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 **ER**adicate **PLAG**iarism.

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

<u>Test Case 2:</u> This teat case d*emonstrates 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.

<u>Test case 3</u> *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
```

<u>Test Case 4</u> *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.

<u>Test Case 5</u> *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
```

<u>Test Case 6</u>: *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**

NOTE: ***The above grammar gives an outline of the language described in this document, but it is not fully described. There are several rules which were left for you to resolve, modify and reconstruct according to the description of the language. You will be asked to work out many things and submit on paper the hand drawn DFA/NFA for the lexical analysis part and hand written modified grammar.***

More updates, test cases and errata will be regularly updated on the course website.

Vandana
February 04, 2017

---------------------------------------*END OF ERPLAG Specifications Document*----------------------------------