BILKENT UNIVERSITY
COMPUTER SCIENCE DEPARTMENT
CS-315 PROGRAMMING LANGUAGE DESIGN
PROJECT 2 REPORT
LANGUAGE NAME: NO_SCRIPT
SPRING 2025
SECTION 1
14.03.2025
TEAM 17

Full Name	ID	Section
Emiralp İlgen	22203114	1
Perhat Amanlyyev	22201007	1
Simay Uygur	22203328	1

Contents

1. BNF of The Language

<program> ::= <BEGIN_TOK> <stmt_list> <END_TOK>

<stmt_list> ::= <stmt>
| <stmt_list> <stmt>

<stmt> ::= <while_stmt>
| <if_stmt>
| <for_stmt>
| <declaration_stmt>
| <function_declaration>
| <display_stmt>
| <no_statement>
| <return_stmt>
| <comment_block>
| <comment_line>
| <exp_statement>
| <block>

<declaration_stmt> ::= <declaration_with_assign_stmt>
| <declaration_without_assign_stmt>

<declaration_with_assign_stmt> ::= <type> <identifier_list> <ASSIGN>
<assignment_exp> <SC>
| <arr_normal_decl> <ASSIGN> <arr_normal_init> <SC>
| <arr_decl_with_size> <ASSIGN> <arr_init_with_elements> <SC>
| <arr_normal_decl> <ASSIGN> <NULL> <SC>

<declaration_without_assign_stmt> ::= <type> <identifier_list> <SC>
| <arr_decl_with_size> <SC>

<return_stmt> ::= return <expression> <SC> | return <SC>

<arr_decl_with_size> ::= <type> <identifier> <LSP> <int_const> <RSP>

<arr_normal_decl> ::= <type> <identifier> <LSP> <RSP>

<arr_normal_init> ::= “new” <type> <LSP> <int_const> <RSP>

<arr_init_with_elements> ::= <LCB> <exp_list> <RCB>

<exp_list> ::= <assignment_exp> <COMMA> <exp_list> | <assignment_exp>

<arr_access> ::= <identifier> <LSP> <expression> <RSP>

<no_statement> ::= <SC>

<display_stmt> ::= “display” <LP> <string> <RP> <SC> | “display” <LP> <expression> <RP> <SC>

<read_func> ::= “read” <LP> <identifier> <RP> | “read” <LP> <RP>

<comment_block> ::= <DIV_OP> <MUL_OP> <text> <MUL_OP> <DIV_OP>

<comment_line> ::= <DIV_OP> <DIV_OP> <comment_text_no_newline> <NEW_LINE>

<text> ::= <comment_text_no_newline> | <comment_text_no_newline> <NEW_LINE> <text>

<comment_text_no_newline> ::= ε | <symbol> | <whitespace> | <letter> | <digit
| <string> | <comment_text_no_newline> <symbol>
| <comment_text_no_newline> <whitespace>
| <comment_text_no_newline> <letter>
| <comment_text_no_newline> <digit>

<if_stmt> ::= if <LP> <logic_expr> <RP> <block> <else_stmt>

<else_stmt> ::= else if <LP> <logic_expr> <RP> <block> <else_stmt> | else <block> | ε

<block> ::= <LCB> <stmt_list> <RCB> | <LCB> <RCB>

<while_stmt> ::= “while” <LP> <logic_exp> <RP> <block>

<for_stmt> ::= “for” <LP> <declaration_with_assign_stmt> <logic_exp> <SC> <statement_exp> <RP> <block>

<function_declaration> ::= <func_header> <block>

<func_header> ::= <type> <method_declarator> | “void” <method_declarator>| <type> <LSP><RSP> <method_declarator>

<type> ::= “int” | “float” | “bool”

<method_declarator> ::= <identifier> <LP> <parameter_list> <RP> | <identifier> <LP> <RP>

<parameter_list> ::= <parameter_list> <COMMA> <parameter>

```
                          | <parameter>

<parameter> ::= <type> <identifier>
                | <type> <identifier>  <LSP> <RSP>

<expression> ::= <assignment_exp>

<exp_statement> ::= <statement_exp> <SC>

<statement_exp> ::= <assignment> | <function_call> |  <read_func>

<assignment_exp> ::= <logic_exp>
                   | <assignment>

<assignment> ::= <left_side> <ASSIGN> <assignment_exp>

<left_side> ::= <arr_access>
                | <identifier>

<logic_exp> ::= <logic_or_exp>

<logic_or_exp> ::= <logic_and_exp>
                     | <logic_or_exp> <OR> <logic_and_exp>

<logic_and_exp> ::= <eq_exp>
                      | <logic_and_exp> <AND> <eq_exp>

<eq_exp> ::= <rel_exp>
             | <eq_exp> <EQUAL_OP> <rel_exp>
             | <eq_exp> <NOT_EQ_OP> <rel_exp>

<rel_exp> ::= <add_exp>
             | <rel_exp> <LESS_OP> <add_exp>
             | <rel_exp> <GREAT_OP> <add_exp>
             | <rel_exp> <LESS_EQ_OP> <add_exp>
             | <rel_exp> <GREAT_EQ_OP> <add_exp>

<add_exp> ::= <mul_div_mod_exp>
              | <add_exp>  <ADD_OP>  <mul_div_mod_exp>
              | <add_exp>  <SUB_OP>  <mul_div_mod_exp>

<mul_div_mod_exp> ::= <factor>
                     | <mul_div_mod_exp> <MUL_OP> <factor>
                     | <mul_div_mod_exp> <DIV_OP> <factor>
```

| <mul_div_mod_exp> <MOD_OP> <factor>

<factor> ::= <u_exp>
| <factor> <EXP_OP> <u_exp>

<u_exp> ::= <ADD_OP> <u_exp>
| <SUB_OP> <u_exp>
| <u_exp_unsigned>

<u_exp_unsigned> ::= <primary_exp> | <NOT_OP> <u_exp>

<primary_exp> ::= <identifier>
| <const>
| <LP> <expression> <RP>
| <function_call>
| <arr_access>
| <read_func>

<const> ::= <int_const>
| <float_const>
| <bool_const>

<float_const> ::= <int_const> <DOT> <digits>

| <DOT> <digits>

<digits> ::= <digit> | <digits> <digit>

<int_const> ::= <digits>

<bool_const> ::= <true> | <false>

<true> ::= “true”

<false> ::= “false”

<function_call> ::= <function_name> <LP> <argument_list> <RP> | <function_name> <LP> <RP>

<function_name> ::= <identifier>

<argument_list> ::= <argument> | <argument> <COMMA> <argument_list>

<argument> ::= <expression>

<identifier_list> ::= <identifier>
| <identifier_list> <COMMA> <identifier>

<identifier> ::= <letter> <identifier_tail>

<identifier_tail> ::= ε
| <identifier_tail> <letter>
| <identifier_tail> <digit>

<string> ::= <QTN> <string_content> <QTN>

<string_content> ::= <string_char> | <string_content> <string_char> | ε

<string_char> ::= <letter> | <digit> | <symbol> | <whitespace>

<symbol> ::= <NOT_OP> | "@" | "#" | <MOD_OP> | <EXP_OP> | "&" | <MUL_OP> | <LP> | <RP> | <SUB_OP> | <ASSIGN> | <LCB> | <RCB> | <LSP> | <RSP> | ":" | <SC> | <QTN> | <COMMA> | <DOT> | <LESS_OP> | <GREAT_OP> | <DIV_OP> | "?" | "`" | "~"

<whitespace> ::= " " | "\t"

<letter> ::= "a" | "b" | "c" | "d" | "e" | "f" | "g" | "h" | "i" | "j" | "k" | "l" | "m" | "n" | "o" | "p" | "q" | "r" | "s" | "t" | "u" | "v" | "w" | "x" | "y" | "z" | "A" | "B" | "C" | "D" | "E" | "F" | "G" | "H" | "I" | "J" | "K" | "L" | "M" | "N" | "O" | "P" | "Q" | "R" | "S" | "T" | "U" | "V" | "W" | "X" | "Y" | "Z" | "_" | "$"

<digit> ::= "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9"

<NULL> ::= “null”

<NEW_LINE> ::= “\n”

<COMMA> ::= “,”

<LP> ::= “(“

<RP> ::= “)”

<LCB> ::= “{“

<RCB> ::= “}”

<LSP> ::= “[“

<RSP> ::= “]”

<SC> ::= “;”

<AND> ::= “&&”

<OR> ::= “||”

<ASSIGN> ::= “=”

<DOT> ::= “.”

<QTN> ::= “\“”

<ADD_OP> ::= “+”

<SUB_OP> ::= “-”

<MUL_OP> ::= “*”

<DIV_OP> ::= “/”

<EXP_OP> ::= “^”

<LESS_OP> ::= “<”

<GREAT_OP> ::= “>”

<LESS_EQ_OP> ::= “<=”

<GREAT_EQ_OP> ::= “>=”

<EQUAL_OP> ::= “==”

<NOT_EQ_OP> ::= “!=”

<MOD_OP> ::= “%”

<NOT_OP>::= “!”

2. Explanations

2.1 Conflict Encountered and Solved

In our previous version of BNF, there was a problem with two statements:

- <func_header> ::= <result_type> <method_declarator>
- <result_type> ::= <type> | “void” | <type> <LSP><RSP>

When trying to run the parser, the system issued a shift/reduce conflict because, during token initialization, it could not understand which of the tokens to use <type> or <result_type>, which is why we changed the structure to:

- <func_header> ::= <type> <method_declarator> | “void” <method_declarator>| <type> <LSP><RSP> <method_declarator>

2.2 Explanation of the Language Construction

1.<program> ::= <BEGIN_TOK> <stmt_list> <END_TOK>

Expression shows the structure of the “NO_SCRIPT” language, consisting of statement lists, representing the program’s start.

2.<stmt_list> ::= <stmt>

| <stmt_list> <stmt>

A statement list is built from one or more statements.

3.<stmt> ::= <while_stmt>

| <if_stmt>

| <for_stmt>

| <declaration_stmt>

| <function_declaration>

| <display_stmt>

| <no_statement>

| <return_stmt>

| <comment_block>

| <comment_line>

| <exp_statement>

| <block>

The statement is separated into different statements.

4.<declaration_stmt> ::= <declaration_with_assign_stmt>
| <declaration_without_assign_stmt>

Represents how values can be declared.

5.<declaration_with_assign_stmt> ::= <type> <identifier_list> <ASSIGN> <assignment_exp> <SC>
| <arr_normal_decl> <ASSIGN> <arr_normal_init> <SC>
| <arr_decl_with_size> <ASSIGN> <arr_init_with_elements> <SC>
| <arr_normal_decl> <ASSIGN> <NULL> <SC>

Represents the creation of data types and value assigning.

6.<declaration_without_assign_stmt> ::= <type> <identifier_list> <SC>
| <arr_decl_with_size> <SC>

Represents just the creation of data types without assignment.

7.<return_stmt> ::= return <expression> <SC> | return <SC>

Commands how the return of functions works.

8.<array_decl_with_size> ::= <type> <identifier> <LSP> <int_const> <RSP>

Declares array with its size.

9.<array_normal_decl> ::= <type> <identifier> <LSP> <RSP>

Declares array without its size.

10. <array_init_with_elements> ::= <LCB> <exp_list> <RCB> <SC>

Declares array by directly assigning to its values.

11.<exp_list> ::= <assignment_exp> <COMMA> <exp_list> | <assignment_exp>

list of assignable expressions for array initialization.

12.<arr_access> ::= <identifier> <LSP> <expression> <RSP>

The way the program accesses the array.

13.<no_statement> ::= <SC>

Empty statement.

14.<display_stmt> ::= “display” <LP> <string> <RP> <SC> | “display” <LP> <expression> <RP> <SC>

Represents the system output function.

15.<read_func> ::= “read” <LP> <identifier> <RP> | “read” <LP> <RP>

Represents the system input function.

16.<comment_block> ::= <DIV_OP> <MUL_OP> <text> <MUL_OP> <DIV_OP>

The way the user leaves the comment in the code.

17.<comment_line> ::= <DIV_OP> <DIV_OP> <comment_text_no_newline> <NEW_LINE>

The way the user leaves the comment in the code.

18.<text> ::= <comment_text_no_newline> | <comment_text_no_newline> <NEW_LINE> <text>

The feature for comments.

19. <comment_text_no_newline> ::= ε | <symbol> | <whitespace> | <letter> | <digit
| <string> | <comment_text_no_newline> <symbol>
| <comment_text_no_newline> <whitespace>
| <comment_text_no_newline> <letter>
| <comment_text_no_newline> <digit>

The possible content of comments.

20.<if_stmt> ::= if <LP> <logic_expr> <RP> <block> <else_stmt>

The way the if statement is initialized.

21.<else_stmt> ::= else if <LP> <logic_expr> <RP> <block> <else_stmt> | else <block> | ε

The way else statement is initialized. It supplies else if, if, if else … recursive statements.

22.<block> ::= <LCB> <stmt_list> <RCB> | <LCB> <RCB>

Represents the storage of statements in the curly brace necessary for if-else statements, loop statements, functions.

23.<while_stmt> ::= “while” <LP> <logic_exp> <RP> <block>

The way language initializes while loop.

24.<for_stmt> ::= “for” <LP> <declaration_with_assign_stmt> <logic_exp> <SC> <statement_exp> <RP> <block>

The way language initializes for loop.

25.<function_declaration> ::= <func_header> <block>

The way language initializes function calls.

26.<func_header> ::= <type> <method_declarator> | “void” <method_declarator>| <type> <LSP><RSP> <method_declarator>

The feature to set the function’s type and name.

27.<type> ::= “int” | “float” | “bool”

Shows types of data.

28.<method_declarator> ::= <identifier> <LP> <parameter_list> <RP> |

<identifier> <LP> <RP>

The way languages declare methods in the code.

29.<parameter_list> ::= <parameter_list> <COMMA> <parameter>

| <parameter>

Represents how many parameters the function will take.

30.<parameter> ::= <type> <identifier>

| <type> <identifier> <LSP> <RSP>

The way language understands parameter statements.

31.<expression> ::= <assignment_exp>

Handles the arithmetic, mathematical, and logical commands.

32.<exp_statement> ::= <statement_exp> <SC>

The format that handles the call of assignments, function calls, and I/O's.

33.<statement_exp> ::= <assignment> | <function_call> | <read_func>

The expression might have 3 different statements.

34.<assignment_exp> ::= <logic_exp>
| <assignment>

The format to handle logic and assignment expressions.

35.<assignment> ::= <left_side> <ASSIGN> <assignment_exp>

The way language is assigned.

36.<left_side> ::= <arr_access>
| <identifier>

Left part of the assignment.

37.<logic_exp> ::= <logic_or_exp>

One of the options of assignment expression. Arithmetic expressions have precedence over boolean expressions.

38.<logic_or_exp> ::= <logic_and_exp>
| <logic_or_exp> <OR> <logic_and_exp>

Tries to manage different types of expressions.

39.<logic_and_exp> ::= <eq_exp>
| <logic_and_exp> <AND> <eq_exp>

Manages the long list of expressions that checks equality.

40.<eq_exp> ::= <rel_exp>
| <eq_exp> <EQUAL_OP> <rel_exp>
| <eq_exp> <NOT_EQ_OP> <rel_exp>

Format of checking equality of 2 or more expressions.

41.<rel_exp> ::= <add_exp>

| <rel_exp> <LESS_OP> <add_exp>

| <rel_exp> <GREAT_OP> <add_exp>

| <rel_exp> <LESS_EQ_OP> <add_exp>

| <rel_exp> <GREAT_EQ_OP> <add_exp>

Format of checking of 2 or more expressions on relativeness.

42.<add_exp> ::= <mul_div_mod_exp>

| <add_exp> <ADD_OP> <mul_div_mod_exp>

| <add_exp> <SUB_OP> <mul_div_mod_exp>

Shows the precedence (+ | -) of arithmetic expressions**.**

43.<mul_div_mod_exp> ::= <factor>

| <mul_div_mod_exp> <MUL_OP> <factor>

| <mul_div_mod_exp> <DIV_OP> <factor>

| <mul_div_mod_exp> <MOD_OP> <factor>

Shows the precedence (* | / | %) of arithmetic expressions.

44.<factor> ::= <u_exp>

| <factor> <EXP_OP> <u_exp>

Shows the precedence (^) of arithmetic expressions.

45.<u_exp> ::= <ADD_OP> <u_exp>

| <SUB_OP> <u_exp>

| <u_exp_unsigned>

Handles the situations when one or more operators (+ | -) stay in front of the numeric values by clearing(removing them recursively). If yacc sees <ADD_OP>, it does not change the value but if yacc sees <SUB_OP>, it multiplies the value by -1. So do not need any minus numbers.

46.<u_exp_unsigned> ::= <primary_exp> | <NOT_OP> <u_exp>

Search if the expression is boolean, if yes it converts the value to the opposite. If it does not then it assumes that expression is the normal expression.

47.<primary_exp> ::= <identifier>

| <const>

| <LP> <expression> <RP>

| <function_call>

| <arr_access>

| <read_func>

Handles the most calculated one. There will be no more process without calculating or giving the fundamental expressions.

48.<const> ::= <int_const>

| <float_const>

| <bool_const>

Indicates the different data types

49.<float_const> ::= <int_const> <DOT> <digits>

| <DOT> <digits>

The list of digits that have floating point

50.<digits> ::= <digit> | <digits> <digit>

The list of digits starts from 0 to 9 and has many

51.<int_const> ::= <digits>

The list of digits without floating point

52.<bool_const> ::= <true> | <false>

Indicates the state of the expression

53.<true> ::= “true”

One of the states of the expression

54.<false> ::= “false”

One of the states of the expression

55.<function_call> ::= <function_name> <LP> <argument_list> <RP> | <function_name> <LP> <RP>

Represents how language understands when a method/function is called

56.<function_name> ::= <identifier>

Name of the function/method

57.<argument_list> ::= <argument> | <argument> <COMMA> <argument_list>

Represents the list of input values of the function/method

58.<argument> ::= <expression> <COMMA> <argument_list> | <expression>

Represents just one input of the function/method

59.<identifier_list> ::= <identifier>

| <identifier_list> <COMMA> <identifier>

Represents the name of variables that are used during the assignment declaration

60.<identifier> ::= <letter> <identifier_tail>

The way language gives the name for functions/types/arrays

61.<identifier_tail> ::= ε

| <identifier_tail> <letter>

| <identifier_tail> <digit>

The list of digits, letters

62.<string> ::= <QTN> <string_content> <QTN>

Initializes the string value

63.<string_content> ::= <string_char> | <string_content> <string_char> | ε

The set of characters represented in (“”)

64<string_char> ::= <letter> | <digit> | <symbol> | <whitespace>

One character that is in the string.

65.<symbol> ::= <NOT_OP> | "@" | "#" | <MOD_OP> | <EXP_OP> | "&" | <MUL_OP> | <LP> | <RP> | <SUB_OP> | <ASSIGN> | <LCB> | <RCB> | <LSP> | <RSP> | ":" | <SC> | <QTN> | <COMMA> | <DOT> | <LESS_OP> | <GREAT_OP> | <DIV_OP> | "?" | "`" | "~"

These symbols can be inside the comment line.

66.<whitespace> ::= " " | "\t"

Specifies the spaces inside the text.

67.<letter> ::= "a" | "b" | "c" | "d" | "e" | "f" | "g" | "h" | "i" | "j" | "k" | "l" | "m" | "n" | "o" | "p" | "q" | "r" | "s" | "t" | "u" | "v" | "w" | "x" | "y" | "z" | "A" | "B" | "C" | "D" | "E" | "F" | "G" | "H" | "I" | "J" | "K" | "L" | "M" | "N" | "O" | "P" | "Q" | "R" | "S" | "T" | "U" | "V" | "W" | "X" | "Y" | "Z" | "_" | "$"

The letters are defined in this way to properly identify the identifiers.

68.<digit> ::= "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9"

The numbers in the 10 system.

69.<NULL> ::= “null”

Indicates that the array is empty and uninitialized.

70.<NEW_LINE> ::= “\n”

New line

71.<COMMA> ::= “,”

Separator for stacked tokens

72.<LP> ::= “(“

Left parenthesis used to open functions (declarations of parameters, calls of arguments), if-else statements, loops

73.<RP> ::= “)”

Right parenthesis used to close functions (declarations of parameters, calls of arguments), if-else statements, loops

74.<LCB> ::= “{“

Used for array initialization list.

75.<RCB> ::= “}”

Used for array initialization list.

76.<LSP> ::= “[“

For general array functionalities.

77.<RSP> ::= “]”

For general array functionalities.

78.<SC> ::= “;”

A semicolon is used for the end of the statements.

79.<AND> ::= “&&”

Boolean and operations.

80.<OR> ::= “||”

Boolean or operations.

81.<ASSIGN> ::= “=”

Used for assignment operations.

82.<DOT> ::= “.”

For float definitions.

83.<QTN> ::= “\“”

For string definitions

84.<ADD_OP> ::= “+”

Used as a summation operation or unary minus

85.<SUB_OP> ::= “-”

Used as a subtraction operation or unary minus

86.<MUL_OP> ::= “*”

Used as a multiplication operation or unary plus

87.<DIV_OP> ::= “/”

Used as a divider or in comment line or comment blocks.

88.<EXP_OP> ::= “^”

Used as exponent indicator.

89.<LESS_OP> ::= “<”

Boolean less than operator.

90.<GREAT_OP> ::= “>”

Boolean greater than operator.

91.<LESS_EQ_OP> ::= “<=”

Boolean less than or equal operator.

92.<GREAT_EQ_OP> ::= “>=”

Boolean less greater or equal operator.

93.<EQUAL_OP> ::= “==”

Boolean equality comparator operator.

94.<NOT_EQ_OP> ::= “!=”

Boolean not equality comparator operator.

95.<MOD_OP> ::= “%”

Used for modulo operations

96.<NOT_OP>::= “!”

Not operator for booleans like (!1=0) or (!0=1).

97. <BEGIN_TOK> ::= “begin”

Token to start a program.

98. <END_TOK> ::= “end”

Token to end the program.

3. Terminals

Operators: +, -, *, /, %, ==, !=, &&, ||, !, ^

Delimiters: {}, (), [], ;, ,

Assignment operator: =

Comparison operators: <, >, <=, >=

Operators: Used for arithmetic (+, -), logical (&&, ||), and comparison (==, !=).

3.1 Operator Precedence Hierarchy

From **highest** to **lowest** precedence:

- Parentheses (()) – Override precedence (N/A associativity)
- Unary Not(!): Logical NOT (!) Associativity: Right
- Unary (+/-): Unary minus or plus(-/+) Associativity: Right
- Exponentiation (^) – Associativity: Right
- Multiplication, Division, Modulo: Operators: *, /, % Associativity: Left
- Addition, Subtraction: Operators: +, - Associativity: Left
- Comparison Operators: Operators: <, >, <=, >= Associativity: Left
- Equality Checks: Operators: ==, != Associativity: Left
- Logical AND (&&) Associativity: Left
- Logical OR (||) Associativity: Left
- Assignment (=) – Variable assignment (Associativity: Right)

4. Tokens

4.1 Comments

- **Comment Block**:
  <comment_block> ::= <comment_block> ::= <DIV_OP> <MUL_OP> <text> <MUL_OP> <DIV_OP>
  <text> ::= <comment_text_no_newline> | <comment_text_no_newline> <NEW_LINE> <text>
- **Comment Line :**
  <comment_line> ::= <DIV_OP> <DIV_OP> <comment_text_no_newline> <NEW_LINE>
- **Common constructs that are used in both comments**:
  <comment_text_no_newline> ::= ε | <symbol> | <whitespace> | <letter> | <digit
  | <string> | <comment_text_no_newline> <symbol>
  | <comment_text_no_newline> <whitespace>
  | <comment_text_no_newline> <letter>
  | <comment_text_no_newline> <digit>

Comments in `NO_SCRIPT` are used for annotating code. Single-line comments (“//”) allow short explanations, while multi-line comments (“/* */”) help with longer documentation. Distinguishes comments from code very well. The simple syntax makes it easy to document code. Also, there are two ways to write the comments, therefore writability is good for our language Ensures comments do not interfere with code execution.

4.2 Identifiers

<identifier> ::= <letter> <identifier_tail>
<identifier_tail> ::= ε | <identifier_tail> <letter> | <identifier_tail> <digit>

Identifiers must start with a letter and can include letters, digits, and underscores in our language. Additionally, our language uses identifiers to name variables, functions, and other user-defined elements. They must start with a letter and can include letters, digits, and (“_”),(“$”). This ensures that all variable names are human-readable and easy to differentiate from numeric literals and reserved keywords. Allows a flexible range of names for variables (integer, making programming easier. Ensures variables are distinct from keywords by requiring them to start with a letter. The restricted format prevents conflicts with system symbols or reserved words.

4.3 Literals

- **Numeric Literals:** <int_const> ::= <digits>

  <float_const> ::= <int_const>.<digits>

- **String Literals:** <string> ::= <QTN> <string_content> <QTN>

  <string_content> ::= <string_char> | <string_content> <string_char> | ε

  <string_char> ::= <letter> | <digit> | <symbol> | <whitespace>

Strings are enclosed in quotation marks (“”) to avoid ambiguity. Prevents confusion with reserved symbols.

- **Boolean Literal:** <bool_const> ::= "true" | "false"

Uses common `true`/`false` values. Ensures strict boolean evaluation.

- **Null Literals:** <NULL> ::= "null"

Uses common and easy-to-remember definitions. Enables users to write **arrays without initializing them**. Guarantees that data collection might be empty.

4.4 Reserved Words

1. begin: is used as a token to indicate the start of the file (program)
2. end: is used as a token to indicate the end of the file (program)
3. return: is used as a token for return statements
4. if: is used as a token for logic statements
5. else: is used as a token for logic statements
6. while: is used as a token for the while loop
7. for: is used as a token for the loop
8. new: is used as a token to init data collection(array)
9. display: is used as a token to print
10. read: is used as a token to take input from the user
11. true: is used in the bool
12. false: is used in the bool
13. null: is used as a token to initialize empty data collection(array)
14. void: is used as a token for function return type
15. int: is used as a token to init int type
16. float: is used as a token to init float type
17. bool: is used as a token to init bool type

Reserved words are predefined keywords that cannot be used as identifiers. They have fixed meanings, reducing confusion. Prevent keywords from being misused as variable names. Ensures clear syntax for control flow and data types.

5. Evaluation of Language Design Criteria

5.1 Readability

To maintain simplicity and clarity, the NO_SCRIPT language was created by adhering to the standards of popular programming languages. Semicolons (;) are used deliberately to divide statements, making every line of execution distinct. To keep things structured, variables and function parameters are separated by commas (,). The language defines control flow with easy-to-understand keywords like `if`, `else`, `while`, `for`, and for data types `int`, `float`, `bool`. These components help to make the language very readable and simple for programmers to understand. Furthermore, code blocks are surrounded by curly brackets ({}), which eliminate ambiguity by clearly defining the start and finish of scopes. Every loop and if-else statement needs a block, which may be empty. In contrast to several scripting languages, NO_SCRIPT does not provide multiple methods for carrying out the same task, guaranteeing uniformity and minimizing the possibility of misunderstanding.

5.2 Writability

The simple and organized syntax of the NO_SCRIPT language contributes to its excellent writability. The language is easy to develop and maintain since variable declarations, loops, conditional statements, and function definitions all adhere to a simple structure. There are only necessary control structures in the language. The only iteration techniques are while loops (while) and for loops (for). The operators (+, -, *, /, ==,!=, <, >, &&, ||) are easy to employ since they correspond to their conventional mathematical and logical meanings. The precedence rules are maintained in operators and are defined for **negative numbers**. Since this construct can identify any primary construct or add/minus operator, numbers can be identified by signs, but those signs are unary operators with high precedence. To avoid implicit actions that could confuse, arrays are handled through explicit initialization using new. Additionally, expressions can be written inside the array after initializing it with {}.

NO_SCRIPT guarantees that programmers can learn and write code fast without having to deal with needless complexity by preserving a single, standard syntax for every construct.

5.3 Reliability

NO_SCRIPT uses strict typing and organized syntax rules to highlight dependability. Each variable has a data type (int, float, or bool) that is explicitly defined, so type-checking stops improper actions (such as adding a string and an integer). However, with implicit conversion. To remove uncertainty in code execution, all control structures (if, while, and for) must be encapsulated in curly brackets {}. To lower the possibility of uninitialized memory access, arrays must be explicitly initialized using new. To prevent interfering with program execution, comments are encased in distinct delimiters (// for single-line, /* */ for multi-line). NO_SCRIPT ensures that programs function predictably by imposing certain structured constraints, which lowers unexpected errors and improves code dependability.

6. Further Explanations

6.1 Cases

1. If an **array** is accessed with an expression like float x = 5.5; arr[x] = 10; or arr[5.5] = 10; meaning that <exp> in arr[<exp>] can be reduced to a **float**, it should result in an **error**. If an **array index is a negative integer**, the **compiler** should **generate** an error because array indices must always be non-negative integers. int x = -5; arr[x] = 10; or arr[-5] = 10;

2. **Types have a hierarchy of precedence: float > int.** According to this hierarchy, if <exp> contains any float, the result will be considered a float.
   For example:
   int x = 5;
   float y = 1.2;
   display(x + y);

   According to the definition, the following example should also result in a compiler error:
   int x = 5;
   float y = 1.2;
   int z = x + y; //error

   int x = 5;
   float y = 1.2;
   float z = x + y; //not error

   Since x + y is an <exp>, and y is a float, x will be converted to float, then added to y, resulting in a float. Consequently, the display function will output a float.

3. For bool variables, if the assigned value is not 0, 0.0, or false, then x should be set to 1 (integer). Otherwise, it should be considered 0 (integer) by the compiler.
   Example:
   bool x = 5;
   bool y = -3.2;
   bool z = true;
   bool w = 0;
   bool t = 0.0;
   bool u = false;

In an if(<exp>) statement, <exp> will always evaluate to true unless it is false, 0 (integer), or **0.0** (float). The same rule applies to while(<exp>) and the second substatement in a for loop.

For example:

```
for (int i = 10; i; i--) {
   display(i);  // This will execute as long as i is nonzero
}
```

If a bool identifier is true, it will be converted to 1 (integer) or 1.0 (float) based on the other expressions in <exp>, following the type precedence rule (float > int).

```
bool x = 5;  (true)
float y = 1.2;
int t = 10;
float z = x + y + t;  //no error because bool and int converted to float.
```

4. In every assignment (excluding arrays), the compiler should check whether the assigned **value matches the declared type (float, int, or bool).**

   For example:

   ```
   int x = 5;        // Valid
   float y = 3.14;   // Valid
   bool z = true;    // Valid
   ```

5. Two functions **cannot have the same name**, regardless of their parameters or return types. If a function is redefined with the same name, the **compiler should generate an** error.

   Example:

   ```
   void func() {
      display("Hello");
   }
   int func() {  //Error: Redeclaration of 'func'
      return 5;
   }
   ```

   Example:

   ```
   void func1() {
      display("Hello");
   }
   int func2() {
      return 5;
   }  // Valid: Different function names
   ```

6. When initializing an array using an <exp> list inside {}, the compiler should check whether each <exp> matches the array's type.
   For example:
   ```
   int arr[] = {1, 2, 3};  // Valid
   float arr2[] = {1.1, 2.2, 3.3};  // Valid
   bool arr3[] = {true, false, true};  //  Valid
   bool arr6[] = {1, 0, 2};  //  Valid, assigned to {true, false, true}
   int arr4[] = {1, 2, 3.5};  // Error
   float arr5[] = {1, 2.2, true};  //  Error
   ```

7. **In a static scope, multiple variables cannot have the same name.** Even if their types are different, redeclaring a variable with the same name should result in a compiler error.
   ```
   bool x;
   int x;  // Error: 'x' is already declared

   void func(int x) {
      bool x = true;  // Error: 'x' is already declared as a parameter
   }
   ```

8. The compiler should check whether the **types of function arguments match the expected parameter types.** (Pass by reference is selected for argument pass)
   For example:
   ```
   void func(int x, float y) {
      display(x, y);
   }

   func(5, 3.14);  // Valid
   func(2.5, 3.14);  // Error
   func(5, true);  // Valid (true is counted as 1.0 because float conversion)
   ```

9. A variable cannot be **assigned** a value **if it has not been declared beforehand**. The compiler should enforce this rule and generate an error if an undeclared variable is used in an assignment.
   Example:
   ```
   num = 25;  // Error: 'num' is not declared
   ```

   To avoid this error, the variable must be declared first:
   ```
   int num;
   num = 25;  // Valid: 'num' was declared before the assignment
   ```

10. In a method, all return statements must return a value that matches the function's declared return type, unless the function is void. The compiler should enforce this rule and generate an error if there is a type mismatch.
    Example:

```
int func(int x) {
    if (1) {
        return 1;  // Valid: 1 is an int
    } else {
        return 0;  // Valid: 0 is an int
    }
}
```

    However, the following cases should cause compiler errors:

```
float func() {
    return 5;  // Error: Function expects a float, but returning an
               //int
}

bool func() {
    return 2.5;  // Error: Function expects a bool, but returning a
                 //float
}

void func() {
    return 10;  // Error: Void function cannot return a value return;
                // is true
}
void func() {
    display("Before return");
    return;  // Valid: Stops function execution
    display("This will never execute");  // Unreachable code
}
```

NOTE: All of the examples mentioned are code fragments therefore in the file each of them should be surrounded with “begin” and “end” words.

6.2 Syntax vs. Semantics

Our language distinguishes between semantic correctness (managed at runtime or post-parsing) and syntactic validity (managed by the parser). The parser (yacc’s role) makes sure that the code complies with grammatical rules (e.g., statements end with `;`, brackets match). Type inconsistencies, incorrect operations, or runtime errors are delayed due to semantics (Post-Parsing) One of the tools for static analysis is **a separate semantic analyzer,** such as a type checker. Validation that is dynamic while the program is running.

Phases are used to classify and address strategy errors:

1. Syntax errors, such as missing `;`, are detected by the parser.
2. Semantic errors: detected by runtime or static analysis (e.g., `arr[5.5]`).
3. Logical errors (such as infinite loops) are left to debugging tools.

For instance:

```
float x = 5;
int arr[10];
arr[x]; // Syntactically correct (accepted by the parser), but subsequently identified as a semantic error.
```

Workflow:
Parser (YACC) → Syntax Tree → Semantic Analyzer → Runtime

6.3 Grammar Simplified

The grammar is kept clear and complexity is decreased by avoiding type checks in the parser:
YACC does not have a symbol table. The parser does not enforce type rules or keep track of variable types.
Advantage: absence of ambiguity. The BNF is made simpler by rules like `<expression>` that do not divide into `<int_expr> or <float_expr>`.

6.4 Conventions Provide Dependability

Reliability is attained without strict compiler-enforced restrictions such as using a separate runtime guard. Types are validated at runtime by essential actions (like array access).

6.5 Trade-offs and Justification

The parser rules have been simplified. The writability is flexible and script-like. The grammar is easier to extend and alter. The **trade-off is that later in the development process**, type mistakes were discovered. Strong runtime tests are necessary. There is less "safety" when coding.

Advantages that are accomplished by separating syntax from semantics:

- Simplicity: Clear grammar that is simple to understand.
- Flexibility: Code can be written and modified.
- Practical Reliability: Important mistakes are detected after parsing without compromising usability.

7. The Commands To Run The Example Programs

1. ./parser < CS315_S25_Team17_1.txt
2. ./parser < CS315_S25_Team17_1_syntax_error.txt
3. ./parser < CS315_S25_Team17_2.txt
4. ./parser < CS315_S25_Team17_2_syntax_error.txt
5. ./parser < CS315_S25_Team17_3.txt
6. ./parser < CS315_S25_Team17_3_syntax_error.txt
7. ./parser < CS315_S25_Team17_4.txt
8. ./parser < CS315_S25_Team17_4_syntax_error.txt
9. ./parser < CS315_S25_Team17_5.txt
10. ./parser < CS315_S25_Team17_5_syntax_error.txt