CD LAB PROJECT SYNOPSIS

Compiler Design for ERPLAG Language

Team members :
Jenit Jain - Roll no. 09
Sahil Garg - Roll no. 08
Amrit Goyal - Roll no. 36

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.

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 **.

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.

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) 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.

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.

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.

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	DEAFULT
while	WHILE

Symbols:

Pattern	**Token (Regular expressions)**
+	PLUS
-	MINUS
*	MUL
/	DIV
<	LT
<=	LE
>=	GE
>	GT
==	EQ
!=	NE
<<	DEF

>>	ENDDEF
:	COLON
..	RANGOP
;	SEMICOL
,	COMMA
:=	ASSIGNOP
[	SQBO
]	SQBC
(	BO
)	BC
**	COMMENTMARK

Grammar :

<program> -> <moduleDeclarations> <otherModules><driverModule><otherModules>
<moduleDeclarations> ->à <moduleDeclaration><moduleDeclarations> | ε
<moduleDeclaration> ->à 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> | <term>
<term> -> <term> <op> <factor> | <factor>
<factor> -BO <arithmeticExpr> BC | <var>
<op> -> PLUS | MINUS | MUL | DIV
<booleanExpr> -> <booleanExpr> <logicalOp> <booleanExpr>
<logicalOp> -> AND | OR
<booleanExpr> -> <arithmeticExpr> <relationalOp> <arithmeticExpr> | 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> -> àNUM RANGEOP NUMà