CS 315-01 Programming Language
Lexical Analyser for a Programming Language for a Boolean Language
Project 1 Report

TEAM 13
EMA LANGUAGE

Name **Surname**:	Bilkent ID:	**Section XX**:
Mithat Emre **GÜRBÜZ**	21901826	**Section 01**
Alphan **TULUKCU**	22003500	**Section 01**
Öykü Elis **TÜREGÜN**	21902976	**Section 01**

CONTENTS

1) BNF OF THE EMA LANGUAGE

```
<program>  ::=  <stmts>

<stmts>    ::=  <stmt> | <stmt> <stmts>

<stmt>     ::=    <decl_stmts>
                | <assign_stmt>
                | <cond_stmt>
                | <input_stmt>
                | <output_stmt>
                | <loop_stmt>
                | <comment>
                | <func_decl>
                | <func_call>
                | <array_decl>
                | <array_call>
                | <return_stmt>

<decl_stmts> ::= <decl_stmt>
                | <decl_stmt> COMMA <decl_stmts>
                | <decl_stmt> NEWLINE <decl_stmts>

<decl_stmt> ::= <type> <var_id> assgnop <expr>
                | <type> <var_id>
                | CONST <decl_stmt>
                | <var_id>

<assign_stmt> ::= <var_id> assgnop <expr>

<var_id> ::= IDENTIFIER

<cond_stmt> ::= if LP <exprs> RP LB <stmts> RB NEWLINE [else
<stmts>]
           |    if LP  <exprs> RP LB <stmts> RB NEWLINE elseif LP
          <exprs> RP LB <stmts> RB NEWLINE [else <stmts>]

<loop_stmts> ::=  <while_loop>
                | <for_loop>

<while_loop> ::= while  LP <exprs> RP LB <stmts> RB

<for_loop> ::=   for <var_id> loop  <array_call> LB <stmts> RB
           |     for <var_id> loop <array_stc> LB <stmts> RB

<exprs> ::=  <expr>
           | <expr> <exprs>
           | <expr> COMMA <exprs>

<expr> ::=   <bool_const>
           | <var_id> <operator> <var_id>
           | <var_id>
           | <input_stmt>
           | <func_call> | <array_call>
```

```
<operator> ::= AND | OR | NOT | IMPLIES | DOUBLE_IMPLIES | EQUAL |
NEQUAL

<type> ::= bool_type

<bool_const> ::= true | false

<func_decl> ::= <func_type> <func_id> LP <decl_stmts> RP LB
<stmts> RB

<func_id> ::=  <var_id>

<func_type> ::= void func | <type> func

<func_call> ::= <func_id> LP <exprs> RP
                | <func_id> LP RP

<return_stmt> ::= return <expr>
                | return LP <expr> RP

<comment> ::= comment <chars> comment

<chars> ::= letter | letter <chars>

<input_stmt> ::= IN LP <”chars”> RP

<output_stmt> ::= OUT LP <out_in> RP

<out_in> ::= <chars>
           | <var_id>
           | <var_id> COMMA <out_in>
           | <chars> COMMA <out_in>

<array_decl> ::= <type> <var_id> LS  RS
                | <type> <var_id> LS RS assgnop <array_stc>

<array_call> ::= <var_id> LS RS | <array_stc>

<array_stc> ::= LS <bool_var> RS

<bool_var> ::= <bool_const> COMMA <bool_var> | <bool_const>
```

2) DESCRIPTION OF NON-TERMINALS

<program>

Starting non-terminal of the EMA language. It contains all statements of the language.

<stmts>

Non-terminal includes all statements and also it is recursive for enabling programmer to write many statements.

<stmt>

Non-terminal describes one statement of the EMA language. Statement is the fundamental element of language. A statement can be many types such as declaration(s), assignment statement(s), condition statement, input, output, loops(for & while), comment, function declaration, function call, array declaration, array call, and return statement.

<decl_stmts>

Non-terminal includes all three types of declarations of EMA language. A declaration can be build by 1. only one declaration statement 2. many declarations separated by comma(recursive) 3. many declarations separated by lines(recursive).

<decl_stmt>

Non-terminal describes one declaration statement of EMA language. A declaration can be done in four ways: 1. declaration and assignment together 2. only declaration 3. constant variable declaration 4. only variable name for more than one declarations in same type.

<assign_stmt>

Non-terminal describes single assignment statement. An assignment statement is done by a variable is connected to an expression by assignment operator(=).

<var_id>

Non-terminal contains all identifiers such as variable names, function names, and paramater names.

<cond_stmt>

Non-terminal includes three types of conditional statements which are 1. only if condition and if block 2. both if and else condition and its blocks, 3. all conditions(if, else if and else) and their blocks. Blocks are separated by curly braces.

<loop_stmts>

Non-terminal describes loop structures of EMA language which are while loop and for-loop.

<while_loop>

Non-terminal describes the structure of a while-loop that consists of one or more conditions as expression <expr> and statements in the while loop as <stmts>. To determine the loop as a while loop, the reserved word “while” is used at the beginning of the declaration.

<for_loop>

Non-terminal describes the structure of for loops which can be done by two ways: 1. directly making an array by its name (ex: arr[]) 2. making an array with all boolean values in it. To determine the loop as a for loop, the reserved word “for” and “loop” is used in the declaration.

<exprs>

Non-terminal describes all expressions of EMA language.

<expr>

Non-terminal describes a single expression which can be one of five types: 1. a boolean constant 2. an operation expression 3. single variable name 4. an input statement 5. a function call.

<operator>

Non-terminal describes the logical operators of the EMA language which are and, or, not, implies, double implies, equal, not equal.

<type>

Non-terminal indicates the only variable type in EMA language, which is boolean.

<bool_const>

Non-terminal describes the two values of boolean type variables, which are true and false.

<func_decl>

Non-terminal indicates the function declaration structure of EMA language. A function can be either void func (func is a reserved word to indicate a function) or bool func to separate both functions with returns(bool func) and non-returns(void).

<func_id>

Non-terminal describes the function name (identifier).

<func_type>

Non-terminal determines the type of a function: void or bool

<func_call>

Non-terminal describes function calls. A function call can be done in two ways: function with no arguments or function with arguments.

<return_stmt>

Non-terminal describes a single return statement which can be done either in brackets or not. “return” reserved word indicates the statement is return statement.

<comment>

Non-terminal describes the structure of comments of EMA language. Comments can be done by string between 3 hashtags before and after (### … ###).

<chars>

Non-terminal describes the strings of EMA language.

<input_stmt>

Non-terminal describes the structure of input statements. Input statement is done by “in” reserved word and following string between brackets.

<output_stmt>

Non-terminal describes the structure of output statements. Input statement is done by “out” reserved word and following expressions between brackets.

<out_inside>

Non-terminal describes the insides of an output statement which can be 1. string 2. variable name 3. both variable name and a string separated by comma.

<array_decl>

Non-terminal describes the array declarations of EMA language. A declaration can be made in two ways: 1. type and name of the array and square brackets 2. type, name, square brackets and assignment of array elements.

<array_call>

Non-terminal indicates the array call structure of EMA language which can be done by two ways: 1.name of the array and square brackets(ex: arr[]) 2. boolean values in square brackets(ex: [true, false, true]).

<array_stc>

Non-terminal describes the array structure of language, which is boolean values separated by comma in brackets.

<bool_var>

Non-terminal describes boolean variables of language with recursion. A <bool_var> can be one bool constant or many bool constants separated by comma.

3) DESCRIPTIONS OF NONTRIVIAL TOKENS

Comments:

In most programming languages, comments are used to increase the readability of the codes. In some cases, programmers can add comments to explain the following code segment; in other cases, it can be used for giving specific messages to users. In the EMA language, users can create comments by using "###" for both the front and the end of the comment sentence. The motivation for building this comment system is that users can easily write their comments as both single-line and multiple-line as long as they add triple hashtags at both the front and end of their comments. An example is given below:

```
1    ### single-line comment example ###
2          •••
3    ### multiple-line
4    comment example ###
```

Identifiers:

Identifiers are the language elements to define the name of variables in most programming languages. EMA language also contains identifiers to define bool-type variables and function names. Developers can define identifiers by using single char, or a word, which contains letters or numbers. The following example shows the identifiers in the EMA:

```
1    bool ans1 = true ### ans1 is an identifier ###
2    void isEqual(bool a, bool b){•••} ### isEqual, a and b are
3    an identifiers ###
```

Literals:

The EMA language allows users to create only bool-type constants when they write code, and it represents whether the value is true or false. Bool data types are genenally beneficial for logical expressions, and the EMA language helps user to create these expressions. The following code lines show the example of bool-type literals in the EMA:

```
1    ### create a bool const ###
2    const bool a = true
3    a = false ### error !!! const values cannot be changed ###
```

Reserved Words:

Reserved words are identifiers or words which are used for specific purposes in language. They are already assigned by language, and users cannot use these words except for their assigned purpose. The following words are reserved words for the EMA language:

if:

It is used for regular if statements (`if (•••) { ••• }`)

else if:

It is used for regular else if statements (`else if (•••) { ••• }`)

else:

It is used for regular else statements (`else(•••) { ••• }`)

void:

When a void function is created, it is used to determine the return type of the function.

(`void func()`)

bool:

When a bool function is created, it is used to determine the return type of the function.

(`bool func(), bool a = true`)

func:

The special front name for all functions. (`type func( )`)

return:

The return token is used in functions to determine the function's return variable.

const:

It is used to determine variables when it should be unchangeable (`const bool a`)

for:

It is the first token to initialize the for loop (`for a loop array[] { ••• }`)

loop:

It is used as the token between single variable and array (`for a loop array[] { ••• }`)

while:

It is used for initializing while loop (`while ( statement ) { ••• }`)

in():

It is used to take input from the user (`in( "•••" )`)

out:

It is a function that shows the output to the console (`out( "•••" )`)

4) EVALUATION OF THE LANGUAGE

a) Readability:

Simplicity:

EMA language is designed with only boolean values, so its features are limited but very simple to understand and write. Since it can deposit more than one value, it helps decide situations. Parentheses, curly braces, or reserved words are also simplified to make language understandable for everyone.

Orthogonality:

Since EMA language contains only one primitive type, boolean, it is easy to write and read. Since no arithmetic operations exist, every program construction is straightforward and depends on the user's wants.

Data Types:

EMA language only contains a boolean variable.

Syntax Considerations:

EMA language allows different identifier combinations, helping users to adapt quickly. Reserved words and loops are not very strict as, well. What makes EMA different is not new words but keeping the simplicity of the language to be understandable by everyone.

b) Writiability:

Simplicity:

Ema language doesn’t have extra reserved words and has a similar syntax to well-known programming languages to make writing easier.

c) Reliability:

Readability and Writability:

Ema language is simple, so it is easy to read and write. Readability and writability make it easier to specify the bugs and the errors.

5) EXAMPLE CODES

Program 1:

```
1     ### example program 1 ###
2     bool x = in(“enter value: “)
3     bool y = in(“enter value: “)
4     bool z = in(“enter value: “)
5
6     while ( x == true && y == true && z == true ) {
7           out(“enter at least one false value!”)
8           x = in(“enter value: “)
9           y = in(“enter value: “)
10          z = in(“enter value: “)
11    }
12
13    bool ans1 = x ==> y
14    bool ans2 = !x <==> y
15    out(“x implies y: ”, ans1)
16    out(“not x doubly-implies y: ”, ans2)
```

Program 2:

```
1     ### example program 2 ###
2     bool func foo(bool p, bool q, bool  r) {
3           out(“Function name: foo”)
4           out(“p = ”, p)
5           out(“q = ”, q)
6           out(“r = ”, r)
7           if ( r == true ) {
8                 return ( p ==> ( q || r ) )
9           }
10          else {
11                return ( q <==> r )
12          }
13    }
14    bool a
15    bool b
16    bool c
17    for a loop [true, false] {
18          for b loop [true, false] {
19                c = foo(a, b, false)
20                out("a = ", a)
21                out(“b = ", b)
22                out("c = ", c)
23          }
24    }
```