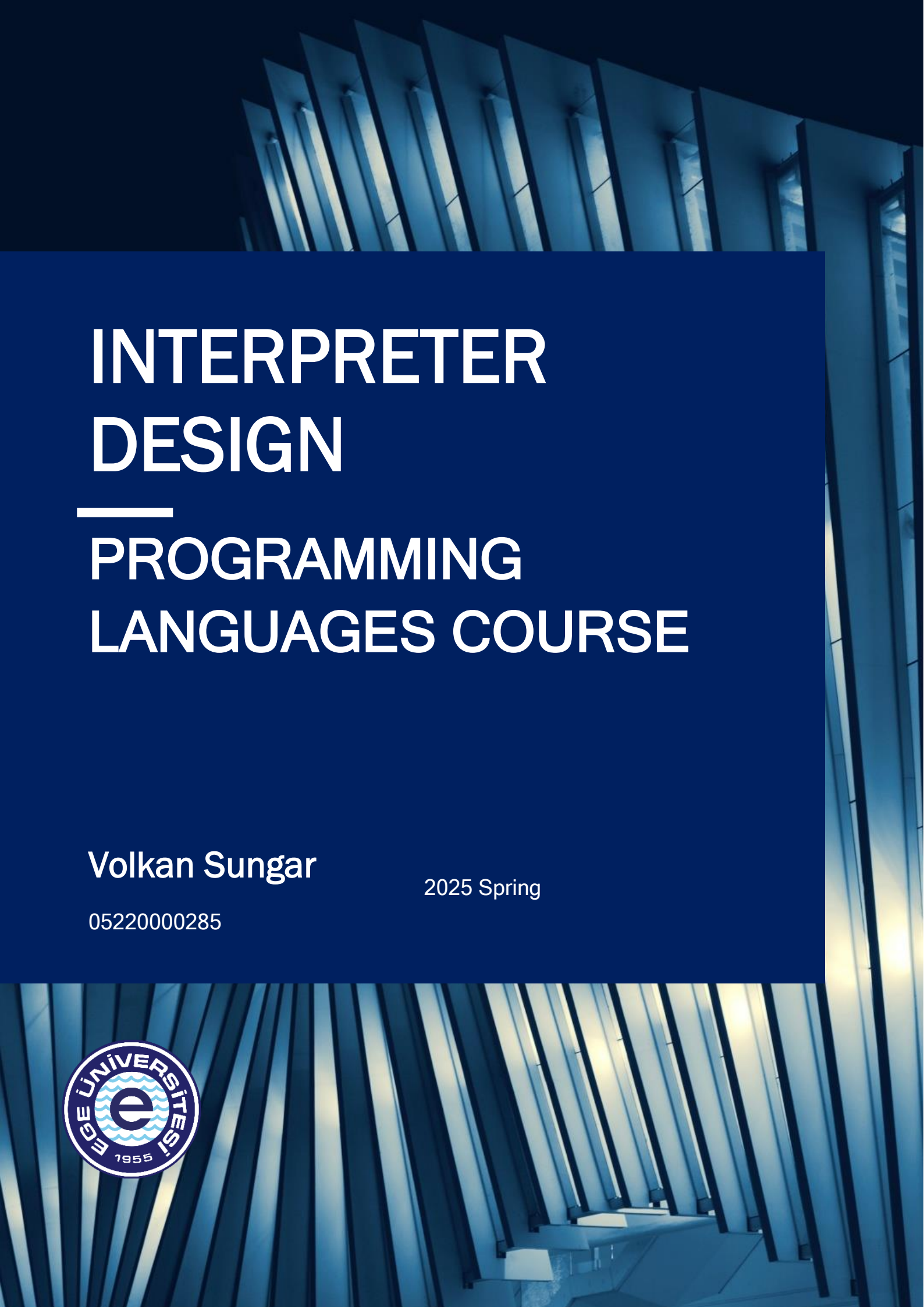

INTERPRETER DESIGN

PROGRAMMING LANGUAGES COURSE

Volkan Sungar

05220000285

2025 Spring

INTERPRETER DESIGN IN C

Introduction

This is a report for implementation of an intepreter for a basic programming language called Plus++ using C language. Previously, the lexical analyzer design and report were submitted, therefore this report will mostly include the parser and code generator design. Unlike the previous report, this one will be more in depth in order to explain the complicated algorithms of the code. Since this is not an automata theory book, the logic behind the common data structures of the intepreter will not be explained as in depth.

Architecture

As the classical approaches suggest, this design implements a three-step architecture. First, the lexical analyzer will generate the tokens and pass it to the parser. Then, the parser will generate the syntax tree. Finally, the code generation will generate the machine code accordingly to the syntax tree.

Plus++ is a language that has been designed in order to create big integers, increment and decrement them at will and display the result on the screen. In Plus++, a number can be as big as 100 decimal digits. Since C code cannot work with 100 decimal digits, we must use arrays to store the data structure and define new operator functions to implement the number data structure.

PARSER

TECHNOLOGY

The parser implements a **LALR(1)** design. LALR stands for **Look-ahead, left-to-right, rightmost derivation.** LALR is a kind of bottom-up approach to the token string, which means that the program will try to reach the start production in the process of creating the syntax tree. The "(1)" denotes one-token lookahead, to resolve differences between rule patterns during parsing.

LALR(1) is among the most powerful parsers. Bottom-up parsers usually perform better than top-down parsers because top-down parsers often employ backtracking, a computationally expensive process. Bottom-up

parsers can also handle complex grammar, which LALR(1) is one of the best even among the other bottom-up parsers.

The balance between performance and strength was the top priority in this project. Among all parser implementations, LALR(1) being the preferred choice was for this reason, even though a CLR(1) is stronger. Memory management also had to be robust. Each step was tested for memory leak.

The parser was also designed to work with further development. Currently the basic Plus++ language does not cause any conflicts, such as shift-reduce or reduce-reduce conflicts. If the user wanted to develop their language further, a simple recursive logic function most likely would fail, as even the simplest grammars such as if-else blocks would cause conflicts. With this strong design, the parser easily handles these conflicts.

STRUCTS

In a complex parser design like this, we must talk about the structs in use.

GrammarSymbol

A Symbol is either a terminal or a non-terminal from the Backus-Naur Form. Three fields for its name, SymbolType and id.

Production

A Production is the most fundamental working method of a parser. This struct defines the productions of Backus-Naur Form. All the products combined define the complete grammar of the language. Fields are left-hand side, right-hand side and a pointer to the semantic action function(for code execution). This production creation happens in the main.c file manually after populating the map with all symbols. (main.c line 207)

LR(1) ITEMS

LR(1) items can be thought of as a data structure that implements the core thinking and look-ahead of the LALR(1) parser.
[A→α·β, a] is an example of a LR(1) item definition.
The dot symbol denotes that every symbol before itself(α) was pushed into the stack therefore is processed and known by the parser. Symbols after it(β) are the ones that the parser expects after the already known symbols according to the production rule which is [A→α β] **a** is a **lookahead terminal**. This is a single terminal symbol from the input stream that is *expected to*

follow the non-terminal A if this production is eventually reduced. This lookahead is crucial for resolving conflicts.

DATA STRUCTURES

To generate a syntax tree, the parser must utilize many other data structures.

RULES TABLE

First, we must let the parser know the rules of our grammar. This can easily be achieved by inserting the rules to an array. Let's call this array '*productions*'. The productions array will sort the rules by their id, which will be used to access them by being the index. Each production will also have fields like left-hand-side and right-hand-side to store the symbols.

FIRST SET (parser.c line 523)

After the production rules are known, the parser can build the **FIRST SET**. This is a special type of array that helps parser to know the possible first terminals of symbols or production rules of the grammar. This information is vital for calculating the **lookahead symbols** in LR(1) items, which are then used to build the LALR(1) parsing table. Essentially, FIRST sets empower the parser to make correct **shift-reduce** and **reduce-reduce decisions**, guiding it to successfully parse input by looking ahead one symbol to resolve potential ambiguities in the grammar.

Rules for computing FIRST(X):

1. **If X is a terminal:** FIRST(X) = {X}. (A terminal symbol starts with itself).
    - Example: FIRST(id) = {id}
    - Example: FIRST(+) = {+}
2. **If X is a non-terminal:** Iterate through each production X -> Y1 Y2 ... Yk:
    - Add FIRST(Y1) to FIRST(X).
    - **If Y1 can derive ϵ (i.e., ϵ∈FIRST(Y1)):** Then add FIRST(Y2) (excluding ϵ) to FIRST(X).
    - **Continue this process:** If all Y1 through Yi can derive ϵ, then add FIRST(Yi+1) (excluding ϵ) to FIRST(X).
    - **If all Y1, Y2, ..., Yk can derive ϵ:** Then add ϵ to FIRST(X).

The FIRST set of a symbol is the set of terminals that can appear at the beginning of strings derived from that symbol.

FOLLOW SET (parser.c line 601)

The FOLLOW set of a non-terminal A is the set of terminals that can appear immediately after A in some derivation.

For LALR(1) parsers, **FOLLOW sets** are primarily used during the **SLR(1) phase** of constructing the LR parsing table. Specifically, when an SLR(1) parser is in a state where a reduction for a production $A \rightarrow \alpha$ is possible, it consults FOLLOW(A) to determine if the current lookahead symbol permits this reduction. If the lookahead is in FOLLOW(A), the reduction is performed; otherwise, it might be a shift or a different reduction. While LALR(1) merges states from LR(1), the underlying principles of how lookaheads (which sometimes involve FOLLOW sets, especially for nullable productions) are derived from the LR(1) canonical collection are still relevant, ensuring correct shift/reduce decisions.

Algorithm:

1. Initialize FOLLOW sets as empty
2. Add $ (end marker) to FOLLOW(start_symbol)
3. Repeat until no changes occur:
    - For each production $A \rightarrow \alpha B \beta$:
        - Add FIRST(β) - {ε} to FOLLOW(B)
        - If $\varepsilon \in$ FIRST(β) or β is empty, add FOLLOW(A) to FOLLOW(B)

ACTION TABLE

ACTION table defines the action that the parser needs to take according to current state and next terminal symbol(lookahead token)

There are 4 ACTION's: **Shift, Reduce, Accept, Error;**

Shift: Push new state and symbol onto stack, advance input

Reduce: Pop symbols according to production, create AST node, push result

Accept: Parsing successful, return AST root

Error: Handle syntax error

GOTO TABLE

GOTO table tells the parser the state it needs to go after executing a reduce process. When the parser recognizes a complete grammar production (a non-terminal), the goto table dictates the next state to transition to, based on the current state and the non-terminal just recognized. Essentially, it directs the parser on how to correctly build the parse tree by pushing the new non-terminal onto the stack and moving to an appropriate subsequent state.

1. Given an item set I and symbol X

2. Let J = {A → αX·β | A → α·Xβ ∈ I}
3. Return CLOSURE(J)

AST Node

To represent an Abstract Syntax Tree in code, we must implement a tree structure with node objects pointing at each other. AST Nodes will have type, location of their token, Node pointers for their children as their fields.

State

An LALR(1) state is a collection of **LR(1) items,** indicating all possible parsing situations the parser might be in at a given point in the input stream. An ItemSet directly corresponds to an LALR(1) state in the code.

Stack

The stack serves as a memory for the parser's current context, enabling it to make informed shift/reduce decisions based on what has been processed so far and what is coming next in the input stream. The stack primarily holds a sequence of "states" from the parser's finite automaton. Each state represents a summary of the portion of the input that has been processed so far and what the parser expects to see next. These states are generated during the LALR(1) table construction.

FUNCTIONS

CLOSURE

Closure is one of the fundamental functions of an LALR(1) parser. We use closure function to build a canonical collection of LR(1) items. This means to take a set of LR(1) items and expand it by adding all new items that can be derived from the current set. It receives a set of LR(1) items, and since a state is a set of LR(1) items, the closure will receive a state. During LR(1) set creation, the closure will receive the augmented grammar's first state:
S' -> . Program $

Closure Algorithm:

1. Start with the given set of items
2. For each item A → α·Bβ where B is a non-terminal:
    - For each production B → γ:
        - Add item B → ·γ to the closure (if not already present)

SEMANTIC ACTION

Abstract Syntax Tree is constructed using the semantic action functions. The main goal of these functions is to take the AST nodes (or data) associated

with the symbols on the right-hand side (RHS) of a production rule and combine them to create a new AST node representing the left-hand side (LHS) non-terminal. This process incrementally builds the parse tree into a more abstract and useful AST.
Meaning Attribution: Beyond just building the tree structure, semantic actions are where "meaning" is attributed to the parsed constructs. For example, when an INTEGER token is parsed, its integer value is stored in the AST node. When an IDENTIFIER is parsed, its name and potentially a symbol table index are stored.

Parse (parser.c line 1136)

This function is the heart of a parser. It will receive the grammar pointer and the token string as parameters. The first thing it does is to initialize the stack and AST. Then in a while loop, decides for the current state by consulting the stack. (start state is 0). After that, it consults the action table by current state and current token type to what action it should take. Depending on the action type(switch-case mechanism) it performs one of the four actions. If it tries to reduce, it must consult the goto state.

Helper Functions

Parser uses various helper functions to build, modify, and check the previous data structures. Notable ones include:
AST node functions; create function will set all the fields to default, add child function will set a parent node a child node by modifying its struct field, create leaf from token function will use a switch-case to create the leaf nodes by calling the create function and passing the token names to them.

CODE FLOW

After the lexer creates token string, GrammarSymbol's are initialised. All symbols are used to populate an array(map) according to their id's. Productions are initialised using previously created symbols. Grammar array is created using the Productions. FIRST, FOLLOW and LR(1) sets are created according to the predefined grammar. **Note:** This is usually done once a language is defined and stored forever. For each code we use the previously created sets and tables. But in this project, the tables generate each time Plus++ code is received. This is extremely inefficient for real life applications but was used here for the sake of debugging and simplicity.

After the FIRST and FOLLOW sets are created, we build the parsing tables. Then start the parsing process which returns a pointed to the AST root. This uses the **parse()** function.

CODE EXECUTION

STRUCTS

BigInt

Even though represented by an individual source file in the original project, it is a part of the code execution step. BigInt is a struct that implements the 100-digit numbers of the Plus++ language. Since C language cannot use 100-digit numbers, it is a must to define a new struct and functions to perform actions such as addition and subtraction. There are no multiplication or division as the current state of the project, so they are not implemented for now.
BigInt uses an array to hold the digits, each element being a limb. To calculate the addition, it calculates each limb then passes the carry to the next one.

CODE FLOW

interpreter.c uses interpret functions to execute code. interpret_program() function is called in the main.c file after AST is created. Each function acts like the left-hand-side of a production and will call the right-hand-side functions. Program will call statement list and statement list call all the statements it represents. interpret_statement(function) will call the statement's type(declaration, assignment etc.) function depending on the type of the AST node. These functions represent the actual working of the code written in C. For example, **interpret_declaration()** will set the symbol's value to 0 in global_runtime_symbol_table.
interpret_write_statement() will get the contents of the output list as an individual ASTNode. Then according to their type(string or number) it will call printf (bigint helpers used for numbers)
interpret_loop_statement() will check if the loop count is zero or negative. Then will call interpret_code_block or interpret_statement functions according to its body node. Each iteration decrements the counter by one but it returns

to original value after the loop ends to make nested loops possible. The counter is also type of bigint like all the other integers in Plus++
interpret_decrement() will first consult the global_runtime_sym_table if the variable is declared by calling a lookup function. Then will call bigint's subtraction function and update the symbol in the table accordingly to the new value. This action is similar for increment.

TESTING AND ERROR REPORT

For easier testing of the output, the comprehensive console prints are commented out. The tester can choose to enable these if they like. Consult Appendix1.

1. LEXER TEST CASES

Lexer testing will be present since the lexer code has changed slightly.

1.1 Token Recognition Tests

1.1.1 Keywords

```
number repeat times write and newline
```

Output: NUMBER, REPEAT, TIMES, WRITE, AND, NEWLINE

1.1.2 Operators and Punctuations

```
:= += -= ; { } ( )
```

Output: ASSIGN, INCREMENT, DECREMENT, SEMICOLON, LBRACE, RBRACE, LPAREN, RPAREN

1.1.3 Valid Identifiers

```
myVar underscore_ var123 a A longVariableName1
```

Output: All should be IDENTIFIER tokens

1.1.4 Invalid Identifiers

```
123invalid _123startUnderscore
thisVariableNameIsWayTooLongToBeValidInPlusPlusLanguage
```

Output: (INTEGER, IDENTIFIER), ERROR, ERROR Identifier name too long.

1.2 Number Literal Tests

1.2.1 Valid Numbers

```
0 123 -456 -0
12345678901234567890123456789012345678901234567890123456789012345678901234567890123456789012345678901234567890
```

Output: All should be INTEGER tokens

1.2.2 Invalid Numbers

```
3.14 +5 --5 - 5 123.456 1e10
```

Output: ERROR for each: ERROR + Line:1 Col:5

1.2.3 Edge Case Numbers

```
-
12345678901234567890123456789012345678901234567890123456789012345678901234567890
12345678901234567890123456789012345678901234567890123456789012345678901234567890
```

Output: Integer Literal too long ERROR

1.3 String Literal Tests

1.3.1 Valid Strings

```
"Hello World" "123" "" "Special chars: !@#$%^&*()" "String with spaces "
"String    with       tabs"
```

Output: All recognized as StringConstant

1.3.2 Invalid Strings

```
"Unterminated string
'Single quotes should not work'
"String with "embedded" quotes"
```

Output: Unterminated string literal at EOF, ERROR,
(STRING, INDENTIFIER, STRING)

1.4 Comment Tests

1.4.1 Valid Comments

```
*single line comment*
number x; *comment after code*
*multi
line
comment*
number y;
```

Output: NUMBER, INDENTIFIER, EOL, NUMBER, IDENTIFIER, EOL

1.4.2 Invalid Comments

```
*unterminated comment
number x;
```

Output: ERROR, Unterminated comment block.

1.5 Whitespace and Line Handling

```
number    x   ;


   number      y    ;
number z; number a;
```

Output: Whitespaces were handled gracefully.

2. PARSER TEST CASES

2.1 Declaration Tests

2.1.1 Valid Declarations

```
number x;
number myVariable;
number underscore_;
number var123;
```

Output:

Program

StatementList

Declaration	Declaration	Declaration	Declaration
Identifier: x	Identifier: myVariable	Identifier: underscore_	Identifier:var123

2.1.2 Invalid Declarations

```
number; number 123invalid; int x; number x number y;
```

Output: Parser Error: No valid action for state 11 on token EndOfLine (';') at line 1, column 6.

2.1.2 Duplicate Declarations

```
number x; number x;
```

Output: `[DEBUG] Declared variable 'x' with initial value 0.`
`Runtime Error: Variable 'x' already declared at line 2, column 6.`

2.2 Assignment Statement Tests

2.2.1 Valid Assignments

```
number x;
```

```
number y;
x := 5;
y := x;
x := -123;
y := 0;
```

Output: Valid AST nodes.

2.2.2 Invalid Assignments

```
number x;
x := ;        // Missing value
:= 5;         // Missing variable
```

Output: `State: 10, Current Token: AssignmentOp (':=', Line:2 Col:1) | Action: SHIFT 21`
`Parser Error: No valid action for state 21 on token EndOfLine (';') at line 2, column 4.`
`State: 5, Current Token: EndOfLine (';', Line:1 Col:8) | Action: SHIFT 17`
`Parser Error: No valid action for state 17 on token AssignmentOp (':=') at line 2, column 0.`

2.3 Increment/Decrement Tests

2.3.1 Valid Increment/Decrement

```
number x;
number y;
x += 5;
y -= 3;
x += y;
y -= x;
```

Output: Valid AST nodes.

2.3.2 Invalid Increment/Decrement

```
x += ; // Missing value += 5; // Missing variable
```

2.4 Write Statement Tests

2.4.1 Valid Write Statements

```
number x;
x := 42;
write "Hello World";
write x;
write "Value is:" and x;
write "Line 1" and newline and "Line 2";
write x and "plus" and x and "equals" and x;
```

Output: Valid AST nodes.

2.4.2 Invalid Write Statements

```
number x;
write;
write "Hello" and;
write and "Hello";
write "Hello" "World";  * Missing 'and'*
write undeclared;
```

2.5 Loop Statement Tests

2.5.1 Valid Simple Loops

```
number x;
x := 5;
repeat 3 times write "Hello";
repeat x times write "World";
repeat 0 times write "Nothing";
```

2.5.2 Valid Block Loops

```
number x;
number sum;
x := 5;
repeat x times {
    write x and newline;
    sum += x;
}
```

2.5.3 Nested Loops

```
number i;
number j;
i := 3;
j := 2;
repeat i times {
    repeat j times {
        write i and "," and j and newline;
    }
}
```

Output:
```
3,2
3,1
2,2
2,1
1,2
1,1
```

2.5.4 Invalid Loops

```
number x;
repeat times write "Hello";        * Missing count*
repeat 3 write "Hello";            * Missing 'times'*
repeat x times;                    * Missing statement*
```

Output: `Parser Error: No valid action for state 13 on token KEYWORD_TIMES ('times') at line 2, column 6.`
`Parser Error: No valid action for state 33 on token KEYWORD_WRITE ('write') at line 2, column 8.`
`Parser Error: No valid action for state 41 on token EndOfLine (';') at line 2, column 14.`

2.6 Code Block Tests

2.5.3 Invalid Code Blocks

```
number x;
repeat 3 times {
    x += 1;
    write x;

repeat 3 times }
    x += 1;
{
```

Output: `Parser Error: No valid action for state 61 on token EndOfFile ('') at line 5, column 0`
`Parser Error: No valid action for state 41 on token CloseBlock ('}') at line 1, column 14.`

3. SEMANTIC ANALYSIS TEST CASES

3.1 Variable Declaration and Usage

3.1.1 Use Before Declaration / Undeclared Variable Use

```
x := 5;
number x;
```

Output: Parsing is successful.
`Runtime Error: Undeclared variable 'x' in assignment at line 1, column 0.`

3.2 Type Checking

3.2.1 Assignment Type Mismatch

```
number x;
x := "string";
```

Output: `No valid action for state 21 on token StringConstant ('"string"') at line 2, column 4.`

3.3 Loop Variable Modification

```
number counter;
counter := 5;
repeat counter times {
    write counter and newline;
    counter -= 1;   *This should be allowed and affect loop*
}
write "Final counter: " and counter;
```

Output: 5, 3, 1 Final counter: -1

4. CODE GENERATION TEST CASES

4.1 Variable Storage Testing

4.1.1 Large Number Handling

```
number big1;
number big2;
number result;

big1 := 12345678901234567890123456789012345678901234567890;
big2 := 98765432109876543210987654321098765432109876543210;
result := big1;
result += big2;
write "Result: " and result;
```

Output: 111111111011111111110111111111101111111110111111111100

4.1.2 Arithmetic Operations

```
number a;
number b;
number c;

a := 1000000000000000000;
b := 999999999999999999;
c := a;
c += b;
write "Sum: " and c and newline;

c := a;
c -= b;
write "Difference: " and c and newline;
```

Output: `Sum: 1999999999999999999 Difference: 1`

4.2 Loop Execution Tests

4.2.1 Loop Counter Behaviour

```
number n;
n := 5;
write "Before loop: " and n and newline;
repeat n times {
    write "Loop iteration, n = " and n and newline;
}
```

Output: `Before loop: 5 Loop iteration, n = 5 Loop iteration, n = 4 Loop iteration, n = 3 Loop iteration, n = 2 Loop iteration, n = 1`

4.2.2 Zero and Negative Loop Counts

```
number zero;
number negative;

zero := 0;
negative := -5;

write "Zero loop:";
repeat zero times write "This should not print";
write "Done" and newline;

write "Negative loop:";
repeat negative times write "This should not print either";
write "Done" and newline;
```

Output: `Done`
`Runtime Error: Loop count cannot be negative at line 12, column 0.`
`Skipping loop.`
`Negative loop:Done`

4.3 Output Formatting Tests

4.3.1 Mixed Output

```
number x;
number y;
x := 42;
y := -17;

write "Numbers: " and x and " and " and y and newline;
write "Calculation: " and x and " + " and y and " = ";
x += y;
write x and newline;
```

Output:
```
Numbers: 42 and -17
Calculation: 42 + -17 = 25
```

4.3.2 Newline Handling

```
write "Line 1" and newline and "Line 2" and newline and newline and
"Line 4";
```

Output:
```
Line 1
Line 2

Line 4
```

5. ERROR HANDLING TEST CASES

5.1 Lexical Errors

```
number x;
x := @#$;
"unterminated string
*unterminated comment
```

```
Lexical error at myscript.plus:2:6: Unknown character encountered.
Lexical error at myscript.plus:4:0: Unterminated string literal.
Lexical error at myscript.plus:3:0: Unterminated comment block.
```

```
number overflow;
overflow :=
12345678901234567890123456789012345678901234567890123456789012
34567890123456789012345678901; *> 100 digits*
```

```
Lexical error at myscript.plus:2:117: Integer literal exceeds maximum
allowed digits.
```

5.2 Syntax Errors

```
number;
x := ;
repeat times;
write and;
{ number x; ;
```

Output: Lexer success.

```
Parser Error: No valid action for state 11 on token EndOfLine (';') at
line 1, column 6.
Parser Error: No valid action for state 21 on token EndOfLine (';') at
line 2, column 4.
Parser Error: No valid action for state 13 on token KEYWORD_TIMES
('times') at line 3, column 6.
Parser Error: No valid action for state 12 on token KEYWORD_AND ('and')
at line 4, column 5.
```

```
Parser Error: No valid action for state 0 on token OpenBlock ('{') at
line 1, column 0.
```

6. STRESS TEST CASES

6.1 Deep Nesting

```
number i1; number i2; number i3; number i4; number i5;
i1 := 2; i2 := 2; i3 := 2; i4 := 2; i5 := 2;

repeat i1 times {
    repeat i2 times {
        repeat i3 times {
            repeat i4 times {
                repeat i5 times {
                    write "Deep";
                }
            }
        }
    }
}
```

Output: `Deep 32 times`

6.2 Many Variables

```
number v1; number v2; number v3; number v4; number v5;
number v6; number v7; number v8; number v9; number v10;
v1 := 1; v2 := 2; v3 := 3; v4 := 4; v5 := 5, v6 := 6; v7 := 7; v8 := 8;
v9 := 9; v10 := v10;
v10 += v5;
write v10;
```

Output: 15

7. INTEGRATION TEST CASES

7.1 Complete Programs

7.1.1 Number Sequence Generator

```
number start;
number end;
number current;
number step;

start := 1;
end := 10;
step := 2;
current := start;
```

```
write "Sequence from " and start and " to " and end and " step " and
step and ":" and newline;

repeat end times {
    write current and " ";
    current += step;
}
write newline;
```

7.1.2 Simple Calculator

```
number a;
number b;
number result;

a := 15;
b := 7;

write "Calculator Demo:" and newline;
write "a = " and a and newline;
write "b = " and b and newline;

result := a;
result += b;
write "a + b = " and result and newline;

result := a;
result -= b;
write "a - b = " and result and newline;
```

Output:

```
Calculator Demo:
a = 15
b = 7
a + b = 22
a - b = 8
```

CONCLUSION: ALL TEST CASES RESULTED WITH NO UNEXPECTED ERRORS. TASK COMPLETED:

1. Demonstrate your interpreter handling various **error cases** as defined in the Plus++ language specification. Show the command used, the erroneous .ppp file content (briefly), and the **exact error message** your interpreter produces. Examples of errors to show:
    - Undeclared variable usage.
    - Syntax errors (e.g., missing ';', incorrect keyword, malformed statement).
    - Invalid number formats (e.g., 3.0, +5, - 5).

USAGE SCREENSHOTS

```
KEYWORD_NUMBER  number                  Line:1    Col:0
Identifier      a                       Line:1    Col:6
EndOfLine       ;                       Line:1    Col:8
KEYWORD_NUMBER  number                  Line:2    Col:0
Identifier      b                       Line:2    Col:6
EndOfLine       ;                       Line:2    Col:8
KEYWORD_NUMBER  number                  Line:3    Col:0
Identifier      result                  Line:3    Col:6
EndOfLine       ;                       Line:3    Col:13
Identifier      a                       Line:4    Col:0
AssignmentOp    :=                      Line:5    Col:1
IntConstant     15                      Line:5    Col:4
EndOfLine       ;                       Line:5    Col:7
Identifier      b                       Line:6    Col:0
AssignmentOp    :=                      Line:6    Col:1
IntConstant     7                       Line:6    Col:4
EndOfLine       ;                       Line:6    Col:6
KEYWORD_WRITE   write                   Line:7    Col:0
StringConstant  "Calculator Demo:"      Line:8    Col:5
KEYWORD_AND     and                     Line:8    Col:24
KEYWORD_NEWLINE newline                 Line:8    Col:28
EndOfLine       ;                       Line:8    Col:36
KEYWORD_WRITE   write                   Line:9    Col:0
StringConstant  "a = "                  Line:9    Col:5
KEYWORD_AND     and                     Line:9    Col:12
Identifier      a                       Line:9    Col:16
KEYWORD_AND     and                     Line:9    Col:18
KEYWORD_NEWLINE newline                 Line:9    Col:22
EndOfLine       ;                       Line:9    Col:30
KEYWORD_WRITE   write                   Line:10   Col:0
```

```
Program
  StatementList
    Declaration
      Identifier: a
    Declaration
      Identifier: b
    Declaration
      Identifier: result
    Assignment
      Identifier: a
      Int_Value
        Integer: 15
    Assignment
      Identifier: b
      Int_Value
        Integer: 7
    WriteStatement
      OutputList
        ListElement
          String: "Calculator Demo:"
        ListElement
          Newline
    WriteStatement
      OutputList
        ListElement
          String: "a = "
        ListElement
          Int_Value
            Identifier: a
```

```
      ListElement
        Newline
  WriteStatement
    OutputList
      ListElement
        String: "b = "
      ListElement
        Int_Value
          Identifier: b
      ListElement
        Newline
  Assignment
    Identifier: result
    Int_Value
      Identifier: a
  Increment
    Identifier: result
    Int_Value
      Identifier: b
  WriteStatement
    OutputList
      ListElement
        String: "a + b = "
      ListElement
        Int_Value
          Identifier: result
      ListElement
        Newline
  Assignment
    Identifier: result
```

```
        Identifier: a
    Decrement
      Identifier: result
      Int_Value
        Identifier: b
    WriteStatement
      OutputList
        ListElement
          String: "a - b = "
        ListElement
          Int_Value
            Identifier: result
        ListElement
          Newline

--- Starting Program Execution ---
[DEBUG] Declared variable 'a' with initial value 0.
[DEBUG] Declared variable 'b' with initial value 0.
[DEBUG] Declared variable 'result' with initial value 0.
[DEBUG] Assigned 'a' := 15.
[DEBUG] Assigned 'b' := 7.
Calculator Demo:
a = 15
b = 7
[DEBUG] Assigned 'result' := 15.
[DEBUG] Incremented 'result' by 7. New value: 22.
a + b = 22
[DEBUG] Assigned 'result' := 15.
[DEBUG] Decremented 'result' by 7. New value: 8.
a - b = 8
```

Output for code 7.1.2 calculator

```
--- Parsing Successful! Generated AS
DEBUG: root_ast type received in mai
Program
  StatementList
    Declaration
      Identifier: n
    Assignment
      Identifier: n
      Int_Value
        Integer: 5
    WriteStatement
      OutputList
        ListElement
          String: "Before loop: "
        ListElement
          Int_Value
            Identifier: n
        ListElement
          Newline
    LoopStatement
```

```
--- Starting Program Execution ---
[DEBUG] Declared variable 'n' with initial value 0.
[DEBUG] Assigned 'n' := 5.
Before loop: 5
[DEBUG] Starting loop for 'n' with initial value 5.
[DEBUG] Entering code block.
Loop iteration, n = 5
[DEBUG] Exiting code block.

[DEBUG] Entering code block.
Loop iteration, n = 4
[DEBUG] Exiting code block.

[DEBUG] Entering code block.
Loop iteration, n = 3
[DEBUG] Exiting code block.

[DEBUG] Entering code block.
Loop iteration, n = 2
[DEBUG] Exiting code block.

[DEBUG] Entering code block.
Loop iteration, n = 1
[DEBUG] Exiting code block.

[DEBUG] Loop for 'n' finished. Iterations completed: 5. Restored 'n' to 5.

--- Program Execution Finished ---
```

Output for code 4.2.1 loop

APPENDIX 1

The file-line locations of comprehensive prints.

FIRST sets: parser.c lines between 586-596
FOLLOW sets: parser.c lines between 697-707
LR(1) item sets: parser.c lines between 828-829
ACTION table: parser.c lines between 1075-1108
ACTIONs taken: parser.c lines 1166, 1167, 1168, 1173, 1205, 1207, 1229, 1239, 1241, 1252