PROJECT REPORT

On

Compiler for a Toy Language- ERPLAG

Department of Computer Science and Engineering

Compiler Design

(CSN 401)

Session: 2019-2020 (19202)

Submitted to:

Dr. Manish Kamboj

Submitted by:

Krittika Chhabra (16103035)

Shivam Sehgal (16103037)

INTRODUCTION:

Toy Language: Toy language refers to any computer programming language that is not considered to be suitable or capable for building general purpose and high-end software and applications. It can be any programming language that lacks the advanced features, capabilities, programming constructs and paradigms of high-level language.

Toy language was primarily created as a means of programming language research and education, proof of concept for a computer science or programming theory and to create a prototype for a new programming language. Typically, toy language has all the capabilities to perform simple to complex mathematical and programming computations. However, it has an incapability in terms of lesser or no library programs support, missing programming constructs such as pointers and arrays, which limits it in creating general-use programs and applications.

Pascal, Treelang and Logo are popular examples of toy language.

ERPLAG: The language ERPLAG (ERadicate PLAGiarism) is a strongly typed language with primitive data types as integer and floating point. It also supports two other data types: Boolean and arrays. The language supports arithmetic and Boolean expressions, simple and input/output statements, declarative statements and conditional and iterative statements. The language supports modular implementation of the functionalities. Functions can return multiple values. The function may or may not return a value as well. The scope of the variables is static and the variable is visible only in the block where it is declared.

(a detailed description of the language is attached as Appendix 1)

Compiler:

1. **Phases of a Compiler**
   1.1. Lexical Analysis:
   Lexical Analysis is the first phase when the compiler scans the source code. This process can be left to right, character by character, and group these characters into tokens.
   Here, the character stream from the source program is grouped in meaningful sequences by identifying the tokens. It makes the entry of the corresponding tickets into the symbol table and passes that token to the next phase.

   The primary functions of this phase are:
   - ✓ Identify the lexical units in a source code
   - ✓ Classify lexical units into classes like constants, reserved words, and enter them in different tables. It will Ignore comments in the source program
   - ✓ Identify token which is not a part of the language

Example:
x = y + 10;

Tokens:

X	*identifier*
=	*Assignment operator*
Y	*identifier*
+	*Addition operator*
10	*Number*

   1.2. Syntax analysis:

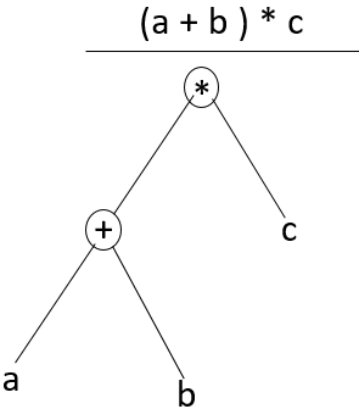
Syntax analysis is all about discovering structure in code. It determines whether or not a text follows the expected format. The main aim of this phase is to make sure that the source code was written by the programmer is correct or not.
Syntax analysis is based on the rules based on the specific programming language by constructing the parse tree with the help of tokens. It also determines the structure of source language and grammar or syntax of the language.

Here, is a list of tasks performed in this phase:

- ✓ Obtain tokens from the lexical analyser
- ✓ Checks if the expression is syntactically correct or not
- ✓ Report all syntax errors
- ✓ Construct a hierarchical structure which is known as a parse tree

Example:
Any identifier/number is an expression.
If x is an identifier and y+10 is an expression, then x= y+10 is a statement.
Consider parse tree for the following example:
*(a + b) * c;*
In Parse Tree:

- ✓ *Interior node: record with an operator filed and two files for children*
- ✓ *Leaf: records with 2/more fields; one for token and other information about the token*
- ✓ *Ensure that the components of the program fit together meaningfully*
- ✓ *Gathers type information and checks for type compatibility*
- ✓ *Checks operands are permitted by the source language*



1.3. Semantic analysis:

Semantic analysis checks the semantic consistency of the code. It uses the syntax tree of the previous phase along with the symbol table to verify that the given source code is semantically consistent. It also checks whether the code is conveying an appropriate meaning.
Semantic Analyzer will check for Type mismatches, incompatible operands, a function called with improper arguments, an undeclared variable, etc.
Functions of Semantic analyses phase are:

- ✓ Helps you to store type information gathered and save it in symbol table or syntax tree
- ✓ Allows you to perform type checking
- ✓ In the case of type mismatch, where there are no exact type correction rules which satisfy the desired operation a semantic error is shown
- ✓ Collects type information and checks for type compatibility
- ✓ Checks if the source language permits the operands or not

Example:
float x = 20.2;
*float y = x*30;*

In the above code, the semantic analyser will typecast the integer 30 to float 30.0 before multiplication

1.4. Intermediate code generator:

Once the semantic analysis phase is over the compiler, generates intermediate code for the target machine. It represents a program for some abstract machine.
Intermediate code is between the high-level and machine level language. This intermediate code needs to be generated in such a manner that makes it easy to translate it into the target machine code.

Functions on Intermediate Code generation:

- ✓ It should be generated from the semantic representation of the source program
- ✓ Holds the values computed during the process of translation
- ✓ Helps you to translate the intermediate code into target language
- ✓ Allows you to maintain precedence ordering of the source language
- ✓ It holds the correct number of operands of the instruction

Example:

```
total = count + rate * 5;
```

Intermediate code with the help of address code method is:

```
t1 := int_to_float(5)
t2 := rate * t1
t3 := count + t2
total := t3
```

1.5. Code optimizer:

The next phase of is code optimization or Intermediate code. This phase removes unnecessary code line and arranges the sequence of statements to speed up the execution of the program without wasting resources. The main goal of this phase is to improve on the intermediate code to generate a code that runs faster and occupies less space.

The primary functions of this phase are:

- ✓ Helps you to translate the intermediate
- ✓ It helps you to establish a trade-off between execution and compilation speed
- ✓ Improves the running time of the target program
- ✓ Generates streamlined code still in intermediate representation
- ✓ Removing unreachable code and getting rid of unused variables
- ✓ Removing statements which are not altered from the loop

Example:

Consider the following code

```
a = intofloat(10);
b = c * a;
d = e + b;
f = d;
```

Can become

```
b =c * 10.0;
f = e + b;
```

1.6. Code generator:

Code generation is the last and final phase of a compiler. It gets inputs from code optimization phases and produces the page code or object code as a result. The objective of this phase is to allocate storage and generate relocatable machine code.
It also allocates memory locations for the variable. The instructions in the intermediate code are converted into machine instructions. This phase coverts the optimize or intermediate code into the target language.

The target language is the machine code. Therefore, all the memory locations and registers are also selected and allotted during this phase. The code generated by this phase is executed to take inputs and generate expected outputs.

Example:
a = b + 60.0;
Would be possibly translated to registers.
MOVF a, R1
MULF #60.0, R2
ADDF R1, R2

Of the above phases, we have implemented the first two phases i.e. Lexical Analysis and Syntax Analysis.

STAGE 1- LEXICAL ANALYSIS

In this stage, the character stream from the source program is grouped in meaningful sequences by identifying the tokens. It makes the entry of the corresponding tickets into the symbol table and passes that token to the next phase.

The list of keywords identified as tokens in the project is as follows:

0. integer
1. real
2. boolean
3. of
4. array
5. start
6. end
7. declare
8. module
9. driver
10. program
11. get_value
12. print
13. use
14. with
15. parameters
16. true
17. false
18. takes
19. input
20. returns
21. AND
22. OR
23. for
24. in
25. switch
26. case
27. break
28. default
29. while

Further, this phase runs in 2 steps:

1. Removal of Comments: The comments start and end with '**'. All statements inside a pair of '**' are treated as comments and are ignored.
2. Each token is identified as either one of the above keywords or a variable name. The tokens can be of types: integer, floating point, arithmetic operator, newline character, ID (mostly var names) or keywords. Further, these tokens are sent to a DFA to check their validity. A dollar is then placed at the end of the file.

   The 'Lexeme' Table contains the final list of tokens.

Files used:

This phase uses the following .c files:

1. Driver.c – this contains the main menu and is the file from where the flow is maintained.
2. Hash.c – this contains the list of keywords in hash map form.
3. Lexer.c – this is the file where the main code involving removal of comments and tokenizing is present.

Code Output:

The make file contains instructions to start the project:

```
shivamsehgal@shivamsehgal:~/Compiler-ToyERPLAG$ make
gcc hash.c -c -o hash.o
gcc lexer.c -c -o lexer.o
gcc parser.c -c -o parser.o
gcc driver.c -c -o driver.o
gcc -o stage1exe hash.o lexer.o driver.o parser.o
shivamsehgal@shivamsehgal:~/Compiler-ToyERPLAG$ ls
```

Running the executable ‘stage1exe’ (here ‘testcase1.txt’ contains the code to be compiled and final output is stored in ‘output.txt’):

```
shivamsehgal@shivamsehgal:~/Compiler-ToyERPLAG$ ./stage1exe testcase1.txt output.txt

ERPLAG COMPILER MENU OPTIONS:
1: For producing clean code by removal of comments.
2: For printing the token list generated by the lexer.
3: For parsing to verify the syntactic correctness of the input source code
4: For creating the parsetree and printing it(inorder traversal)
5: To exit.
```

Running option 1- Removal of comments: After removal of comments, a file named ‘cleancode.txt’ is created which contains the code without any comments.

```
shivamsehgal@shivamsehgal:~/Compiler-ToyERPLAG$ ./stage1exe testcase1.txt output.txt

ERPLAG COMPILER MENU OPTIONS:
1: For producing clean code by removal of comments.
2: For printing the token list generated by the lexer.
3: For parsing to verify the syntactic correctness of the input source code
4: For creating the parsetree and printing it(inorder traversal)
5: To exit.
1

************************************************************************************
cleancode.txt created

************************************************************************************
```

Running option 2: Printing the token list generated by the lexer:

```
ERPLAG COMPILER MENU OPTIONS:
1: For producing clean code by removal of comments.
2: For printing the token list generated by the lexer.
3: For parsing to verify the syntactic correctness of the input source code
4: For creating the parsetree and printing it(inorder traversal)
5: To exit.
2

************************************************************************************
Line Number : 1, Lexeme : DRIVERDEF - <<<
Line Number : 1, Lexeme : KEYWORD - DRIVER
Line Number : 1, Lexeme : KEYWORD - PROGRAM
Line Number : 1, Lexeme : DRIVERENDDEF - >>>
Line Number : 2, Lexeme : KEYWORD - START
Line Number : 4, Lexeme : KEYWORD - END
Line Number : 5, Lexeme : DEF - <<
Line Number : 5, Lexeme : KEYWORD - MODULE
Line Number : 5, Lexeme : ID - empty
Line Number : 5, Lexeme : ENDDEF - >>
Line Number : 6, Lexeme : KEYWORD - TAKES
Line Number : 6, Lexeme : KEYWORD - INPUT
Line Number : 6, Lexeme : SQBO - [
Line Number : 6, Lexeme : ID - a
Line Number : 6, Lexeme : COLON - :
Line Number : 6, Lexeme : KEYWORD - BOOLEAN
Line Number : 6, Lexeme : SQBC - ]
Line Number : 6, Lexeme : SEMICOL - ;
Line Number : 7, Lexeme : KEYWORD - START
Line Number : 8, Lexeme : KEYWORD - END
Line Number : 9, Lexeme : $

************************************************************************************
```

STAGE 2- SYNTAX ANALYSIS

For syntax analysis, we need the first and follow sets. They are as given below:

NON-TERMINALS	FIRST	FOLLOW
<program>	DECLARE, DEF, DRIVERDEF	$
<moduleDeclarations>	DECLARE, ε	DEF, DRIVERDEF
<moduleDeclaration>	DECLARE	DEF, DRIVERDEF, DECLARE
<otherModule>	DEF, ε	DEF, $
<module>	DEF	DEF, DRIVERDEF, $
<driverModule>	DRIVERDEF	DEF, $
<ret>	RETURNS, ε	START
<input_plist>	ID	SQBC
<input_plistRec>	COMMA, ε	SQBC
<output_plist>	ID	SQBC
<output_plistRec>	COMMA, ε	SQBC
<type>	INTEGER, REAL, BOOLEAN	SQBC, COMMA, SEMICOL
<dataType>	INTEGER, REAL, BOOLEAN, ARRAY	COMMA, SQBC, SEMICOL
<moduleDef>	START	DEF, DRIVERDEF, $
<statements>	DECLARE, PRINT, USE, FOR, GET_VALUE, SWITCH, WHILE, ID, SEMICOL, SQBO, ε	BREAK, END
<statement>	DECLARE, PRINT, USE, FOR, GET_VALUE, SWITCH, WHILE, ID, SEMICOL, SQBO	DECLARE, PRINT, USE, FOR, END, GET_VALUE, SWITCH, BREAK, WHILE, ID, SEMICOL, SQBO
<ioStmt>	GET_VALUE, PRINT	DECLARE, PRINT, USE, FOR, END, GET_VALUE, SWITCH, BREAK, WHILE, ID, SEMICOL, SQBO
<whichId>	SQBO, ε	AND, OR, PLUS, MINUS, MUL, DIV, LT, LE, GT, GE, NE, EQ, SEMICOL, ASSIGNOP, BC
<index>	NUM, ID	SQBC

<simpleStmt>	ID, USE, SQBO	DECLARE, PRINT, USE, FOR, END, GET_VALUE, SWITCH, BREAK, WHILE, ID, SEMICOL, SQBO
<assignmentStmt>	ID	DECLARE, PRINT, USE, FOR, END, GET_VALUE, SWITCH, BREAK, WHILE, ID, SEMICOL, SQBO
<moduleReuseStmt>	SQBO, USE	DECLARE, PRINT, USE, FOR, END, GET_VALUE, SWITCH, BREAK, WHILE, ID, SEMICOL, SQBO
<optional>	SQBO, ε	USE
<idList>	ID	SEMICOL, SQBC, COLON
<idListRec>	COMMA, ε	SEMICOL, SQBC, COLON
<arithOrBoolExpr>	TRUE, FALSE, ID, NUM, RNUM, BO	SEMICOL, BC
<arithOrBoolExprRec>	AND, OR, ε	SEMICOL, BC
<anyTerm>	TRUE, FALSE, ID, NUM, RNUM, BO	AND, OR, SEMICOL, BC
<anyTermRec>	EMPTY, LT, LE, GT, GE, NE, EQ	AND, OR, SEMICOL, BC
<arithmeticExpr>	TRUE, FALSE, ID, NUM, RNUM, BO	AND, OR, LT, LE, GT, GE, NE, EQ, SEMICOL, BC
<arithmeticExprRec>	PLUS, MINUS, ε	AND, OR, LT, LE, GT, GE, NE, EQ, SEMICOL, BC
<term>	TRUE, FALSE, ID, NUM, RNUM, BO	AND, OR, PLUS, MINUS, LT, LE, GT, GE, NE, EQ, SEMICOL, BC
<termRec>	MUL, DIV, ε	AND, OR, PLUS, MINUS, LT, LE, GT, GE, NE, EQ, SEMICOL, BC
<factor>	TRUE, FALSE, ID, NUM, RNUM, BO	AND, OR, PLUS, MINUS, MUL, DIV, LT, LE, GT, GE, NE, EQ, SEMICOL, BC
<var>	TRUE, FALSE, ID, NUM, RNUM	AND, OR, PLUS, MINUS, MUL, DIV, LT, LE, GT, GE, NE, EQ, SEMICOL, BC
<pm>	PLUS, MINUS	TRUE, FALSE, ID, NUM, RNUM, BO

<md>	MUL, DIV	TRUE, FALSE, ID, NUM, RNUM, BO
<logicalOp>	AND, OR	TRUE, FALSE, ID, NUM, RNUM, BO
<relationalOp>	LT, LE, GT, GE, EQ, NE	TRUE, FALSE, ID, NUM, RNUM, BO
<declareStmt>	DECLARE	DECLARE, PRINT, USE, FOR, END, GET_VALUE, SWITCH, BREAK, WHILE, ID, SEMICOL, SQBO
<conditionalStmt>	SWITCH	DECLARE, PRINT, USE, FOR, END, GET_VALUE, SWITCH, BREAK, WHILE, ID, SEMICOL, SQBO
<caseStmts>	CASE	DEFAULT, END
<caseStmtsRec>	CASE, ε	DEFAULT, END
<value>	NUM, TRUE, FALSE	COLON
<default>	DEFAULT, ε	END
<iterativeStmt>	FOR, WHILE	DECLARE, PRINT, USE, FOR, END, GET_VALUE, SWITCH, BREAK, WHILE, ID, SEMICOL, SQBO
<range>	NUM	BC, SQBC

The code uses Grammer.txt as the input file for grammar and then constructs a grammar and consequently a parse table from it. The first and follow sets are input from the grammar itself (the above table is just for our reference).

After the Grammar, First and Follow sets and the Parse Tree are ready, the compiler runs in a DFS fashion to check the syntax of the given code.

As an output, a parse tree is generated and stored in ‘output.txt’.

Code Output:

Running option 3- Parsing the file (while parsing, a file ‘clean.txt’ is created to free the code from comments. If we first execute option 1 and then option 3 then both the files ‘cleancode.txt’ (mentioned above in stage 1) and the file ‘clean.txt’ will be generated with the same output.):

```
ERPLAG COMPILER MENU OPTIONS:
1: For producing clean code by removal of comments.
2: For printing the token list generated by the lexer.
3: For parsing to verify the syntactic correctness of the input source code
4: For creating the parsetree and printing it(inorder traversal)
5: To exit.
3

********************************************************************************

Parsing successful

********************************************************************************
```

Running option 4- Creation of the parse tree (the parse tree is created and stored in ‘output.txt’):

```
ERPLAG COMPILER MENU OPTIONS:
1: For producing clean code by removal of comments.
2: For printing the token list generated by the lexer.
3: For parsing to verify the syntactic correctness of the input source code
4: For creating the parsetree and printing it(inorder traversal)
5: To exit.
4

********************************************************************************

Parsing successful
output.txt successfully created.


********************************************************************************
```

Appendix 1 - ERPLAG Specifications

The language ERPLAG is a strongly typed language with primitive data types as integer and floating point. It also supports two other data types: boolean and arrays. The language supports arithmetic and boolean expressions, simple and input/output statements, declarative statements and conditional and iterative statements. The language supports modular implementation of the functionalities. Functions can return multiple values. The function may or may not return a value as well. The scope of the variables is static and the variable is visible only in the block where it is declared. The language is designed for the course CS F363 and the name of the language is motivated by the drive to ERadicate PLAGiarism.

1. Lexical structure

1.1 Keywords and identifiers

The reserved keywords are **declare, driver, program, for, start, end, module, get_value, print, use, with, parameters, true, false, takes, input, returns, AND, OR, switch, case, break, default** and **while**. A complete list of tokens for the keywords is given in table 1. An identifier is a sequence of letters, digits and an underscore starting with a letter. Identifiers can be at most 8 characters long. The language is case sensitive. This means that a lexeme switch is a keyword, different from **SWITCH** (use of uppercase) or **Switch**(use of uppercase in S only) which should be tokenized as identifiers. The lexemes **Value** and **value** are different from each other and represent different variables. An identifier is tokenised as **ID**. Identifiers are separated from keywords through a white space.

1.2 White Spaces and comments

The white spaces are blanks, tabs and new line characters. These are used to separate the tokens. Any number of white spaces is ignored and need not be tokenized. Keywords and identifiers must be separated by a white space or any other token which is different from keyword and identifiers. For example, **valueabc** is a single identifier, while **value:integer** is a stream of lexemes **value, :,** and **integer** and are tokenized as ID, a COLON and INTEGER respectively. A comment starts with ** and ends with **.

1.3 Numbers

An integer is a sequence of digits. The numbers 234, 1,45, 90123 etc represent integers and are tokenised as **NUM**. The type of the integer numbers is **integer**. A floating point number can be either a sequence of digits followed by a decimal point, followed by the fraction part of it again as a continuous sequence of digits, for example, a number 23.89, 908.567 are valid floating point numbers. These numbers can also be represented in mantissa exponent form, for example, 123.2E+6, 124.2E-1, 124.2e-1 etc. E can be both in upper case and lower case. Signs are optional and if no sign is used with E, the exponent value should be assumed as positive. The floating point number will not start with a decimal point, for example .124E+2 is not valid while 1.24E+1 is a valid lexeme. A floating point number is of data type **real** and is tokenised as **RNUM**.

1.4 Operators

The arithmetic operations are recognized by the lexemes +, -, *, / in their usual meaning as plus, minus, multiplication and division respectively. The relational operators are simply <, <=, >, >=, == and !=, known in their usual meaning as **less than, less than or equal to, greater than, greater than or equal to, equal** to and **not equal to** respectively. The logical and and or operations are permissible and the lexemes **AND** and **OR** (only in uppercase letters) are valid to be recognized as these two operations.

A special operator .. (dot dot) represents the range in defining the array range. An array is declared as **declare C:array[1..10] of integer;** The pair of these dots is different from the dot appearing in the lexeme of floating point number.
The assignment operator is := and is tokenised as ASSIGNOP. The tokens corresponding to these are given in table 2.

2. Language Features

2.1. Data Types

ERPLAG supports four data types-integer, real, boolean and array. The supported primitive data types are two, integer and floating point numbers represented by the types **integer** and **real**. The language also supports a **boolean** data type. The variables of boolean data type can attain one of the two values **true** and **false**. A conditional expression which evaluates to true or false is also of boolean type. A constructed type is **array** defined over a range of indices. An array is of elements of any of the three types above supported by this language. For instance, the declaration **declare C:array[1..10] of real**; represents the identifier name C as an array of real number and is of size 10. The array elements can be accessed as C[1], C[2], C[3], ...till C[10]. A range always starts with a 1 and ends with a positive integer. The array index is always a positive integer or an identifier.

2.2.Expressions

The expressions are of two types: **arithmetic** and **boolean**. An arithmetic expression is a usual infix expression. The precedence of operators * and / is high over + and -, while * and / are of the same precedence and + and – are of the same precedence. As an example, an arithmetic expression 15+29-12*2 evaluates to 20. A parenthesis pair has the largest precedence. For instance 15 + (29-12)*2 is computed as 49.
A boolean expression is an expression obtained by the conjunction of the arithmetic expression through the relational operators (<,<=, >, >=, ==, or !=). Two or more boolean expressions when ANDed or ORed, through the operators AND and OR, also compute to the values true or false. Example: ((x>=10 AND y<0) OR x<10) evaluates to true for values of x and y as 6 and -10 respectively. The static values of true and false cannot be taken as 1 and 0 as is usually assumed with C programming language.

2.3.Statements

The language supports five types of statements, **declarative, simple, input/output statements, conditional and iterative statements.** Declarative statements declare variables (identifiers) of defined type . As the language is strongly typed, each variable must have a type associated with it. The expression comprising of a number of variables also has a type based on the context.
A declarative statement **declare a,b,c:integer;** declares the names (identifiers) **a,b and c** of type integer. A **simple** statement has following structure
<left value> := <right expression>
A left value can be a simple identifier or a constructed expression for accessing an array element say A[i] = x+y; assigns value of the right hand side expression to the ith element of the array A.
The input statement get_value(v); intends to read value from the keyboard and associate with the variable v. The statement print(v); intends to write the value of variable v on the monitor.
The **only conditional statement** supported by this language is the C-like **switch-case** statement. There is no statement of C-like if. The switch applies to both integers and boolean numbers, but is not applicable to real numbers. Any boolean expression consisting of integers or real numbers equates to boolean TRUE or FALSE. A C-like if statement is not supported by ERPLAG but can be constructed using the switch case statement. Consider a C-like if statement

```
if(boolean condition)
        statements S1;
```

```
else
        statements S2;
```

This if-statement can be equivalently coded in ERPLAG as follows:

```
declare flag:boolean;
flag = <boolean condition>;
switch(flag)
start
        case FALSE      :<statements S1>;
                        break;
        case TRUE       :<statements S2>;
                        break;
end
```

The switch statement with a boolean identifier must have cases with values TRUE and FALSE. There is no default value associated with a switch statement with a boolean identifier. While a switch statement with an integer value can have any number of case statements. A default statement must follow the case statements. The case statements are separated by a break.

The **iterative statements** supported are **for** and **while** loops. The **for** loop iterates over the range of positive integer values. The execution termination is controlled by the range itself. There is no other guard condition controlling the execution in the **for loop** construct. An example is as follows.

```
for( k in 2..8)
Start
        x=x+2*k;
end
```

The value of k is not required to be incremented. The purpose of range itself takes care of this. The above piece of ERPLAG code produces x (say for its initial value as 5) as 9,15, 23, 33, 45, 59, 75 across the iterations.

The **while** loop iterates over a code segment and is controlled by a guard condition.

2.4 Functions

A function is a modular implementation which allows **parameter passing only by value** during invocation. A sample function definition is as follows

```
<<module sum>>
takes input [a:integer, b:integer];
returns [x:integer, abc:real];
start
        <function body>
end
```

A function can return multiple values unlike its C counterpart. The above function is invoked as follows [refer test case 1]

```
[r,m] := use mod1 with parameters v, w;
```

The compiler always verifies the type of the input and output parameters when the function is invoked with that of the definition. The input parameters can be of type integer, real, boolean and array type. The output parameters cannot be of array type, but can be of integer, real and boolean.

The local variables have static scope. The definition of variables is valid only within the function where it is defined. A variable must be declared before its use and should not be defined multiple times.

Function invocation should follow function definition or must have function declaration preceding it (refer test case 3)

2.5 Scope of Variables

An identifier scope is within the START-END block and is not visible outside. The local variables and parameters in the function definition have scope within the definition.

2.6 Program Structure

A Program is implemented with all modules written in a single file. The language does not support multiple file implementations. A program must have a single driver module. Other modules (functions) may or may not be exist and their definitions may precede or succeed the driver module. A module declaration becomes essential at the beginning of the program when the module definition is after the invocation of it.

3. Sample ERPLAG programs

Test Case 1: Demonstrates the usage of function declaration statement, driver function definition, function invocation, variable declaration, input, output, expression and assignment statement. Notice that the module definition is written after its invocation and it is therefore essential to declare the module before its invocation.

```
declare module mod1;
<<driver Program>>
start
        declare v, w, r :integer;
        get_value(v);
        w:=22;
        declare m:real;
        [r,m] := use mod1 with parameters v, w;
        print(r);
        print(m);
end

<<module mod1>>
takes input [a:integer, b:integer];
returns [x:integer, abc:real];
start
        declare c:real;
        c:=10.4;
        x:=a+b-10;
        abc:=b/5+c;
end
```

Expected output of the above test source code written in ERPLAG is 31 and 14.8 for input v read as 19 through keyboard. The driver prints 31 and 14.8 for variables r and m respectively

Test Case 2: This test case demonstrates the usage of boolean data type and the switch case statement. Notice that 'boolean' is not an enumerated type or user defined data type as is in C-like languages. Boolean data type is a high level abstraction of two values TRUE and FALSE supported by ERPLAG.

```
<<driver program>>
start
        declare a,b:integer;
        declare c:boolean;
```

```
        a:=21;
        b:=23:
        c:=(b-a>3);
        switch(c)
        start
                case TRUE:      b:=100;
                                break;
                case FALSE:     b= -100;
                                break;
        end
end
```

The expected value of b is -100. The compiler verifies the types of expression (b-a>3) and identifier c. On finding both of boolean type proceeds further to produce the target code.

Test case 3: This test case demonstrates the identifier pattern with underscore, a function that does not return any value, usage of logical operator AND, declartion of variables anywhere before usage,

```
<<module mod1>>
takes input [index:integer, val_:integer];
** this function does not return any value**
start
        declare i_1: integer;
        i_1:= val_+index/4;
        print(i_1);
end

<<driver Program>>
start
        declare a,b, dummy:integer;
        a:=48;
        b:=10;
        dummy:=100;
        declare flag:boolean;
        flag:=(a>=30)AND(b<30);
        switch(flag)
        Start
                case FALSE :    print(100);
                                break;
                case TRUE :     use mod1 with parameters a, b;
                                break;
        end
End
```

Test Case 4: This test case demonstrates the usage of for loop

```
<<driver Program>>
start
        declare num, k:integer;
        num:=9;
        for( k in 2..8)
        start
                num:=(num – k)*(num-k);
                print(num);
        end
```

```
end
```
The above code computes num iteratively for value of k ranging from 2 to 8 with an increment of 1 always. It produces output as 49, 36, 25, 16,9, 4, 1 iteratively.

Test Case 5: This test case demonstrates the use of array variables.

```
<<driver Program>>
start
        declare num, k:integer;
        declare A:array [1..10] of integer;
        num:=5;
        for( k in 1..10)
        Start
                A[k]:=(num – k)*(num-k);
                print(A[k]);
        end
End
```

Test Case 6: This test case demonstrates use of array elements and use of for loop.

```
<<module arraysum>>
takes input[list:array[1..20] of real];
returns [sum:real];
start
        declare s: real;
        s := 0.0;
        declare index : integer;
        for (index in 1..20)
        start
                s := s + list[index];
        end
        sum := s;
end

<<driver Program>>
start
        declare num, k:integer;
        declare A:array [1..10] of integer;
        for( k in 1..10)
        start
                A[k]:=(num – k)*(num-k);
                print(A[k]);
        end
        [num]:=use module arraysum with parameters A;
        print(num);
end
```

4. TOKEN LIST

The lexemes with following patterns are tokenized with corresponding token names. Lexemes AND and OR are in upper case letter while all other lexemes are in lower case letters. Token names are represented in upper case letters.

Table 1: Keywords

Pattern	**Token**
integer	INTEGER
real	REAL
boolean	BOOLEAN
of	OF
array	ARRAY
start	START
end	END
declare	DECLARE
module	MODULE
driver	DRIVER
program	PROGRAM
get_value	GET_VALUE
print	PRINT
use	USE
with	WITH
parameters	PARAMETERS
true	TRUE
false	FALSE
takes	TAKES
input	INPUT
returns	RETURNS
AND	AND
OR	OR
for	FOR
in	IN
switch	SWITCH
case	CASE

break	BREAK
default	DEFAULT
while	WHILE

Table 2. Symbols

Pattern (Regular expressions)	**Token**
+	PLUS
-	MINUS
*	MUL
/	DIV
<	LT
<=	LE
>=	GE
>	GT
==	EQ
!=	NE
<<	DEF
>>	ENDDEF
:	COLON
..	RANGEOP
;	SEMICOL
,	COMMA
:=	ASSIGNOP
[	SQBO
]	SQBC
(	BO
)	BC
**	COMMENTMARK

5. Grammar : Start Symbol : <program>

<program> → <moduleDeclarations> <otherModules><driverModule><otherModules>

<moduleDeclarations> → <moduleDeclaration><moduleDeclarations> | **ε**
<moduleDeclaration> → **DECLARE MODULE ID SEMICOL**
<otherModules> → <module><otherModules>| **ε**
<driverModule> → **DEF DRIVER PROGRAM ENDDEF** <moduleDef>
<module> → **DEF MODULE ID ENDDEF TAKES INPUT SQBO** <input_plist> **SQBC SEMICOL**<ret><moduleDef>
<ret> → **RETURNS SQBO** <output_plist> **SQBC SEMICOL | ε**
<input_plist> → <input_plist> **COMMA ID COLON** <dataType> | **ID COLON** <dataType>
<output_plist> → <output_plist> **COMMA ID COLON** <type> | **ID COLON** <type>
<dataType> → **INTEGER | REAL | BOOLEAN | ARRAY SQBO** <range> **SQBC OF** <type>
<type> → **INTEGER | REAL | BOOLEAN**
<moduleDef> → **START** <statements> **END**
<statements> → <statement> <statements> | **ε**
<statement> → <ioStmt>|<simpleStmt>|<declareStmt>|<condionalStmt>|<iterativeStmt>
<ioStmt> → **GET_VALUE BO ID BC SEMICOL | PRINT BO** <var> **BC SEMICOL**
<var> → **ID** <whichId> **| NUM | RNUM**
<whichId> → **SQBO ID SQBC | ε**
<simpleStmt> → <assignmentStmt> | <moduleReuseStmt>
<assignmentStmt> → **ID** <whichStmt>
<whichStmt> → <lvalueIDStmt> | <lvalueARRStmt>
<lvalueIDStmt> → **ASSIGNOP** <expression> **SEMICOL**
<lvalueARRStmt> → **SQBO** <index> **SQBC ASSIGNOP** <expression> **SEMICOL**
<index> → **NUM | ID**
<moduleReuseStmt> → <optional> **USE MODULE ID WITH PARAMETERS** <idList>**SEMICOL**
<optional> → **SQBO** <idList> **SQBC ASSIGNOP | ε**
<idList> → <idList> **COMMA ID | ID**
<expression> → <arithmeticExpr> | <booleanExpr>
<arithmeticExpr> → <arithmeticExpr> <op> <term>
<arithmeticExpr> → <term>
<term> → <term> <op> <factor>
<term> → <factor>
<factor> → **BO** <arithmeticExpr> **BC**
<factor> → <var>
<op> → **PLUS | MINUS | MUL | DIV**
<booleanExpr> → <booleanExpr> <logicalOp> <booleanExpr>
<logicalOp> → **AND | OR**
<booleanExpr> → <arithmeticExpr> <relationalOp> <arithmeticExpr>
<booleanExpr> → **BO** <booleanExpr> **BC**
<relationalOp> → **LT | LE | GT | GE | EQ | NE**
<declareStmt> → **DECLARE** <idList> **COLON** <dataType> **SEMICOL**
<condionalStmt> → **SWITCH BO ID BC START** <caseStmt><default> **END**
<caseStmt> → **CASE** <value> **COLON** <statements> **BREAK SEMICOL** <caseStmt>
<value> → **NUM | TRUE | FALSE**
<default> → **DEFAULT COLON** <statements> **BREAK SEMICOL | ε**
<iterativeStmt> → **FOR BO ID IN** <range> **BC START** <statements> **END** |
WHILE BO <booleanExpr> **BC START** <statements> **END**
<range> → **NUM RANGEOP NUM**