION COMPLETE” indicates successful end-to-end functionality.

All components worked as intended. The interpreter passed all test cases including edge and error scenarios.