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. The language supports assignment statements, input/output statements, declarative, conditional, iterative, and function call statements. The language supports the 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. The language has been evolving to provide a minimal feature set for the project in Compiler Construction course.

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, takes, input, returns, switch, case, break, default*** and ***while, etc***. 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 lower or upper case letter in a-z. Regular expression for an identifier is [a-z|A-Z|_][a-z|A-Z|0-9|_]*. Identifiers can be at most 20 characters long. The language is case sensitive. This means that the lexeme ***switch*** is considered as a keyword which is 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**. Valid identifiers are computeSum, C, A_1, a2cD2___, abc_123, abc123, etc. The names 1abc, _Abc are not valid identifiers. Identifiers are separated from keywords through a white space.

1.2 White Spaces, separator 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 white space can be a blank character, a tab or a new line character. Therefore, ***value : integer*** is also the same sequence of tokens as described above. A white space between two sequences of characters as in ***val ue*** makes them different than ***value.*** These will be tokenized as ID ID and not as one token ID. A semicolon ; separates the statements from each other. A comment starts with ** and ends with **.

1.3 Numbers

An integer number 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, numbers 23.89, 908.567 and 25.0 are valid floating point numbers, but 26. is not because there is no digit after the decimal point. Also, the number 0.25 is valid but .25 is not. These numbers can also be represented in mantissa and 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 and can be applied on operands of integer and real number types. However, the array variables themselves cannot be added or subtracted while the elements of array can be used as operands of these operators (to be explained in detail later). 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 data type is a **single dimensional array,** defined over a range of indices. An array can be of elements of any of the above three types supported by this language. For instance, the declaration **declare** C**: array[5..10] of real;** represents the identifier name *C* as an array of real numbers and is of size 6 where the array elements are accessed using constructed names C[5], C[6], C[7], C[8], C[9] and C[10] . However, an array element formation cannot be recursive, e.g., *A[A[i]]* is invalid. A range can start with an integer number (e.g. 5) and end with another integer (e.g. 10). The language also supports negative integers as part of range description e.g. **declare** M**: array[-5..10] of real;** is an array of size 16 where elements are accessed as M[-5], M[-4], M[0], and so on. The array can also be declared as a dynamic array using variable identifiers as range values. For example: **declare C: array [a..b] of real;** or **declare B:array [5..-b] of real;** where a and b are of integer type. The *address computation and type checking* for elements of dynamic arrays are done at *run time*. This means that your compiler should generate the assembly language code for such dynamic checks which eventually are performed when the translated user code is executed. The array elements can be accessed using an integer constant or an integer variable e.g. M[-2], M[8], M[0], M[k], C[5], C[u], C[k], C[10] etc. The *bound checking* for C[k] is done at *run time*. While accessing an array element, its index can be an integer, an identifier or an expression. If there are three array variables P, Q and R, declared as **declare P, Q, R: array[3..9] of real;** , then operations such as P+Q, P-Q, P*Q and P/Q are not allowed. But, P[7]+Q[3], P[4]/3, Q[5]*P[a+b-7] are valid for array elements. However, an array variable (not its elements) can be assigned to another structurally equivalent array variable as in P := Q;. This makes P to refer to Q's elements and not its own. If a prior assignment of P to another variable R using R := P; was made before its reassignment to elements of Q, then all the elements of P could be accessed using R, else this will be treated as memory leak for an amount equal to the memory allocated to all elements of P. The language supports structural type equivalence for arrays and other permissible identifiers. More details on type checking rules will be posted later.

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. For 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. The operators are left associative. The language also supports unary plus and minus operators for arithmetic expressions.

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. The logical operators AND and OR are of the same precedence.

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 declaration statement, **declare a, b, c:integer;** declares the names (identifiers) **a, b and c** to be of type integer. A declaration statement can appear anywhere in the program and is valid for any use thereafter with in its scope. An identifier cannot be declared multiple times in the same scope. Also, an identifier must be declared before its use.

A **simple** statement has the 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 i^{th} element of the array A. The right hand side expression can be a function call and correspondingly the left hand side changes to a list of actual parameters as given below

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

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 input statement get_value can read only variable identifier but cannot read any static constant (integer, real or boolean) or any array element. e.g. get_value(A[i]) and get_value(5) are syntactically incorrect (error). The output statement print can print variable identifier, a static constant (integer, real, boolean), an array element, and even a complete array as well. e.g. print(A), print(true), print(false), print(5), print(56.34), print(A[4]), print(A[k]) etc. are syntactically correct.

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. A switch statement with an identifier of type real is not valid and an error should be reported. A switch statement with an integer type identifier associated with it, can have case statement with case keyword followed by an integer only and the case statements must be followed by a default statement.

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 true       :<statements S1>;
                         break;
        case false      :<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. A switch statement with an integer value can have any number of case statements and a default statement must follow these case statements. The case statements are separated by a break. A switch statement must have at least one case statement.

The **iterative statements** supported are of two types- counter controlled **for** and logic controlled **while** loops. The **for** loop iterates over the range of 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..3)
start
        x=x+2*k;
end
```

The loop variable is implicitly of type integer upon entry to the loop and implicitly undeclared after loop termination. The value of k is incremented or decremented by step size of 1 implicitly depending upon the range and is not required to be changed in the body of the loop by the user. The range itself takes care of this through your compiler implementation. The above piece of ERPLAG code produces x (say for its initial value as 12) as 8, 6, 6, 8, 12, 18 across the iterations. A for statement must not redefine the variable that participates in the iterating over the range.

The **while** loop iterates over a code segment and is controlled by a guard condition. Any one of the variables used in the boolean expression used as conditional check for the while loop must be assigned a value to ensure the change during the iteration. For example, the following while code is correct as the variable i (one of the two variables i and k) has been assigned a value through the assignment statement i:=i+1;

```
while(i<=k)
start
```

```
            get_value(tempvar);
            arr1[i]:=tempvar;
            i:=i+1;
    end
```

2.4. Functions

A function is a modular implementation which allows **parameter passing by value** for all variable types including an array variable during invocation. An input array variable imitates **pass by reference** mechanism by simply copying the address value of the array base. An array element is different than array variable itself in its type and behaviour. 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  module sum 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 list of parameters used in function definition. The input parameters can be of type integer, real, boolean and array type as described above. The output parameters cannot be of array type, but can be of integer, real and boolean type. An array type input parameter copies the array's base address in the activation record of the called function. Hence, any changes in array element values are done in the caller's memory. No separate keyword is required for the array parameter for imitating the pass-by reference mechanism. The formal parameters and local variables have static scope. The definition of variables is valid only within the function where it is defined unless another nested definition shadows it. Formal parameters are declared in the separate scope of the function from the function's local variables. Therefore, the local variables can shadow an input parameter. But, a user is not allowed to define a local variable with the same name as that of the output parameter. A variable must be declared before its use and should not be defined multiple times. An identifier used beyond its scope must be viewed as undefined.

A function declaration refers to the prototype which is mentioned earlier than its definition as is illustrated in test case 1. A function declaration statement is as follows

```
declare module mod1;
```

Function invocation should follow function definition or must have function declaration preceding it (refer test case 1). In this test case, a function declaration for a function being used (mod1) by another function (driver) precedes the definition of the driver function because the function definition of mod1 does not precede the definition of the driver function. If the function definition of mod1 precedes the function definition of the driver function which invokes mod1, then the function declaration of mod1 is redundant and is not valid.

The types and the number of parameters returned by a function must be the same as that of the parameters

used in invoking the function. The parameters being returned by a function must be assigned a value. If a parameter does not get a value assigned within the function definition, it should be reported as an error. The function that does not return any value, must be invoked appropriately. Function input parameters passed while invoking it should be of the same type as those used in the function definition. However, the language does not support recursion and function overloading.

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 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:=5;
        declare m:real;
        [r,m] := use  module 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 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, declaration 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  module 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, a, k:integer;
        num:=9;
        for( k in 2..8)
        start
                a:=(num – k)*(num-k);
                print(a);
        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 -7..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[4..20] of real];
returns [sum:real];
start
        declare s: real;
        s := 0.0;
        declare index : integer;
        for (index in 4..20)
        start
                s := s + list[index];
        end
        sum := s;
end
```

```
<<<driver Program>>>
start
        declare num, k:integer;
        declare A:array [4..10] of integer;
        for( k in 6..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 the following patterns are tokenized with corresponding token names. Lexemes AND and OR are in upper case letters 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
takes	TAKES
input	INPUT
returns	RETURNS
for	FOR
in	IN
switch	SWITCH
case	CASE
break	BREAK
default	DEFAULT
while	WHILE

Table 2. Regular expressions and tokens

RE	Token
[a-z|A-Z|_][a-z|A-Z|0-9|_]*	ID
[0-9][0-9]*	NUM
[0-9][0-9]*[.] [0-9][0-9]*	RNUM
[0-9][0-9]*[.] [0-9][0-9]*[E|e] [+|-|ε] [0-9][0-9]*	RNUM
AND	AND
OR	OR
true	TRUE
false	FALSE
+	PLUS
-	MINUS
*	MUL
/	DIV
<	LT
<=	LE
>=	GE
>	GT
==	EQ
!=	NE
<<	DEF
>>	ENDDEF
<<<	DRIVERDEF
>>>	DRIVERENDDEF
:	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> → **DRIVERDEF DRIVER PROGRAM DRIVERENDDEF** <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 in the first stage and semantic rules in the second stage.*

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

Vandana
February 2, 2023