CS315 Project 2
Team 9

Dilara Mandıracı 22101643 Section 2
Yusuf Toraman 22002885 Section 2
Işıl Özgü 22102276 Section 2

Fall 2023

Table of Contents

MINT LANGUAGE - BNF

PROGRAM START

<program> ::= start <stmt_list> end

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

<stmt> ::= <conditional_stmt>
| <non_conditional_stmt>

IF - ELIF - ELSE STATEMENTS

<conditional_stmt> ::= if <LP><logic_expr><RP><LC><stmt_list><RC><else_stmt>
| if
<LP><logic_expr><RP><LC><stmt_list><RC><elif_stmts><else_stmt>
| if <LP><logic_expr><RP><LC><stmt_list><RC>
| if <LP><logic_expr><RP><LC><stmt_list><RC><elif_stmts>

<else_stmt> ::= else <LC><stmt_list><RC>

<elif_stmts> ::= <elif_stmt>
| <elif_stmt><elif_stmts>

<elif_stmt> ::= elif <LP><logic_expr><RP><LC><stmt_list><RC>

LOGICAL EXPRESSIONS

<logic_expr> ::= <logic_expr> xor <basic_logic_expr>
| <logic_expr> or <basic_logic_expr>
| <and_operation>
| <basic_logic_expr>

<and_operation>::= <logic_expr> and <basic_logic_expr>

<basic_logic_expr> ::= <arithmetic_operation> >= <arithmetic_operation>
| <arithmetic_operation> <= <arithmetic_operation>
| <relational_operation>
| <LP><logic_expr><RP>

<relational_operation>::= <arithmetic_operation> > <arithmetic_operation>
| <arithmetic_operation> < <arithmetic_operationxp>
| <equality_operation>

<equality_operation>::= <arithmetic_operation> == <arithmetic_operation>
| <arithmetic_operation> != <arithmetic_operation>

NON-CONDITIONAL STATEMENTS

<non_conditional_stmt> ::= <arithmetic_operation><SC>
| <loop_stmt>
| <assignment_stmt><SC>
| <declaration_stmt><SC>
| <initalization_stmt><SC>
| <func_stmt>
| <io_stmt><SC>

DECLARATION, INITIALIZATION, ASSIGNMENT

<declaration_stmt> ::= Int <identifier>
| List <identifier> with size <arithmetic_operation>
| Matrix <identifier> with dimension <arithmetic_operation>

<initialization_stmt> ::= <int_initialization>
| List <identifier> = <list>
| Matrix <identifier> = <matrix>

<int_initialization>::= Int <identifier> = <arithmetic_operation>

<assignment_stmt>::= <identifier> = <arithmetic_operation>
| <list_indexing> = <arithmetic_operation>
| <identifier> = <list>

LIST CREATION

<list> ::= [<list_helper>]

<list_helper> ::= <one_d_list> | <matrix>

<one_d_list>::=<arithmetic_operation>
| <arithmetic_operation> , <one_d_list>

<matrix> ::= [<one_d_list>]
| [<one_d_list>] , <matrix>

LIST INDEXING

<list_indexing>::= <identifier>[<arithmetic_operation>]
| <identifier>[<arithmetic_operation>][<arithmetic_operation>]

ARITHMETIC OPERATIONS

<arithmetic_operation> ::= <arithmetic_operation> + <term>
| <arithmetic_operation> - <term>
| <term>

<term> ::= <term> * <factor>
| <term> / <factor>
| <term> % <factor>
| <factor>

<factor> ::= <exp> ** <factor>
| <exp>

<exp> ::= <LP><arithmetic_operation><RP>
| <const_or_var>

LOOPS

<loop_stmt>::= <while>
| <for>
| <do_while>

<while>::= while <LP><logic_expr><RP><LC><stmt_list><RC>

<for>::= for<LP><int_initialization><SC><logic_expr><SC><identifier> =
<arithmetic_operation><SC><RP><LC><stmt_list><RC>

<do_while>::= do<LC><stmt_list><RC>while<LP><logic_expr><RP><SC>

INPUT/OUTPUT

<io_stmt> ::= <input_stmt>
| <output_stmt>

<input_stmt> ::= in >> <identifier>

<output_stmt> ::= out << <output_body>

<output_body>::= <string_or_number>
| <string_or_number> , <output_body>

<string_or_number> ::= <string>
| <arithmetic_operation>

FUNCTIONS

<parameter_dec>::= Int <identifier>
| Int <identifier> ,<parameter_dec>
|

<parameter_call>::= <const_or_var>
| & <identifier>
| <const_or_var> , <parameter_call>
| & <identifier> , <parameter_call>
|

<func_stmt>::= func <identifier><LP><parameter_decs><RP><LC><stmt_list> return <arithmetic_operation><SC><RC>
| func <identifier><LP><parameter_decs><RP><LC> return <arithmetic_operation><SC><RC>

<func_call>::= <identifier><LP><parameter_calls><RP>

NUMBERS and VARIABLES

<const_or_var>::= <identifier>
| <const_int>
| <list_indexing>
| <func_call>

<const_int> ::= <digits>
|<sign><digits>

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

<identifier> ::= <letter>
| <identifier><letter>
| <identifier><digits>

COMMENTS

<comment> ::= <line_comment>
| <multiline_comment>

<line_comment> ::= // <text> \n

<multiline_comment> ## <text> ##

<text> ::=<characters>

<string>::= “<text>”

<characters>::=<character>
|<character> <characters>

<character>::= <digit>
|<letter>
|<symbol>

SYMBOLS

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

<symbol>::= <RP> | <LP> | [|] | { | } | , | <SC> | = | “ | | <sign> | <NL> | & | | | ! | < | > | @ | ~ | / | \ | * | _ | - | & | ½ | % | ^ | ? | . |

<NL>::= \n

<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

<LP>::= (
<RP>::=)
<SC>::= ;
<LC>::= {
<RC>::= }
<sign>::= + | -

EXPLANATION OF MINT LANGUAGE

1- PROGRAM START

<program> ::= start <stmt_list> end

This is the non-terminal state of our program. The program starts with the "start" keyword and ends with the "end" keyword. Between these keywords, a program can be written with statements. The reserved keywords increase the program's readability as the users can identify these simple words while scanning the file.

<stmt> ::= <conditional_stmt>
| <non_conditional_stmt>

A statement can be a non-conditional statement (arithmetic operations, loop statements, assignments, declarations, etc.), or a conditional statement (if, else and elif conditions). Since non-conditional and conditional statements cover all statements, the entire program is written with non-conditional and conditional statements.

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

To be able to write multiple statements into a single program, we have a statements list.

2- IF - ELIF - ELSE STATEMENTS

<conditional_stmt> ::= if <LP><logic_expr><RP><LC><stmt_list><RC><else_stmt>
| if
<LP><logic_expr><RP><LC><stmt_list><RC><elif_stmts><else_stmt>
| if <LP><logic_expr><RP><LC><stmt_list><RC>
| if <LP><logic_expr><RP><LC><stmt_list><RC><elif_stmts>

The matched and unmatched statements are removed, they are written to solve the dangling else problem in the first part of the project. However, left and right curly braces allowed us to solve this problem more easily. Therefore, conditional statements were first defined. <conditional_stmt> is designed to include all possible combinations of if, elif and else statements. Solving the dangling else problem using curly braces greatly increased both readability and writability.

<elif_stmts> ::= <elif_stmt>
| <elif_stmt><elif_stmts>

<elif_stmt> ::= elif <LP><logic_expr><RP><LC><stmt_list><RC>

The grammar of an “elif” (else if) statement is given. “elif” statements can have a place between “if” and “else” blocks. To have multiple “elif” statements followed by each other, we provide <elif_stmts> with right recursive. This convention allows the users to create many conditional branches, increasing the writability for users.

<else_stmt> ::= else <LC><stmt_list><RC>

This grammar rule specifies the structure of an "else" statement. It is used to define the statements executed when the preceding "if" or "elif" conditions are not met. The <stmt_list> within the braces <LC> and <RC> allows for multiple statements to be included in the "else" block, providing a straightforward way to handle scenarios where none of the conditional cases apply.

3- LOGICAL EXPRESSIONS

<logic_expr> ::= <logic_expr> xor <basic_logic_expr>
| <logic_expr> or <basic_logic_expr>
| <and_operation>
| <basic_logic_expr>

A logical expression can be a basic logical expression, which means only two operands and one logical operator. For example *(a < b)*. However, these basic logical expressions can be concatenated with logical operands like “and,” “or,” and “xor.” Notice that these operands are only logical and cannot perform bitwise operations. That is why the BNF notation of MINT language limits its users from notations like “a xor b.” Moreover, to give precedence to comparison expressions before logical operations, we introduced new abstractions like <basic_logic_expr> and to give precedence to “and” type logical operations <and_operation> was introduced as well.

Logical expression handles concatenating multiple basic logical expressions such as *a<b and c<d; also*, it enables the placement of parentheses between logical expressions such as *((a<b) and d<c)*. This combination is not only restricted to 2 expressions with the help of the second declaration of a logical expression, which is a <logic_expression> and can be a <basic_logic_expr>. Notice that these expressions are left-associative to increase readability and follow universal math rules.

<and_operation>::= <logic_expr> and <basic_logic_expr>

Multiple basic logical expressions can be connected with “and,” “or” and “xor” connectors. These reserved keywords increase the readability and writability for users with no previous programming experience and allow the creation of complex logical operations.

<basic_logic_expr> ::= <const_or_var> >= <const_or_var>
| <const_or_var> <= <const_or_var>
| <relational_operation>
| <LP><basic_logic_expr><RP>

<relational_operation>::= <const_or_var> > <const_or_var>
| <const_or_var> < <const_or_var>
| <equality_operation>

<equality_operation>::= <const_or_var> == <const_or_var>
| <const_or_var> != <const_or_var>

These abstractions allows the and logical operations to get parsed first by pushing them to the lower levels of the parse tree. This way, “==” and “!=” get parsed first, then “<” and “>,” then “<=” and “>=” then “xor” and “or,” and lastly, “and” statements.

4- NON-CONDITIONAL STATEMENTS

<non_conditional_stmt> ::= <arithmetic_operation><SC>
| <loop_stmt>
| <assignment_stmt><SC>
| <declaration_stmt><SC>
| <initalization_stmt><SC>
| <func_stmt>
| <io_stmt><SC>

A non-conditional statement can be an arithmetic operation, a loop statement, an assignment, a declaration, or an initialization statement. Also, function statements and input/output stream statements are considered non-conditional statements.

5- DECLARATION, INITIALIZATION, ASSIGNMENT

<declaration_stmt> ::= Int <identifier>
| List <identifier> with size <arithmetic_operation>
| Matrix <identifier> with dimension <arithmetic_operation>

A declaration statement means to declare a variable before assigning its right-hand side value. Our language allows us to declare three types of variables: an integer, a list, and a matrix. For convention, matrices are defined as square matrices only. Additionally, lists and matrices can only store integer values. In addition to Project 1, the programmer is able to declare the list's and matrix's size/dimension with an arithmetic operation. This feature significantly improves the programming language's writability by replacing the rigid requirement of declaring lists and matrices with fixed numbers, with a more flexible approach that allows for dynamic sizing.

<initialization_stmt> ::= <int_initialization>
| List <identifier> = <list>
| Matrix <identifier> = <matrix>

<int_initialization>::= Int <identifier> = <arithmetic_operation>

An initialization statement is when you declare a variable, and give its right-hand side value immediately by defining its type. The int initialization has its own abstraction to be used in "for" loops.

<assignment_stmt>::= <identifier> = <arithmetic_operation>
| <list_indexing> = <arithmetic_operation>
| <identifier> = <list>

An assignment statement means to assign a right-hand side value of a variable declared before or can change its value if it was initialized before. Lists and matrices can be assigned to their own type.

6- LIST CREATION

<list>::= [<list_helper>]

Represents <one_d_list> or <matrix> surrounded by the square brackets. The use of square brackets here makes the program easier to read.

<list_helper> ::= <one_d_list> | <matrix>

The "list helper" abstraction connects to one-dimensional lists and matrices. With this rule, MINT language provides writability benefits by differentiating between these structures, allowing programmers to quickly declare the data type, even if it is a <one_d_list> or a <matrix>.

<one_d_list>::=<arithmetic_operation>
| <arithmetic_operation>, <one_d_list>

This structure provides that <one_d_list> can hold multiple <arithmetic_operation> data separated by commas. <arithmetic_operation> includes <const_or_var> as a lower level (inside in the <exp>). We wanted to make it broader, so we changed <const_ot_var> to <arithmetic_operation> to allow users to put values into the list such as 'n+2', as all arithmetic operations obtain an integer value. This grammar for the <one_d_list> ensures reliability by providing a clear pattern for list definitions.

<matrix> ::= [<one_d_list>]
| [<one_d_list>], <matrix>

The collection of one-dimensional lists represents a multi-dimensional list matrix. The syntax provides writability for complex data structures by allowing lists to be nested.

7- LIST INDEXING

<list_indexing>::= <identifier>[<arithmetic_operation>]
| <identifier>[<arithmetic_operation>][<arithmetic_operation>]

A list or matrix can be indexed with square brackets, like some languages like Java and C++. This allows access to the element from the given position or assigns a new value to that position. The same thing for <arithmetic_operation> is valid here. Users can get values such as list[1], list[n] or list[i+1].

8- ARITHMETIC OPERATIONS

<arithmetic_operation> ::= <arithmetic_operation> + <term>
| <arithmetic_operation> - <term>
| <term>

An arithmetic operation for our language can be one of those: addition (+), subtraction (-), multiplication (*), division (/), exponentiation (**), and mod operation(%). The MINT language introduces addition and subtraction at the same parsing level as "arithmetic operation" to give precedence to operations. Notice that arithmetic operations are left-associative. This increases the readability of MINT language as it follows mathematical conventions. Thus, the users can easily follow the precedence of their complex arithmetic operations.

*<term> ::= <term> * <factor>*
| <term> / <factor>

| <term> % <factor>
| <factor>

The division, multiplication, and mod operations are introduced under “term” abstractions to give them higher precedence than addition and subtraction.

*<factor> ::= <exp> ** <factor>*
|<exp>

The last operation that has the highest precedence is exponentiation. So, this operation is in the lower level of the parse tree than the other operations. Notice that exponential operation is right associative by introducing right recursion. This increases the readability as it follows the same convention in mathematics.

<exp> ::= <LP><arithmetic_operation><RP>
| <const_or_var>

The lowest level is for adding parentheses to any arithmetic operation to give the highest precedence over the whole operations. The extra allowance for parentheses increases the readability and writability of complex arithmetic operations, as some users may want to enclose some parts or the whole arithmetic operation.

9- LOOPS

<loop_stmt>::= <while>|<for>|<do_while>

A loop statement can be a while loop, a for loop, or a do while loop.

<while>::= while <LP><logic_expr><RP><LC><stmt_list><RC>

While the logical expression is valid, the statements list will be executed continuously.

<for>::= for<LP><int_initialization><SC><logic_expr><SC><identifier> = <arithmetic_operation><SC><RP><LC><stmt_list><RC>

For a certain amount of time stated with the logical expression, the statement list will be executed repeatedly.

<do_while>::= do<LC><stmt_list><RC>while<LP><logic_expr><RP><SC>

Do while works similar to the Java and C programming languages. It first executes the statement list once and then checks the condition later. While the condition is valid, the statement will be executed until the condition is invalid.

10- INPUT/OUTPUT

<io_stmt>::= <input_stmt> | <output_stmt>

This rule specifies the fundamental format of an IO statement in MINT language. Two types of IO statements are input statements and output statements. The user must choose whether to send an output or take an input.

<input_stmt>::= in >> <identifier>

<output_stmt> ::= out << <output_body>

A variable and the keyword in >> are used to define the phrase "input_stmt." With the help of this rule, the application can accept user input and save it in a variable designated by the <identifier> tag.

A variable and the keyword out << define the phrase "output_stmt." With the help of this rule, the application can display output. These representations provide readability and writability to our language because the "in" and "out" keywords and the direction of the arrows represent the purpose of the rule. For example, "in >> x" indicates that the input will be transferred into the x variable.

<output_body>::= <string_or_number>
| <string_or_number> , <output_body>

<string_or_number> ::= <string>
| <arithmetic_operation>

The <output_body> represents the output specified in the program. The <output_body> can contain <string_or_number>. Multiple outputs can be shown in a single out << separated by commas. The <string_or_number> rule determines that the output can be either a string (<string>) or a constant/variable/arithmetic operation (<arithmetic_operation>). This provides orthogonality in what can be the output.

11- FUNCTIONS

<parameter_decs>::=<parameter_dec>
|

<parameter_calls>::=<parameter_call>
|

Parameters can take any number of integers in variable or constant form; additionally, they can be empty. By separating our abstractions into <parameter_…> and <parameters_…s>, we do not allow empty spacing while defining parameters separated with commas. For example, “x, 3, ,y” cannot be written. This convention increases the reliability of the MINT Programming language.

<parameter_dec>::= Int <identifier>
| Int <identifier> ,<parameter_dec>
<parameter_call>::= <const_or_var>
| & <identifier>
| <const_or_var> , <parameter_call>
| & <identifier> , <parameter_call>

Parameter definitions change based on function definition and function call, the separate abstraction forces the users to follow syntactic rules, which increases the reliability but takes away from writability. Moreover, users can pass parameters by value or by reference. Again, by separating the notations MINT language does not allow the regular integers to be passed as reference. This increases the writability, readability and prevents premature errors.

<func_stmt>::= func <identifier><LP><parameter_decs><RP><LC><stmt_list> return <arithmetic_operation><SC><RC>
| func <identifier><LP><parameter_decs><RP><LC> return <arithmetic_operation><SC><RC>

While defining a function, it is mandatory to return a single number, as the Project specification indicated. Notice that this definition only accepts one return statement. This may challenge experienced developers as they tend to use premature return statements in if-else statements, but as designers, we wanted to impose the Project specifications well. It is still possible to use other language conventions and return complex operations or even make recursive calls. This convention takes away from the writability of the code but increases the trustability as it forces the users to make only one return statement.

<func_call>::= <identifier><LP><parameter_calls><RP>

Function calls can be made with just the function identifier alongside the parameters enclosed by parentheses.

12- NUMBERS and VARIABLES

<const_or_var>::= <identifier>
| <const_int>
| <list_indexing>
| <func_call>

<const_or_var> format can be specified by this rule because <const_or_var> takes <const_int>, <list_indexing>, <func_call> and <arithmetic_operation>. It means that these five forms can be used when we see <const_or_var>.

<const_int>::= <digits>
|<sign><digits>

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

<identifier>::= <letter>
| <identifier><letter>
| <identifier><digits>

<const_int> is a sequence of digits that allows a sign (12, -35). <digits> is the representation of the numbers. <identifier> starts with a letter and can be followed by more letters or digits. This allows for variables like cs and cs315.

13 -COMMENTS

<comment>::= <line_comment>
| <multiline_comment>

Users can write both one-line comments and multi-line comments. This adds to the readability of the code as the users have the chance to describe their code without them being compiled.

<line_comment>::= // <text> \n
<multiline_comment>::= ## <text> ##

The line and multi-line comments are defined with different symbols, which may detract from the writability. However, after careful consideration and research, the developers decided for this convention to make the lexical analyzer understand the difference between each comment type. We derived our inspiration from [1]. Moreover, the comments are detected in Lex but are not returned as a token, because Yacc does not need to take them into consideration while parsing.

<text> ::=<characters>

<text> is a sequence of characters. 2 paragraphs below is the specified rule for <characters>.

<string>::= “<text>”

Defines a <text>, which is enclosed by quotation marks. This structure gives readability to the program because it allows the user to recognize easily the string in the code.

<characters>::=<character>
|<character> <characters>

This abstraction produces sequences of distinctive characters. The recursive structure reliably handles short and large strings and provides consistency in the language.

<character>::= <digit>
|<letter>
|<symbol>

<character> is a single character that can have a single <digit>, <letter>, or <symbol>. This non-terminal allows the users to freely define strings using different character types, increasing the orthogonality.

14- SYMBOLS

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

A numeric value from 0 to 9 is called <digit>.

*<symbol>::= <RP> | <LP> | [|] | { | } | , | <SC> | = | “ | |<NL>| <sign> | & | | | ! | < | > | @ | ~ | / | \ | * | _ | - | & | ½ | % | ^ | ? | . |*

Symbols are the specific characters that are used in our language.

<NL>::= \n

New line.

<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

Specified lower and upper case letters

<LP>::= (

Left Parenthesis.

<RP>::=)

Right Parenthesis.

<SC>::= ;

Denotes semicolon.

<LC>::= {

Left curly bracket.

<RC>::= }

Right curly bracket.

<sign>::= + | -

Denotes specified signs.

RESERVED WORDS

- **start :** is reserved to indicate where the program starts.
- **if :** is reserved to create a conditional statement.
- **elif :** is reserved to create another conditional statement to execute if the “if” part or any “elif” parts above it is.
- **else :** is reserved to create another conditional statement. If the “if” or any of the “elif” parts are not executed, the else statement will be executed.
- **and :** is reserved to connect basic logical expressions. E.g., “a<b and b<c”.
- **or :** is reserved to connect basic logical expressions. E.g., “a<b or x<y”.
- **xor :** is reserved to connect basic logical expressions. E.g., “a<b xor b<c”.
- **func :** is a reserved keyword to indicate the creation of a function.
- **with size :** is a reserved keyword used to initialize a list. With these reserved keywords, the size of the list can be expressed.
- **with dimension:** is a reserved keyword used to initialize Matrix. With these reserved keywords, the size of the matrix, a square matrix for convention, can be expressed. E.g., *Matrix m with dimension 3* indicates a square matrix with 3x3 dimensions.
- **do :** is reserved for creating a bottom checked loop. Notice that this loop executes once no matter what and a “while” reserved keyword is expected after the loop body. E.g., “do {...} while(i<4)” will run at least once and then the while condition is checked.
- **while :** is reserved for creating a top checked loop. E.g., “while(i<4)” will run while the i variable is smaller than 4.
- **for :** is reserved for creating a loop that runs based on count. E.g., “for(Int i =0; i<5; i=i+1)” will be executed 5 times.
- **return :** is reserved to indicate the return value of a function.
- **in :** is reserved for scanning input from the user.
- **out :** is reserved to demonstrate an output. To print desired texts or numeric values.
- **Int :** is reserved to indicate the type of a variable. E.g., *Int x = 5*.
- **List :** is reserved to indicate the type of a variable. E.g., *List mylist with size 4*.
- **Matrix :** is reserved to indicate the type of a variable. E.g., *Matrix m1 with dimension 4*.

- **end :** is reserved to indicate the end of the program.

LEX FILE

```
digit [0-9]
letter [A-Za-z]
sign [+-]
alphanumeric ({digit}|{letter})
%x IN_COMMENT
%%
"["                     return LSB;
"]"                     return RSB;
<INITIAL>{
"##"                     {
                        BEGIN(IN_COMMENT);}
                        }
                        <IN_COMMENT>{
"##"                    BEGIN(INITIAL);
[^#\n]
"#"
\n                      {extern int lineno; lineno++ ;}
}
start                   return START;
end                     return END;
func                    return FUNC;
return                   return RETURN;
while                   return WHILE;
for                     return FOR;
do                      return DO;
if                      return IF;
elif                    return ELIF;
else                    return ELSE;
Int                     return INT;
List                    return LIST;
```

```
Matrix                          return MATRIX;
"with size"                     return WITH_SIZE;
"with dimension"                return WITH_DIMENSION;
and                             return AND;
or                              return OR;
xor                             return XOR;
">="                            return GREATER_OR_EQUAL;
">"                             return GREATER;
"<"                             return SMALLER;
"<="                            return SMALLER_OR_EQUAL;
"=="                             return EQUALS;
"!="                            return NOT_EQUAL;
in[ \t]*>>                       return INPUT_STREAM;
out[ \t]*<<                     return OUTPUT_STREAM;
\%                              return MOD_OP;
\{                              return LC;
\}                              return RC;
\(                              return LP;
\)                              return RP;
\;                              return SC;
\&                              return REFERENCE;
\+                               return PLUS;
\-                              return MINUS;
\*                              return MULT_OP;
\/                              return DIV_OP;
\*\*                            return EXP_OP;
\,                              return COMMA;
\=                              return ASSIGN_OP;
\"[^(\")]*\"                    return STRING;
{letter}+{alphanumeric}*         return IDENTIFIER;
\n                              {extern int lineno; lineno++;}
```

```
{digit}+                    return CONST;

\V.*                        {}

.

%%

int yywrap(void){

return 1;

}
```

Notice that the CONST is taken without the sign because the unary or binary differentiation of - and + is not possible with lex. However, this convention was chosen to let Yacc make the differentiation of these symbols.

YACC FILE

```
%{
#include <stdio.h>
#include <stdlib.h>
int yylex(void);
void yyerror(char* s);
%}
%token  LSB
%token  RSB
%token  REFERENCE
%token  START
%token  END
%token  FUNC
%token  RETURN
%token  WHILE
%token  FOR
%token  DO
%token  IF
%token  ELIF
%token  ELSE
%token  INT
%token  LIST
%token  MATRIX
%token  WITH_SIZE
%token  WITH_DIMENSION
%token  INPUT_STREAM
%token  OUTPUT_STREAM
%token  MOD_OP
%token  LC
%token  RC
%token  LP
```

```
%token  RP

%token  SC

%token  PLUS

%token  MINUS

%token  MULT_OP

%token  DIV_OP

%token  EXP_OP

%token  COMMA

%token  ASSIGN_OP

%token  STRING

%token  IDENTIFIER

%token  CONST

%token  GREATER_OR_EQUAL

%token  GREATER

%token  SMALLER

%token  SMALLER_OR_EQUAL

%token  EQUALS

%token  NOT_EQUAL

%left   XOR OR

%left   AND

%%

// PROGRAM START

program:

START stmt_list END

stmt_list : stmt

          | stmt stmt_list

stmt : conditional_stmt

| non_conditional_stmt

| error SC {yyerrok;}
```

```
// CONDITIONAL (IF ELSE) STATEMENTS
conditional_stmt :
  IF LP logic_expr RP LC stmt_list RC else_stmt
        | IF LP logic_expr RP LC stmt_list RC elif_stmts else_stmt
        | IF LP logic_expr RP LC stmt_list RC
          | IF LP logic_expr RP LC stmt_list RC elif_stmts
else_stmt :
ELSE LC stmt_list RC
elif_stmts : elif_stmt
        | elif_stmt elif_stmts
elif_stmt : ELIF LP logic_expr RP LC stmt_list RC
// LOGICAL EXPRESSIONS
logic_expr :  logic_expr XOR basic_logic_expr
                    | logic_expr OR basic_logic_expr
                    | logic_expr AND basic_logic_expr
                    | basic_logic_expr
basic_logic_expr : arithmetic_operation GREATER_OR_EQUAL arithmetic_operation
                                | arithmetic_operation SMALLER_OR_EQUAL arithmetic_operation
                                | relational_operation
                                | LP logic_expr RP
relational_operation : arithmetic_operation GREATER arithmetic_operation
                                   | arithmetic_operation SMALLER arithmetic_operation
                                   | equality_operation
equality_operation :
          arithmetic_operation EQUALS arithmetic_operation
          | arithmetic_operation NOT_EQUAL arithmetic_operation
// NON-CONDITIONAL STATEMENTS
non_conditional_stmt: arithmetic_operation SC
          | loop_stmt
          | assignment_stmt SC
          | declaration_stmt SC
```

```
        | initialization_stmt SC
        | func_stmt
        | io_stmt SC
//DECLARATION, INITIALIZATION, ASSIGNMENT
declaration_stmt: INT IDENTIFIER
        | LIST IDENTIFIER WITH_SIZE arithmetic_operation
        | MATRIX IDENTIFIER WITH_DIMENSION arithmetic_operation
initialization_stmt: int_initiliazation
        | LIST IDENTIFIER ASSIGN_OP list
        | MATRIX IDENTIFIER ASSIGN_OP matrix
int_initiliazation:  INT IDENTIFIER ASSIGN_OP arithmetic_operation
assignment_stmt:
 IDENTIFIER ASSIGN_OP arithmetic_operation
        | list_indexing ASSIGN_OP arithmetic_operation
        | IDENTIFIER ASSIGN_OP list
// LIST CREATION
list : LSB list_helper RSB
list_helper : one_d_list
        | matrix
one_d_list : arithmetic_operation
        | arithmetic_operation COMMA one_d_list
matrix : LSB one_d_list RSB
        | LSB one_d_list RSB COMMA matrix
// LIST INDEXING
list_indexing: IDENTIFIER LSB arithmetic_operation RSB
        | IDENTIFIER LSB arithmetic_operation RSB LSB arithmetic_operation RSB
// ARITHMETIC OPERATION
arithmetic_operation: arithmetic_operation PLUS term
        | arithmetic_operation MINUS term
        | term
term :  term MULT_OP factor
```

```
| term DIV_OP factor

| term MOD_OP factor

| factor

factor :exp EXP_OP factor

            | exp

exp : LP arithmetic_operation RP

             | const_or_var

// LOOPS

loop_stmt:  while

            | for

            | do_while

while:  WHILE LP logic_expr RP LC stmt_list RC

for:  FOR LP int_initiliazation SC logic_expr SC IDENTIFIER ASSIGN_OP arithmetic_operation SC RP LC
stmt_list RC

do_while:  DO LC stmt_list RC WHILE LP logic_expr RP SC

// INPUTS / OUTPUTS

io_stmt: input_stmt

            | output_stmt

 input_stmt: INPUT_STREAM IDENTIFIER

output_stmt: OUTPUT_STREAM output_body

output_body: string_or_number

            | string_or_number COMMA output_body

string_or_number: STRING

            | arithmetic_operation

// FUNCTIONS

parameter_decs: parameter_dec

                        |

parameter_dec:    INT const_or_var

                        | INT const_or_var COMMA parameter_dec

 parameter_calls: parameter_call

                        |
```

```
parameter_call:   const_or_var
                | REFERENCE IDENTIFIER
                | const_or_var COMMA parameter_call
                | REFERENCE IDENTIFIER COMMA parameter_call
func_stmt:  FUNC IDENTIFIER LP parameter_decs RP LC stmt_list RETURN arithmetic_operation SC RC
          | FUNC IDENTIFIER LP parameter_decs RP LC RETURN arithmetic_operation SC RC
func_call:  IDENTIFIER LP parameter_calls RP
// NUMBERS and VARIABLES
const_or_var: IDENTIFIER
          | const_int
          | list_indexing
          | func_call
const_int:  CONST
          | MINUS CONST
          | PLUS CONST
%%
#include "lex.yy.c"
int lineno=1;
void yyerror(char *s){
  fprintf(stderr, "%s on line %d\n", s,lineno);
}
int main(){
 yyparse();
if(yynerrs==0)
{printf("Input program is valid\n");}
  return 0;
}
```

Notice that some association conventions like “%left” and “%right” used to simplify the BNF but the same rules are present between document specifications and yacc file. Also in “stmt” abstraction “error” token is being utilized to catch multiple syntax errors after skipping the error until a semicolon is seen. This way our parser works without any conflicts and parses MINT language as expectedly.

TEST PROGRAMS

TEST PROGRAM 1

```
start
out << "Enter ", 2 + -3 + +5 - 1 ," integers x, y and z:\n" ;
Int x ;
Int ; // SYNTAX ERROR ON LINE 4! It must be 'Int y ;'
Int z ;
in >> x ;
in >> y ;
in >> z ;

while ( x == 0 or y == 0 or z == 0 ) {
        out << "Enter only non-zero values. \n" ;
in >> x ;
in >> y ;
in >> z ;

}

Int result = x * y * z ) ; // SYNTAX ERROR ON LINE 18! There are unmatched parentheses.
out << result ;
/ comment // SYNTAX ERROR ON LINE 20! There is no such action.
end
```

TEST PROGRAM 2

```
start
func foo ( Int p , Int q )
{
   out    << "Function Name: foo\n" ;
   out << "p: " ;
   out << p ;
   out << "\nq: " ;
   out << q ;
   Int result ;
   if ( p > q ) {
      result = p ;
   }
   else { //13
      result = q ;
   }
return result ;
}

//This is a random recursive function to show the flexibility of our functions. No call is made for that function.
func recursiveTest (Int x)
{
   if(x % 101 == 0)
   {
      x = x / 2;
   }
   return 2 + recursiveTest(&x);
}

List listA with size 4 ;
List listB with size 3 ;
```

```
listA = [ 5 , 0 , 3 , -7 ] ; //22
listB = [ 9 , -2 , -1 ] ;

Int c ;
for ( Int i = 0 ; i < 4 ; i = i + 1 ; ) //27
{
   for ( Int j = 0 ; j < 3 ; j = j + 1 ; )
   {
      // Call foo function to store result in c
      c = foo (  A [ i ] , B [ j ] ) ;

      // Print
      out << "a: " ;
      out << A [ i ] ;
      out << "\nb: " ;
      out << B [ j ] ;
      out << "\nc: " ;
      out << c ;
   }
}

end
```

TEST PROGRAM 3

```
start

out << " This program aims to teach xor logical operator to cs 223 students. The program will initially take two signal
inputs from the user and calculate their xor results, if xor result is 1 it will print Correct ! and the program will terminate.
Else they will try again until they find the correct = true xor results of two signals. " ;

Int signal1 ;
Int signal2 ;
outtt << "Enter signal1 and signal2 : \n" ; // SYNTAX ERROR ON LINE 7! There is no such keyword for output stream.

do {
        in > signal1 ; // SYNTAX ERROR ON LINE 10! There is no such action. It must be 'in >>'
        in >> signal2 ;

        if ( ( ( signal1 == 1 ) xor ( signal2 == 0 ) ) or ( ( signal1 == 0) xor ( signal2 == 1 ) ) ) {
                out << "Correct ! \n" ;
        }
        else {
                out << "Try again. XOR result is wrong.\n" ;
        }

} while ( signal1 == signal2 ) ;

end
```

TEST PROGRAM 4

```
start

Int year ;
Int angel = 0 ;
Int digit = 10xys ; // SYNTAX ERROR ON LINE 5! It should be equal to an integer value, or identifier. Identifier cannot start with a number

out << "This program calculates your angel number based on your year of birth and creates a 2x2 Matrix filled with your angel number. Please enter your birth year:\n" ;

in >> year ;

while ( year > 0 ) {
        angel = (angel + year % digit ) ;
        year = year / 10 ;
}

angel = angel % 10 +(angel / 10)  %%% 10 ; // SYNTAX ERROR ON LINE 16! There is no such action '%%%'.

Matrix luckyM with dimension 2 ;
for ( Int i = 0; i < 2 ; i = i + 1 ; ){
        for ( Int j = 0 ; j < 2 ; j = j + 1 ; ) {
        luckyM [ i++ ] [ j ] = angel ; // SYNTAX ERROR ON LINE 21! There is no post increment operation in our language.
        }
}

out << "Your angel number is: " , angel ;
end
```

TEST PROGRAM 5

```
start

out << "This program has three functions, and it operates these functions according to the list that user determine.\n" ;

##mutlline
Comment
test##

out << "Choose a function: \n1. Sum odd/even check \n2. Find the largest number \n3. Find the smallest number\n" ;
Int choice ;
in >> choice ;

//one line comment test;random symbols2379479202**???##

Int size ;
out << "Enter the size of the list: " ;
in >> listSize ;
List list with size listSize ;

//Fill the List
for ( Int i = 0 ; i < size ; i = i + 1 ; ) {
        Int number ;
        in >> number ;
        list [ i ] = number ;
}
```

```
if ( choice == 1 ) {
        Int total = 0 ;
        for ( Int i = 0 ; i < listSize ; i = i + 1 ; ) {
                total = ( total + list [ i ] ) ;
        }

        if ( ( total % 2 ) == 0 ) {
                out << "The sum of the numbers is even.\n" ;
        }
else {
                out << "The sum of the numbers is odd.\n" ;
        }
}
elif ( choice == 2 ) {
        Int maxNum = list [ 0 ] ;
        for ( Int i = 0 ; i < listSize ; i = i + 1 ; ) {
                if ( list [ i ] > maxNum ) {
                        maxNum = list [ i ] ;
                }
        }
        out << "The largest number in the list is " , maxNum , ".\n" ;
}
else {
        Int minNum = list [ 0 ] ;
        for ( Int i = 0 ; i < listSize; i = i + 1 ; ) {
                if ( list [ i ] < minNum ) {
                        minNum = list [ i ] ;
                }
        }
        out << "The smallest number in the list is " , minNum , ".\n" ;
}

end
```

REFERENCES

[1] “Lexical Analysis With Flex, for Flex 2.6.2: How can I match C-style comments?,” *westes.github.io.* [Online] https://westes.github.io/flex/manual/How-can-I-match-C_002dstyle-comments_003f.html. [Accessed Oct. 14, 2023].