BILKENT UNIVERSITY
COMPUTER SCIENCE DEPARTMENT
CS 315

PROGRAMMING LANGUAGE DESIGN REPORT

LANGUAGE NAME: NO_SCRIPT
SPRING 2025
SECTION 1
23.02.2025
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> ::= <stmt_list>

<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>

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

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

<array_normal_init> ::= “new” <type> <LSP> <int_const> <RSP> <SC>

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

<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> <SC> <logic_exp> <SC>
<statement_exp> <RP> <block>

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

<func_header> ::= <result type> <method_declarator>

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

<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> | ! <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> <COMMA> <argument_list> | <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. Terminals

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

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

Assignment operator: =

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

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

Precedence: Multiplication (*), division (/), and modulo (%) come before addition (+) and subtraction (-).

3. Tokens

3.1 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.

3.2 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, identifiers are used for naming variables, functions, and other user-defined elements in our language. 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.

3.3 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.

3.4 Reserved Words

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

4. Evaluation of Language Design Criteria

4.1 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.

4.2 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.

4.3 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.

5. Explanation of the Language Construction

1.<program> ::= <stmt_list>

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

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 user leaves the comment in the code.

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

The way 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 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> <SC> <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> ::= <result type> <method_declarator>

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

27.<result_type> ::= <type> | “void” | <type> <LSP><RSP>

Shows types of the functions.

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

Shows types of data.

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

<identifier> <LP>   <RP>

The way languages declare methods in the code.

30.<parameter_list> ::= <parameter_list> <COMMA> <parameter>
| <parameter>

Represents how many parameters the function will take.

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

The way language understands parameter statements.

32.<expression> ::= <assignment_exp>

Handles the arithmetic, mathematical, and logical commands.

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

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

34.<statement_exp> ::= <assignment> | <function_call> | <read_func>
Expression might have 3 different statements.

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

The format to handle logic and assignment expressions.

36.<assignment> ::= <left_side> <ASSIGN> <assignment_exp>
The way language assigns.

37.<left_side> ::= <arr_access>
| <identifier>
Left part of assignment.

38.<logic_exp> ::= <logic_or_exp>
One of the options of assignment expression. Arithmetic expressions have precedence on boolean expressions.
39.<logic_or_exp> ::= <logic_and_exp>
| <logic_or_exp> <OR> <logic_and_exp>
Tries to manage different types of expressions.
40.<logic_and_exp> ::= <eq_exp>
| <logic_and_exp> <AND> <eq_exp>
Manages the long list of expressions that checks equality.
41.<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.
42.<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.

43.<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**.**

44.<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.

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

Shows the precedence(^) of arithmetic expressions.

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

Handles the situations when one or more operators(+ | -) staying 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.

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

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

48.<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 calculate or giving the fundamental expressions.

49.<const> ::= <int_const>

| <float_const>

| <bool_const>

Indicates the different data types

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

| <DOT> <digits>

The list of digits that have floating point

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

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

52.<int_const> ::= <digits>

The list of digits without floating point

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

Indicates the state of the expression

54.<true> ::= "true"

One of the states of the expression

55.<false> ::= "false"

One of the states of the expression

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

Represents how language understands when a method/function is called

57.<function_name> ::= <identifier>

Name of the function/method

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

Represents the list of input values of the function/method

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

Represents just one input of the function/method

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

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

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

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

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

The list of digits, letters

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

Initializes the string value

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

The set of characters represented in (“”)

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

One character that is in the string.

66.<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.

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

Specifies the spaces inside the text.

68.<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.

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

The numbers in the 10 system.

70.<NULL> ::= “null”

Indicates that the array is empty and uninitialized.

71.<NEW_LINE> ::= “\n”

New line

72.<COMMA> ::= “,”

Separator for stacked tokens

73.<LP> ::= “(“

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

74.<RP> ::= “)”

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

75.<LCB> ::= “{“

Used for array initialization list.

76.<RCB> ::= “}”

Used for array initialization list.

77.<LSP> ::= “[“

For general array functionalities.

78.<RSP> ::= “]”

For general array functionalities.

79.<SC> ::= “;”

Semicolon is used for end of the statements.

80.<AND> ::= “&&”

Boolean and operations.

81.<OR> ::= “||”

Boolean or operations.

82.<ASSIGN> ::= “=”

Used for assignment operations.

83.<DOT> ::= “.”

For float definitions.

84.<QTN> ::= “\“”

For string definitions

85.<ADD_OP> ::= “+”

Used as a summation operation or unary minus

86.<SUB_OP> ::= “-”

Used as a subtraction operation or unary minus

87.<MUL_OP> ::= “*”

Used as a multiplication operation or unary plus

88.<DIV_OP> ::= “/”
Used as a divider or in comment line or comment blocks.

89.<EXP_OP> ::= “^”
Used as exponent indicator.

90.<LESS_OP> ::= “<”
Boolean less than operator.

91.<GREAT_OP> ::= “>”
Boolean greater than operator.

92.<LESS_EQ_OP> ::= “<=”
Boolean less than or equal operator.

93.<GREAT_EQ_OP> ::= “>=”
Boolean less greater or equal operator.

94.<EQUAL_OP> ::= “==”
Boolean equality comparator operator.

95.<NOT_EQ_OP> ::= “!=”
Boolean not equality comparator operator.

96.<MOD_OP> ::= “%”
Used for modulo operations

97.<NOT_OP>::= “!”
Not operator for booleans like (!1=0) or (!0=1).

6. Lexical Definition

```
%option main
letter       [a-zA-Z_$]
digit        [0-9]
float        {digit}*(\.)?{digit}+
alphanumeric ({letter}|{digit})
int          {digit}+
identifier   {letter}({alphanumeric}|_)*
bool         (true)|(false)
comment      \/\*([^*]|\*+[^\/])*\*+\/
string       \"[^\"]*\"
%%
if          { printf("IF "); }
else        { printf("ELSE "); }
while       { printf("WHILE "); }
do          { printf("DO ") ; }
for         { printf("FOR "); }
display     { printf("DISPLAY "); }
read        { printf("READ "); }
void        { printf("VOID "); }
return      { printf("RETURN "); }
new         { printf("NEW "); }
null        { printf("NULL "); }
{int}       { printf("INTEGER_CONST "); }
{float}     { printf("FLOAT_CONST "); }
{bool}      { printf("BOOLEAN ");}
"int"       { printf("INT_ID ");}
"float"     { printf("FLOAT_ID ");}
"bool"      { printf("BOOL_ID ");}
{identifier}  { printf("IDENTIFIER "); }
\&\&    { printf("AND "); }
\|\|    { printf("OR "); }
\=      { printf("ASSIGN "); }
\+      { printf("ADD_OP "); }
\-      { printf("SUB_OP "); }
\*      { printf("MUL_OP "); }
\/      { printf("DIV_OP "); }
\%      { printf("MOD_OP "); }
\^      { printf("EXP_OP "); }
\!      { printf("NOT_OP "); }
\<\=    { printf("LESS_EQ_OP "); }
\>\=    { printf("GREAT_EQ_OP "); }
\<      { printf("LESS_OP "); }
```

```
\>       { printf("GREAT_OP "); }
\=\=      { printf("EQUAL_OP "); }
\!\=     {printf("NOT_EQ_OP "); }
\(       { printf("LP "); }
\)       { printf("RP "); }
\{       { printf("LCB "); }
\}       { printf("RCB "); }
\[       { printf("LSP "); }
\]       { printf("RSP "); }
\;       { printf("SC "); }
\,       { printf("COMMA "); }
{string}      { printf("STRING "); }
[ \t]+        ;
\/\/.*      { printf("COMMENT_ONE_LINE "); }
{comment}  { printf("COMMENT_BLOCK "); }
%%
```

7. Example Programs

7.1 Program 1

```
/*
This is program 1
*/
int num = 25;
bool isEven = false;

// Func checking a number is even
bool checkEven(int x) {
   return (x % 2 == 0);
}

// Main Execution
while (num > 0) {
   isEven = checkEven(num);

   if (isEven) {
      display("Even number detected.");
   } else {
      display("Odd number detected.");
```

```
    }
    display("new num: " );
    display(num - 1);
    num = num - 1;

}
```

7.2 Program 2

```
/*
This is program 2
*/

float values[] = new float[5];
int sum = 0;

// Populate the array
for (int i = 0; i < 5; i = i + 1) {
    values[i] = i * 2.5;
    sum = sum + values[i];
}

display("Total sum is: ");
display(sum);
```

7.3 Program 3

```
/*
This is program 3
*/
int compute(int x, int y) {
    if (x > 10 && y < 5 || x == y) {
        return x + y * 2;
    } else {
        return x - y / 2;
    }
}

int result = 0;
for (int i = 1; i <= 10; i = i + 1) {
    for (int j = 0; j < i; j = j + 1) {
        result = compute(i, j);
        display("Result is: ");
        display(result);
    }
}

int factorial(int n) {
  if (n < 1) {
    return 1;
  }
  else if( n==1 ){   //to check else if
    display("one is reached. ");
    return 1;
  }
  else if {
    return n * factorial(n - 1);
  }
}

float[] initializeArray(int size) {
    float arr[] = new float[size];
    for (int i = 0; i < size; i = i + 1) {
        arr[i] = i * 2.5;
    }
    return arr;
}

int main() {

    int x = 10;
    float y = 3.14;
    bool isPositive = true;
    string message = "Welcome!";

    display(message);
    display("Factorial of x: ");
    display(factorial(x));

    display("Enter a number: ");
    x=read();
    // Logical Condition
    if (x > 0 && isPositive) {
        display("Number is positive.");
    } else {
        display("Number is not positive.");
    }

    // Loops
    for (int i = 0; i < 5; i = i + 1) {
        display("Loop iteration: ");
        display(i);
    }

    while (x > 0) {
        display("Decrementing x...");
        x = x - 1;
    }

    // Array Operations
    floatList numbers = initializeArray(5);
    display("Array Element: ");
    display(numbers[2]);

    /* Multiline Comment:
      This program tests all features.
       Including recursion, loops, arrays, and logical operations.
    */

    return 0;
}
```

7.4 Pseudo Program 1

```
/*
This is the first program that was given in the pseudocode
*/

int x,y,z;

display("x: ");
```

```
x= read();
y = read();
z = read();

while( x== 0 || y == 0 || z==0 ) {
  display("Numbers you entered should have non-zero values. Please enter again! \n x: ");
  x= read();
  display("y: ");
  y = read();
  display("z: ");
  z = read();
}
display( x * y * z);
```

7.5 Pseudo Program 2

```
/*
This is the second program that was given in the pseudocode
*/
float foo(float p, float q) {
    display("Function Name: foo");
    display("Parameter 1 (p): ");
    display(p);
    display("Parameter 2 (q): ");
    display(q);

    if (p > q) {
        return p;
    } else {
        return q;
    }
}

/* Main Execution */
int main() {
    float aList[4] = {3.15, 7, 0.03, -1.7};
    float bList[3] = {9, -1.2, +1.2};

    for (int i = 0; i < 4; i = i + 1) {
        for (int j = 0; j < 3; j = j + 1) {
            float a = aList[i];
            float b = bList[j];
            float c = foo(a, b);

            display("Values:");
            display("a = ");
            display(a);
            display("b = ");
            display(b);
            display("c (which is the maximum of a and b) = ");
            display(c);
        }
    }

    return 0;
}
```

8. Makefile

```
all: build run

build:
lex noscript.l
gcc -o noscript lex.yy.c

run:
./noscript < test1.txt
./noscript < test2.txt
./noscript < test3.txt
./noscript < pseudo1.txt
./noscript < pseudo2.txt

clean:
rm -f noscript
rm -f lex.yy.c
clear
```