Compiler Construction / Compiler Design Assignment Stage 2

Name and ID of Team Members:

Group 37 Team 2: Shubham Gupta 2012A7PS086P

Ravi Shankar Pandey 2012C6PS676P

Name of Programming Language :

C - -

I. The language supports the following set of features

1. Static Scope of variables
2. **Global scope of variable**
3. **Block Level Scoping**
4. Typed variables
5. Basic data types :- int, real, char
   Derived data types: - **bool, string, array and tuple** data types
6. Operations on strings- string size.
7. Iteration is supported
   Iterative statement- while
8. Branching is supported
   Branching statement- if, else, break, next
9. Function can return multiple values.
10. Arithmetic operators are: plus(+), minus(-), multiplication(*), division(/), modulus(%), **power(^^), size(#)**
11. Boolean values are true and false.
12. Relational operators are: <= (less than or greater than), < (strictly less than), > (strictly greater than), >= (greater than or equal to), != (not equal to), == (equal to).
13. Logical operators are: ‘and’(AND), ‘or’(OR) and ‘not’(NOT).
14. Comments - not executable single line
15. Compilation - single file

A semicolon is used as the statement separator and comma is used as separator inside the statement.

//: - symbol is used to start the comment. The end of line is the end of comment marker.

The white spaces and comments are non-executable and will be cleaned as a pre-processing step of the lexical analyser.

INDEX

II. Sample Code

```
// function demo takes integer a and b and returns int
func _demo (int a, int b) -> (int)                    //function definition
{
    int ans;                                          //declaration of integer
    ans=a+b;                                          //initialization of ans
    return (ans);                                     //return ans to calling function
}
__main__                                              //main function
{
        int a, b, c;                                  //declaration of integers a, b, c
        if (a<=b)                                     //condition statement
        {
                a=a+b;
        }
        else                                          //condition statement
        {
                a=a-b;
        }
        c = _demo(a,b);                               //statement to call function demo
        write(a);                                     //writing to standard output
}
```

III. Programming Constructs

1. Identifiers

Variable Identifiers:

- An identifier is a combination of alphanumeric characters.
- An identifier does not have an underscore in it.
- The maximum length of an identifier is 20.
- Valid variable identifiers are: [a-zA-Z][a-zA-Z0-9]*
    - computeSum, algorithm1, FIND2, algorithmNEW etc.
    - The identifier sum_calculate, kjkjkjkjkghghjghgghjgjhghjgjgj, 6abc7def8, etc are invalid.

2. Data Types

The language supports the following data types

Integer type: The keyword used for representing integer data type is **int**. Integer type is of the pattern [+|-]?[0-9][0-9]*

Real type: The keyword used for representing real data type is **real**. Only 4 decimal places are allowed. Real number has the pattern [+|-]?[0-9][0-9]*[.][0-9][0-9]* and is of type real. A number .123 is invalid. Precision is same as IEEE floating point precision.

String type: The keyword used for representing string data type is **string** and follows a pattern ["][a-zA-Z0-9]*["]. Valid string type data are "abcdef", "hello" etc. Strings are not readable at run time so the user is required to give every input of the string type in the code itself. The string values are available statically, therefore the size of the string is known at compile time. The upper limit on the size of the string is 20.

Char type: The keyword used for representing char data type is **char** and can contain all characters with ASCII value of (0-127). For Example 'a', '', '. ' .

Bool type: The keyword used for representing bool type is **bool** and contain one of two possible values (a) true and (b) false.

Array type: The array is declared by: datatype identifier (size).

The array is initialized by: identifier(size)=[list of values]

Example: int arr(2); arr(2)=[1,2];

Array of type tuple is not allowed.

Multidimensional array can be declared by: datatype identifier (size,size*). They need to be initialized element by element.

Tuple type: The keyword used for representing tuple type is **tuple** and contains list of different datatypes all identified by a different identifier.

The tuple is declared by: tuple (list of datatypes with identifiers) tuple_identifier;

Example: tuple (int a, char ch, string s) t;

The tuple is initialized by: tuple_identifier = |list of values|

Example: t = | 2,'a',"abcd" | ;

3. Expressions

(i) Arithmetic: Supports all expressions in usual infix notation with the precedence of parentheses pair over multiplication and division. While addition and subtraction operators are given less precedence with respect to *, / and %. The power operator (^^) is given the highest precedence.

(ii) Boolean: Boolean expressions are required for the conditional if-else statements. The relational operators are

- <= (less than or greater than)
- < (strictly less than)
- > (strictly greater than)
- >= (greater than or equal to)
- != (not equal to)
- == (equal to).

A relational operator can be applied only to values or variables or numbers of type integer and real. If the operands of relational operators are strings, then the expression is invalid (semantics!!). We do not use arithmetic expressions as arguments of Boolean expressions, nor do we have string variables used in the Boolean expressions. Valid Boolean expressions are a==b, a!=10 and so on. Invalid expressions are a+b > c, (a-b) <=10+a.

(iii) Logical: The logical AND and OR operators are 'and' and 'or' respectively. An example of logical expression is ((a<=b) and (b>=c)). The not operator is 'not'.

4. Operators

The Operators are as follows

(i) Plus (+): Plus operation is valid if both the operands are

- integers. Type of the expression b+c is integer
- real. Type of the expression b+c is real

(ii) Minus (-): Minus operation is valid if both the operands are

- integers. Type of the expression b - c is integer

- real. Type of the expression b- c is real

Minus operation is invalid if the operands are of string type.

(iii) Multiplication (*): Multiplication operation is valid if both the operands are

- integers. Then the type of the expression b * c is integer
- real. Then the type of the expression b*c is real

Multiplication operation is invalid if the operands are of string type. The multiplication of an integer by a real number is not allowed.

(iv) Division (/): Division operation is valid if both the operands are

- integers. Then the type of the expression b/c is integer
- real. Then the type of the expression b/c is real

Division operation is invalid if the operands are of string type or tuple type. The division of an integer by a real number/real number by an integer is not allowed.

(v) Power operator (^^): Power operation b^^c is valid if both the operands are

- integers. Then the type of the expression b^^c is integer

Or

- b is real and c is integer. Then the type of the expression b^^c is real

Power operation is invalid if the operands are of string type or tuple type.

(vi)Modulus operator (%): Modulus operation is valid only if both the operands are

- integers. Then the type of the expression b%c is integer.

Modulus operation is invalid if the operands are of string type or tuple type or real type. The modulus of an integer by a real number/real number by an integer is not allowed.

(vii) Size operator (#): This is applicable only for the string variables and retrieves the respective size of the variable. This is a unary operator.

(viii) logical and, or, not: The operators 'and' and 'or' are applied between two boolean expressions. Patterns of and, or and not are ‘and’, ‘or’ and ‘not’ respectively.

5. Functions

(a) Function Definition: Keyword **func** precedes the definition followed by the function name followed by a list of input parameters followed by **‘->’** followed by list of output

parameters. Only previously implemented functions can be used. **A Function identifier must start with an underscore(_)**.

```
func  funcid (var list) -> (return list)
```

Constraints:

- var list and return list cannot contain tuple or array.
- Every function must return atleast 1 value.
- Function overloading is not allowed.
- Nested functions are not allowed.

(b) Visibility of variables: Scoping is static. A variable is visible within the scope of the function definition.

(c) Main function: This is the only driver function and specifies the complete scope.

(d) Function Call: A function which is not defined yet cannot be called. Any function can call any other function except main. Example:

```
func _demo (int a, int b) -> (int)
{
    int ans;
    ans=a+b;
    return (ans);
}
__main__
{
        int a, b, c;
        if (a<=b)
        {
                a=a+b;
        }
        else
        {
                a=a-b;
        }
        c = _demo(a,b);
```

```
        write(a);
}
```

6. Statements:

The language supports following type of statements:

Assignment Statement: Examples are:

```
value = x + (y /10) *(5-z);

c = _demo(a,b);
```

Function call returning more than one value:

```
int a;

char ch;

(a, ch) = _demo1(c, d);   // demo1 returns int and char
```

Declaration Statement: Declaration statements can be anywhere in the code. Example:

```
string s;

int arr(2);

 int a;

real  v;

tuple (int a, char ch, string s) t;

bool b;
```

Conditional Statements: Conditional statements are: **if**, **if-else**.

Example:

```
if (a<=b)

 {

        a=a+b;

}

else{

        a=a-b;

}
```

Iterative Statements: Iterative statement is: **while**

Example:

```
while(a>b)
{
}
```

Input-Output Statements: : Statement for reading data in the variable x is **read(x)**, and that for printing the value of x on the screen is **write(x)**.

Example:

```
read(x);
write(x);
```

Function Call Statement: Example

```
c = _demo(a,b);
(a, ch) = _demo1(c, d);          // demo1 returns int and char
```

IV. Lexical Units

Table 1: Lexical Units

Pattern	Token	Purpose
'__main__'	TK_MAIN	Keyword main
'#'	TK_SIZE	Size Operator
'='	TK_ASSIGN	Assignment operator
'while'	TK_WHILE	Keyword while
'break'	TK_BRK	Keyword break
'next'	TK_NEXT	Keyword next
'return'	TK_RETURN	Keyword return
'if'	TK_IF	Keyword if
'else'	TK_ELSE	Keyword else
[‘true’ + ‘false’]	TK_BCONST	Boolean constant
any character with ASCII value of (0-127).	TK_CCONST	Character constant
[+|-]?[0-9][0-9]*	TK_NUM	Integer number
[+|-]?[0-9][0-9]*[.] [0-9][0-9]*	TK_RNUM	Real number
["][a-z A-Z 0-9]*["]	TK_STR	String value
':'	TK_COLON	Used to access tuple
'func'	TK_FUNC	Keyword func
[a-zA-Z][a-zA-Z0-9]*	TK_FUNCID	Function identifier
[a-zA-Z][a-zA-Z0-9]*	TK_ID	Identifier
'('	TK_LP	Left parenthesis
')'	TK_RP	Right parenthesis
‘[‘	TK_LS	Left square parenthesis
‘]’	TK_RS	Right square parenthesis
'->'	TK_OPRET	Operator to specify return data Types
'{'	TK_LC	Left curly parenthesis
'}'	TK_RC	Right curly parenthesis
','	TK_COMMA	Comma separator
'int'	TK_INT	Keyword int
'real'	TK_REAL	Keyword real
'char'	TK_CHAR	Keyword char
'bool'	TK_BOOL	Keyword bool
'string'	TK_STRING	Keyword string
'tuple'	TK_TUPLE	Keyword tuple
';'	TK_SEMICOLON	Semicolon separator
'+'	TK_PLUS	Addition operator
'-'	TK_MINUS	Subtraction operator
'*'	TK_MUL	Multiplication operator
'/'	TK_DIV	Division operator
'%'	TK_MOD	Modulus operator
'^^'	TK_POW	Power operator

'and'	TK_AND	Logical and
'or'	TK_OR	Logical or
'not'	TK_NOT	Logical not
'<'	TK_LT	Relational operator less than
'<='	TK_LE	Relational operator less than or equal to
'|'	TK_PIPE	Used in tuple initialization
'=='	TK_EQ	Relational operator equal to
'>'	TK_GT	Relational operator greater than
'>='	TK_GE	Relational operator greater than or equal to
'!='	TK_NE	Relational operator not equal to
'//'	TK_COMMENT	Comment Beginning
'write'	TK_WRITE	Keyword write
'read'	TK_READ	Keyword read

Table 2: Keywords

Token	**Purpose**
TK_MAIN	Identifier of the starting function
TK_WHILE	Iterative statement
TK_BRK	Used to break from a loop
TK_NEXT	Used to move to next iteration
TK_RETURN	Used to return from a function
TK_IF	Conditional statement
TK_ELSE	Conditional statement
TK_FUNC	Used in function declaration
TK_INT	Used in integer declaration
TK_REAL	Used in real declaration
TK_CHAR	Used in character declaration
TK_BOOL	Used in boolean declaration
TK_STRING	Used in string declaration
TK_TUPLE	Used in tuple declaration
TK_WRITE	Used in standard output
TK_READ	Used in standard input

Example:

List of tokens for the program:

Program	Token Streams

1. func _demo (int a, int b) -> (int)
2. {
3. int ans;
4. ans=a+b;
5. return (ans);
6. }
 	
7. __main__
8. {
9. int a, b, c;
10. if (a<=b)
11. {
12. a=a+b;
13. }
14. else
15. {
16. a=a-b;
17. }
18. c = _demo(a,b);
19. write(a);
20. }
	1. TK_FUNC, TK_FUNCID, TK_LP, TK_INT, TK_ID, TK_COMMA, TK_INT, TK_ID, TK_RP,TK_OPRET, TK_LP, TK_INT, TK_RP
	2. TK_LC
	3. TK_INT , TK_ID, TK_SEMICOLON
	4. TK_ID, TK_ASSIGN, TK_ID, TK_PLUS, TK_ID, TK_SEMICOLON
	5. TK_RETURN, TK_LP, TK_ID,TK_RP, TK_SEMICOLON
	6. TK_RC
	
	7. TK_MAIN
	8. TK_LC
	9. TK_INT, TK_ID, TK_COMMA, TK_ID, TK_COMMA, TK_ID, TK_SEMICOLON
	10.TK_IF, TK_LP, TK_ID, TK_LE, TK_ID, TK_RP
	11.TK_LC
	12.TK_ID, TK_ASSIGN, TK_ID, TK_PLUS, TK_ID, TK_SEMICOLON
	13.TK_RC
	14.TK_ELSE
	15.TK_LC
	16.TK_ID, TK_ASSIGN, TK_ID, TK_MINUS, TK_ID, TK_SEMICOLON
	17.TK_RC
	18.TK_ID, TK_ASSIGN, TK_FUNCID, TK_LP, TK_ID, TK_COMMA, TK_ID, TK_RP,TK_SEMICOLON
	19.TK_WRITE, TK_LP, TK_ID, TK_RP, TK_SEMICOLON
	20.TK_RC

V. Grammar for Programming Language, Named C - -

1. <module> -> <stmtVarFunctionDef> <main>
2. <stmtVarFunctionDef> -> <globalDeclarationStmt> <stmtVarFunctionDef>
3. <stmtVarFunctionDef> -> <functionDef> <stmtVarFunctionDef>
4. <stmtVarFunctionDef> -> EPSILON
5. <functionDef> -> <functionHeader> <block>
6. <functionHeader> -> TK_FUNC TK_FUNCID TK_LP <paramList> TK_RP TK_OPRET TK_LP <typeList> TK_RP
7. <paramList> -> <type> TK_ID <restParams>
8. <restParams> -> TK_COMMA <paramList>
9. <restParams> -> EPSILON
10. <typeList> -> <type> <restType>
11. <restType> -> TK_COMMA <typeList>
12. <restType> -> EPSILON
13. <type> -> TK_INT
14. <type> -> TK_REAL
15. <type> -> TK_CHAR
16. <type> -> TK_BOOL
17. <type> -> TK_STRING
18. <globalDeclarationStmt> -> <type> <varList> TK_SEMICOLON
19. <globalDeclarationStmt> -> TK_TUPLE <tuple> TK_ID TK_SEMICOLON
20. <varList> -> TK_ID <varListFollow>
21. <varListFollow> -> <array> <moreIds>
22. <varListFollow> -> <moreIds>
23. <moreIds> -> TK_COMMA <varList>
24. <moreIds> -> EPSILON
25. <array> -> TK_LP <constVars> <moreArr> TK_RP
26. <moreArr> -> TK_COMMA <constVars> <moreArr>
27. <moreArr> -> EPSILON
28. <tuple> -> TK_LP <typeList> TK_RP
29. <main> -> TK_MAIN <block>
30. <block> -> TK_LC <stmts> TK_RC
31. <stmts> -> <declarationStmt> <stmts>
32. <stmts> -> <assignmentStmt> <stmts>
33. <stmts> -> <iterationStmt> <stmts>
34. <stmts> -> <conditionalStmt> <stmts>
35. <stmts> -> <branchStmt> <stmts>
36. <stmts> -> <ioStmt> <stmts>
37. <stmts> -> <functionCallStmt> TK_SEMICOLON <stmts>
38. <stmts> -> EPSILON
39. <declarationStmt> -> <type> <varList> TK_SEMICOLON
40. <declarationStmt> -> TK_TUPLE <tuple> TK_ID TK_SEMICOLON
41. <assignmentStmt> -> <assignStmtSingle>
42. <assignmentStmt> -> <assignStmtList>

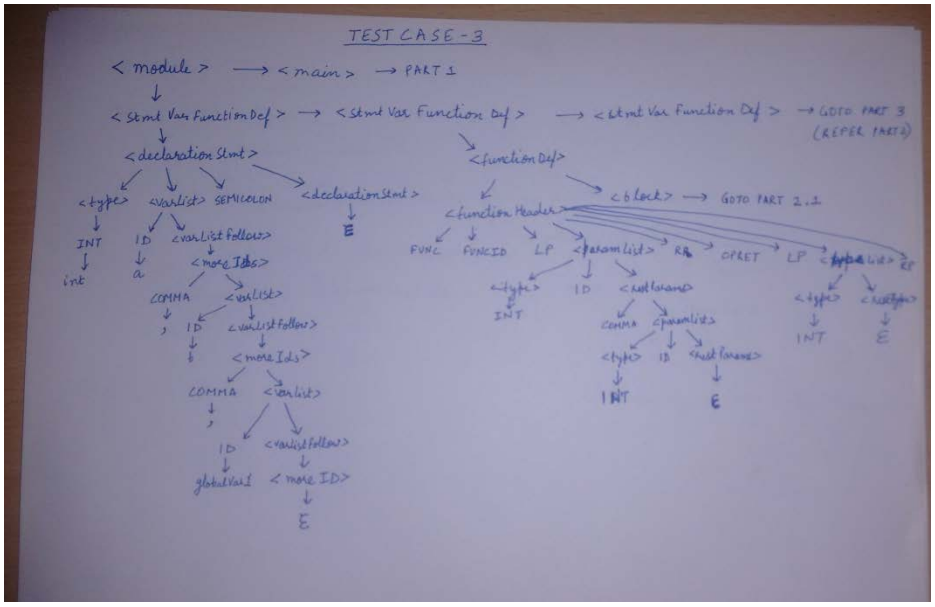
43. <assignStmtSingle> -> TK_ID <derivedTypes> TK_ASSIGN <assignStmtSingleFollow> TK_SEMICOLON
44. <assignStmtSingleFollow> -> <size>
45. <assignStmtSingleFollow> -> <arithExpr>
46. <assignStmtSingleFollow> -> <functionCallStmt>
47. <assignStmtSingleFollow> -> TK_PIPE <tupleInit> TK_PIPE
48. <assignStmtSingleFollow> -> <arrayInit>
49. <assignStmtList> -> TK_LP <varList> TK_RP TK_ASSIGN <assignStmtListFollow> TK_SEMICOLON
50. <assignStmtListFollow> -> <functionCallStmt>
51. <assignStmtListFollow> -> TK_PIPE <tupleInit> TK_PIPE
52. <arrayInit> -> TK_LS <constList> TK_RS
53. <tupleInit> -> TK_CCONST <otherInit>
54. <tupleInit> -> TK_NUM <otherInit>
55. <tupleInit> -> TK_RNUM <otherInit>
56. <tupleInit> -> TK_STR <otherInit>
57. <tupleInit> -> TK_BCONST <otherInit>
58. <otherInit> -> TK_COMMA <tupleInit>
59. <otherInit> -> EPSILON
60. <constList> -> <charList>
61. <constList> -> <intList>
62. <constList> -> <realList>
63. <constList> -> <boolList>
64. <constList> -> <strList>
65. <charList> -> TK_CCONST <moreChar>
66. <moreChar> -> TK_COMMA <charList>
67. <moreChar> -> EPSILON
68. <intList> -> TK_NUM <moreInt>
69. <moreInt> -> TK_COMMA <intList>
70. <moreInt> -> EPSILON
71. <realList> -> TK_RNUM <moreReal>
72. <moreReal> -> TK_COMMA <realList>
73. <moreReal> -> EPSILON
74. <boolList> -> TK_BCONST <moreBool>
75. <moreBool> -> TK_COMMA <boolList>
76. <moreBool> -> EPSILON
77. <strList> -> TK_STR <moreStr>
78. <moreStr> -> TK_COMMA <strList>
79. <moreStr> -> EPSILON
80. <size> -> TK_SIZE TK_ID
81. <derivedTypes> -> <arrayElement>
82. <derivedTypes> -> <stringElement>
83. <derivedTypes> -> <tupleElement>
84. <derivedTypes> -> EPSILON
85. <arrayElement> -> TK_LP <constVars> <moreDimension> TK_RP
86. <moreDimension> -> TK_COMMA <constVars> <moreDimension>

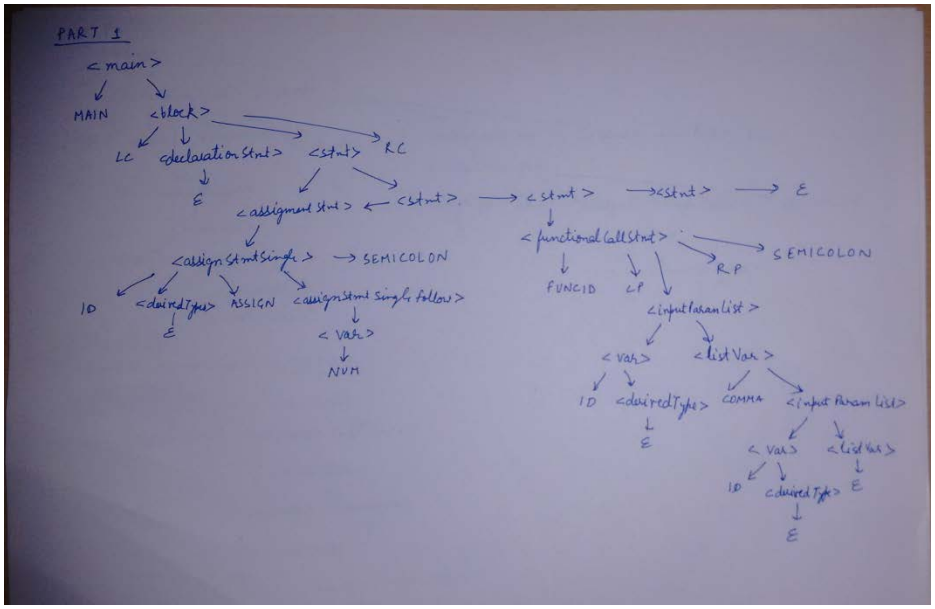
87. <moreDimension> -> EPSILON
88. <stringElement> -> TK_LS <constVars> TK_RS
89. <tupleElement> -> TK_COLON <constVars>
90. <arithExpr> -> <multExpr> <moreArith>
91. <moreArith> -> TK_PLUS <multExpr> <moreArith>
92. <moreArith> -> TK_MINUS <multExpr> <moreArith>
93. <moreArith> -> EPSILON
94. <multExpr> -> <powExpr> <moreMult>
95. <moreMult> -> TK_MUL <powExpr> <moreMult>
96. <moreMult> -> TK_DIV <powExpr> <moreMult>
97. <moreMult> -> TK_MOD <powExpr> <moreMult>
98. <moreMult> -> EPSILON
99. <powExpr> -> <primaryExpr> <morePow>
100. <morePow> -> TK_POW <primaryExpr> <morePow>
101. <morePow> -> EPSILON
102. <primaryExpr> -> <constVars>
103. <primaryExpr> -> TK_LP <arithExpr> TK_RP
104. <iterationStmt> -> TK_FOR TK_LP TK_ID TK_IN <range> TK_RP TK_LC <stmts> TK_RC
105. <iterationStmt> -> TK_WHILE TK_LP <boolExpr> TK_RP TK_LC <stmts> TK_RC
106. <range> -> TK_LP <constVars> TK_COMMA <constVars> TK_RP
107. <boolExpr> -> TK_LP <boolExpr> TK_RP <logicalOp> TK_LP <boolExpr> TK_RP
108. <boolExpr> -> <constVars> <boolExprFollow>
109. <boolExpr> -> TK_NOT TK_LP <boolExpr> TK_RP
110. <boolExprFollow> -> <relationalOp> <constVars>
111. <boolExprFollow> -> EPSILON
112. <constVars> -> TK_ID <derivedTypes>
113. <constVars> -> TK_CCONST
114. <constVars> -> TK_BCONST
115. <constVars> -> TK_NUM
116. <constVars> -> TK_RNUM
117. <constVars> -> TK_STR
118. <conditionalStmt> -> TK_IF TK_LP <boolExpr> TK_RP TK_LC <stmts> TK_RC <elseStmt>
119. <elseStmt> -> TK_ELSE TK_LC <stmts> TK_RC
120. <elseStmt> -> EPSILON
121. <branchStmt> -> TK_BRK TK_SEMICOLON
122. <branchStmt> -> TK_NEXT TK_SEMICOLON
123. <branchStmt> -> TK_RETURN TK_LP <retVars> TK_RP TK_SEMICOLON
124. <retVars> -> <constVars> <moreRet>
125. <moreRet> -> TK_COMMA <retVars>
126. <moreRet> -> EPSILON
127. <ioStmt> -> TK_READ TK_LP TK_ID TK_RP TK_SEMICOLON
128. <ioStmt> -> TK_WRITE TK_LP TK_ID TK_RP TK_SEMICOLON
129. <functionCallStmt> -> TK_FUNCID TK_LP <inputParamList> TK_RP
130. <inputParamList> -> <constVars> <listVar>
131. <inputParamList> -> EPSILON

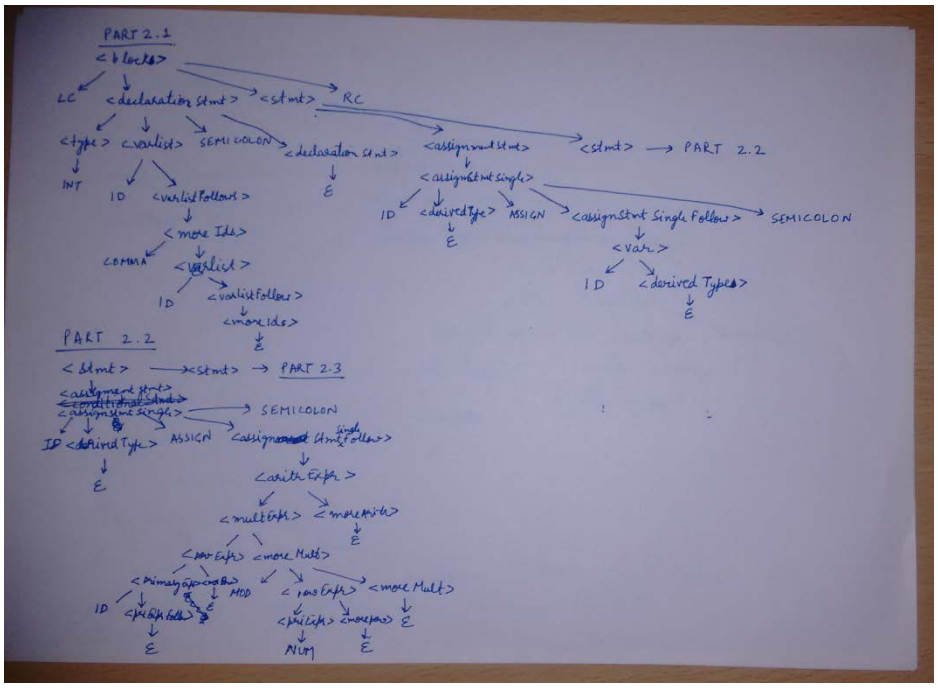
132. <listVar> -> TK_COMMA <inputParamList>
133. <listVar> -> EPSILON
134. <logicalOp> -> TK_AND
135. <logicalOp> -> TK_OR
136. <relationalOp> -> TK_LT
137. <relationalOp> -> TK_LE
138. <relationalOp> -> TK_EQ
139. <relationalOp> -> TK_GT
140. <relationalOp> -> TK_GE
141. <relationalOp> -> TK_NE

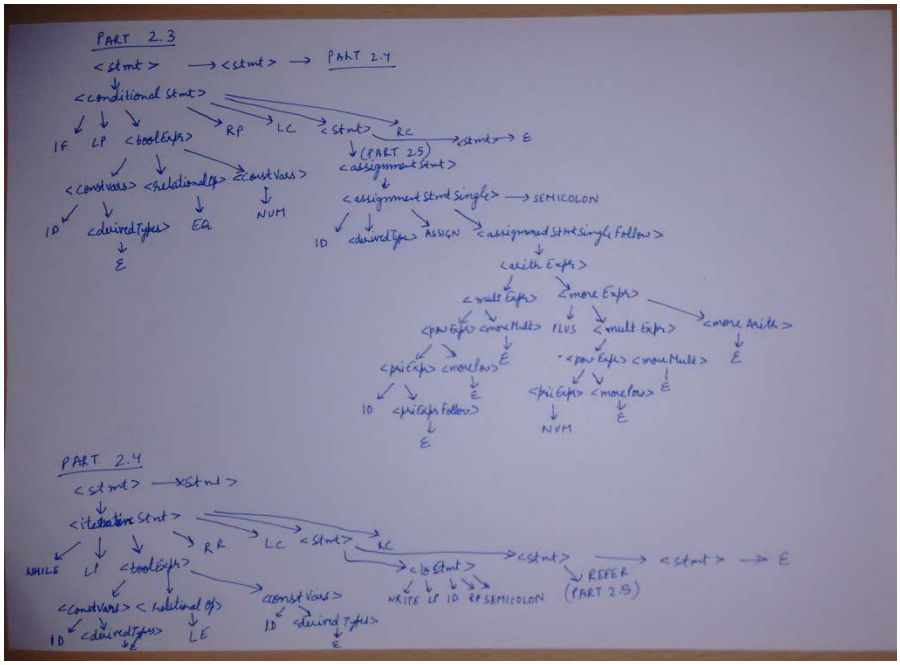
VI. Test Codes and Parse Tree for the above Grammar

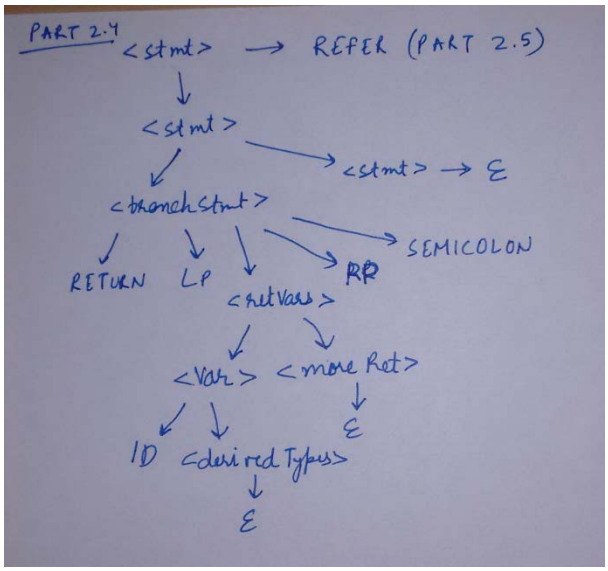
```
//Test Code 1
//prints all even integer between 40 and 5000
//global variables
int a, b, globalVar1;

//prints all the variables in range a and b
//returns number of lines printed
func _printinrange(int a, int b) -> (int) {
	int i,j;
	i = a;
	j = i%2;
	if(j==1) {
		i = i+1;
	}
	while( i <= b ) {
		write(i);
		i = i + 2;
	}
	i = b - a;
	return (b);
}

__main__ {
	a=40;
	b=500;
	_printinrange(a,b);
}
```




PART 1
<main>
MAIN
<block>
LC
<declaration Stmt>
ε
<stmt>
RC
<assignment Stmt>
<stmt>
<assignStmtSingle>
SEMICOLON
ID
<derivedType>
ASSIGN
<assignStmt Single follow>
ε
<Var>
NUM
<stmt>
<stmt>
ε
<functionalCallStmt>
SEMICOLON
FUNCID
LP
RP
<inputParamList>
<Var>
<listVar>
ID
<derivedType>
COMMA
<inputParamList>
ε
<Var>
<listVar>
ID
<derivedType>
ε
ε


PART 2.1
<block>
LC
<declaration Stmt>
<stmt>
RC
<type>
<varlist>
SEMICOLON
<declaration Stmt>
<assignment Stmt>
<stmt>
PART 2.2
INT
ID
<varlistFollows>
ε
<assignmentStmtSingle>
<moreIds>
ID
<derivedType>
ASSIGN
<assignStmt Single Follow>
SEMICOLON
ε
COMMA
<varlist>
<Var>
ID
<derived Type>
ID
<varlistFollows>
ε
<moreIds>
ε
PART 2.2
<stmt>
<stmt>
PART 2.3
SEMICOLON
ID
<derivedType>
ASSIGN
ε
<arithExpr>
<multExpr>
<moreAriths>
ε
<powExpr>
<moreMult>
ID
MOD
<powExpr>
<moreMult>
ε
ε
NUM
ε


PART 2.3
<stmt> → <stmt> → PART 2.4
<conditional Stmt>
IF LP <boolExpr> RP LC <stmt> RC <stmt> → ε
(PART 2.5)
<assignment Stmt>
<assignment Stmt Single> → SEMICOLON
ID <derivedTypes> ASSIGN <assignment Stmt Single Follow>
<arith Expr>
<mult Expr> <more Expr>
<more Arith>
PLUS
<more Mult>
<priExpr>
ID
NUM
ε
PART 2.4
<stmt> → <stmt>
<iteration Stmt>
WHILE LP <boolExpr> RP LC <stmt> RC
<io Stmt>
WRITE LP ID RP SEMICOLON
REFER (PART 2.5)
<constVars>
<relational Op>
LE
<derivedTypes>


PART 2.4
<stmt> → REFER (PART 2.5)
<stmt>
<stmt> → ε
<branch Stmt>
RETURN
LP
RP
SEMICOLON
<retVars>
<Var>
<more Ret>
ε
ID
<derived Types>
ε

//Test Code 2

```
__main__ {
        string str1, str2;
        int a,b,c,d,e,f;
        str1 = "intais";
        a = 10;
        b = 30;
        c = 3;
        d = (a^^c);
        e = b*d-a^^(c-1);
        c = e % 1000;
        str2 = "fifth character of variable str1 is ";
        write(str2);
        char q;
        q = str1(4);
        write(q);
        str2 = "variable str1 is ";
        write(str2);
        write(str1);
        str1 = "a to power c is ";
        write(str1);
        write(d);
}
```

//Test Code 3

```
//code for computing factorial of 12
__main__ {
        int b,fact;
        string str1;
        fact = 1;
        a = 12;
        while(not(a==0))
        {
                fact = fact*a;
                a = a-1;
        }
        str1 = "Factorial of 12 is ";
        write(str1);
        write(fact);
}
```

//Test Code 4

```
//demostrating tuple eand multiple return
//return a random record
func _getanotherrecord(int a) -> (int, string, real, char, bool) {
	bool b1;
	int b;
	b = a%2;
	if(b == 0) {
		b1 = true;
	}
	else {
		b1 = false;
	}
	return (a, "anotherRecord", 3.98, 'w', b1);
}

__main__ {
	tuple (int, string, real, char, bool) record1;
	string str1,str2;
	str2 = "printing record 1 ";
	record1 = |1, "ghost", 89.5, 8, true|;
	write(str2);
	int a;
	string b;
	a = record1:0;
	b = record1:1;
	write(a);
	write(b);
	str1 = "";
	record1 = _getanotherrecord(1);
	write(str1);
	write(str2);
	a = record1:0;
	b = record1:1;
	write(a);
	write(b);
}
```

//Test Code 5

```
//swap the given numbers
func _swapnumber(int a, int b) -> (int, int) {
	return (b,a);
}
//reverse a given array
__main__ {
	int list(10), i, j, k, l;
	list = [1,2,3,4,5,6,7,8,9,10];
	i = 9;
	j = 0;
	while (j<i) {
		k = j;
		while(k<i) {
			l = k + 1;
			int a,b;
			a = list(l);
			b = list(k);
			(a, b) = _swapnumber(a, b);
			k = k + 1;
		}
		j = j + 1;
	}
	//print new list
	j=0;
	while(j<i) {
		k = list(j);
		write(k);
		j = j + 1;
	}
}
```

VII. ERRATA

1. **Tuple Access:** Tuples can now be accessed with a help of a integer index which can be accessed by constants as well as integer variables.

2. **Ranged for loop:** Included range based for loops which will be given two integers as beginning and ending.

3. **Block Level Scoping:** As suggested during phase 1 review, added block level scoping.

4. **Removed redundancy in Grammar:** Many rules which were redundant were removed after making first sets and follow sets. (Attached in VIII heading)

VIII. REFERENCES

1. https://en.wikipedia.org/wiki/LL_parser

2. Engineering A Compiler – 2nd Edition by Keith Cooper

3. Compilers Principles, Techniques, and Tools – 2004 by Aho, Lam, Sethi and Ullman.

IX. APPENDIX

1. FIRST SET AND FOLLOW SET

Symbol	First set	Follow set
<module>	TK_FUNC, TK_INT, TK_REAL, TK_CHAR,TK_BOOL, TK_STRING, TK_TUPLE, TK_MAIN	$
<stmtVarFunctionDef>	TK_FUNC, TK_INT, TK_REAL, TK_CHAR,TK_BOOL, TK_STRING, TK_TUPLE	TK_MAIN
<functionDef>	TK_FUNC	TK_FUNC, TK_INT, TK_REAL, TK_CHAR,TK_BOOL, TK_STRING, TK_TUPLE, TK_MAIN
<functionHeader>	TK_FUNC	TK_LC
<paramList>	TK_INT, TK_REAL, TK_CHAR, TK_BOOL,TK_STRING	TK_RP
<restParams>	TK_COMMA	TK_RP
<typeList>	TK_INT, TK_REAL, TK_CHAR, TK_BOOL,TK_STRING	TK_RP
<restType>	TK_COMMA	TK_RP
<type>	TK_INT, TK_REAL, TK_CHAR, TK_BOOL,TK_STRING	TK_RP, TK_ID, TK_COMMA
<globalDeclarationStmt>	TK_INT, TK_REAL, TK_CHAR, TK_BOOL,TK_STRING, TK_TUPLE	TK_FUNC, TK_INT, TK_REAL, TK_CHAR,TK_BOOL, TK_STRING, TK_TUPLE, TK_MAIN
<varList>	TK_ID	TK_RP, TK_SEMICOLON
<varListFollow>	TK_LP, TK_COMMA	TK_RP, TK_SEMICOLON
<moreIds>	TK_COMMA	TK_RP, TK_SEMICOLON
<array>	TK_LP	TK_RP, TK_COMMA, TK_SEMICOLON
<moreArr>	TK_COMMA	TK_RP
<tuple>	TK_LP	TK_ID
<main>	TK_MAIN	$

<block>	TK_LC	TK_FUNC, TK_INT, TK_REAL, TK_CHAR,TK_BOOL, TK_STRING, TK_TUPLE, TK_MAIN, $
<stmts>	TK_FUNCID, TK_LP, TK_ID, TK_INT, TK_REAL,TK_CHAR, TK_BOOL, TK_STRING, TK_TUPLE,TK_FOR, TK_WHILE, TK_IF, TK_BRK, TK_NEXT,TK_RETURN, TK_READ, TK_WRITE	TK_RC
<declarationStmt>	TK_INT, TK_REAL, TK_CHAR, TK_BOOL,TK_STRING, TK_TUPLE	TK_FUNCID, TK_LP, TK_ID, TK_INT, TK_REAL,TK_CHAR, TK_BOOL, TK_STRING, TK_TUPLE,TK_RC, TK_FOR, TK_WHILE, TK_IF, TK_BRK,TK_NEXT, TK_RETURN, TK_READ, TK_WRITE
<assignmentStmt>	TK_LP, TK_ID	TK_FUNCID, TK_LP, TK_ID, TK_INT, TK_REAL,TK_CHAR, TK_BOOL, TK_STRING, TK_TUPLE,TK_RC, TK_FOR, TK_WHILE, TK_IF, TK_BRK,TK_NEXT, TK_RETURN, TK_READ, TK_WRITE
<assignStmtSingle>	TK_ID	TK_FUNCID, TK_LP, TK_ID, TK_INT, TK_REAL,TK_CHAR, TK_BOOL, TK_STRING, TK_TUPLE,TK_RC, TK_FOR, TK_WHILE, TK_IF, TK_BRK,TK_NEXT, TK_RETURN, TK_READ, TK_WRITE
<assignStmtSingleFollow>	TK_FUNCID, TK_LP, TK_ID, TK_PIPE, TK_LS,TK_CCONST, TK_NUM, TK_RNUM, TK_STR,TK_BCONST, TK_SIZE	TK_SEMICOLON

<assignStmtList>	TK_LP	TK_FUNCID, TK_LP, TK_ID, TK_INT, TK_REAL,TK_CHAR, TK_BOOL, TK_STRING, TK_TUPLE,TK_RC, TK_FOR, TK_WHILE, TK_IF, TK_BRK,TK_NEXT, TK_RETURN, TK_READ, TK_WRITE
<assignStmtListFollow>	TK_FUNCID, TK_PIPE	TK_SEMICOLON
<arrayInit>	TK_LS	TK_SEMICOLON
<tupleInit>	TK_CCONST, TK_NUM, TK_RNUM, TK_STR,TK_BCONST	TK_PIPE
<otherInit>	TK_COMMA	TK_PIPE
<constList>	TK_CCONST, TK_NUM, TK_RNUM, TK_STR,TK_BCONST	TK_RS
<charList>	TK_CCONST	TK_RS
<moreChar>	TK_COMMA	TK_RS
<intList>	TK_NUM	TK_RS
<moreInt>	TK_COMMA	TK_RS
<realList>	TK_RNUM	TK_RS
<moreReal>	TK_COMMA	TK_RS
<boolList>	TK_BCONST	TK_RS
<moreBool>	TK_COMMA	TK_RS
<strList>	TK_STR	TK_RS
<moreStr>	TK_COMMA	TK_RS
<size>	TK_SIZE	TK_SEMICOLON
<derivedTypes>	TK_LP, TK_LS, TK_COLON	TK_RP, TK_COMMA, TK_SEMICOLON, TK_ASSIGN,TK_RS, TK_PLUS, TK_MINUS, TK_MUL, TK_DIV,TK_MOD, TK_POW, TK_LT, TK_LE, TK_EQ, TK_GT,TK_GE, TK_NE

<arrayElement>	TK_LP	TK_RP, TK_COMMA, TK_SEMICOLON, TK_ASSIGN,TK_RS, TK_PLUS, TK_MINUS, TK_MUL, TK_DIV,TK_MOD, TK_POW, TK_LT, TK_LE, TK_EQ, TK_GT,TK_GE, TK_NE
<moreDimension>	TK_COMMA	TK_RP
<stringElement>	TK_LS	TK_RP, TK_COMMA, TK_SEMICOLON, TK_ASSIGN,TK_RS, TK_PLUS, TK_MINUS, TK_MUL, TK_DIV,TK_MOD, TK_POW, TK_LT, TK_LE, TK_EQ, TK_GT,TK_GE, TK_NE
<tupleElement>	TK_COLON	TK_RP, TK_COMMA, TK_SEMICOLON, TK_ASSIGN,TK_RS, TK_PLUS, TK_MINUS, TK_MUL, TK_DIV,TK_MOD, TK_POW, TK_LT, TK_LE, TK_EQ, TK_GT,TK_GE, TK_NE
<arithExpr>	TK_LP, TK_ID, TK_CCONST, TK_NUM, TK_RNUM,TK_STR, TK_BCONST	TK_RP, TK_SEMICOLON
<moreArith>	TK_PLUS, TK_MINUS	TK_RP, TK_SEMICOLON
<multExpr>	TK_LP, TK_ID, TK_CCONST, TK_NUM, TK_RNUM,TK_STR, TK_BCONST	TK_RP, TK_SEMICOLON, TK_PLUS, TK_MINUS
<moreMult>	TK_MUL, TK_DIV, TK_MOD	TK_RP, TK_SEMICOLON, TK_PLUS, TK_MINUS
<powExpr>	TK_LP, TK_ID, TK_CCONST, TK_NUM, TK_RNUM,TK_STR, TK_BCONST	TK_RP, TK_SEMICOLON, TK_PLUS, TK_MINUS,TK_MUL, TK_DIV, TK_MOD
<morePow>	TK_POW	TK_RP, TK_SEMICOLON, TK_PLUS, TK_MINUS,TK_MUL, TK_DIV, TK_MOD
<primaryExpr>	TK_LP, TK_ID, TK_CCONST, TK_NUM, TK_RNUM,TK_STR, TK_BCONST	TK_RP, TK_SEMICOLON, TK_PLUS, TK_MINUS,TK_MUL, TK_DIV, TK_MOD, TK_POW

<iterationStmt>	TK_FOR, TK_WHILE	TK_FUNCID, TK_LP, TK_ID, TK_INT, TK_REAL,TK_CHAR, TK_BOOL, TK_STRING, TK_TUPLE,TK_RC, TK_FOR, TK_WHILE, TK_IF, TK_BRK,TK_NEXT, TK_RETURN, TK_READ, TK_WRITE
<range>	TK_LP	TK_RP
<boolExpr>	TK_LP, TK_ID, TK_CCONST, TK_NUM, TK_RNUM,TK_STR, TK_BCONST, TK_NOT	TK_RP
<boolExprFollow>	TK_LT, TK_LE, TK_EQ, TK_GT, TK_GE, TK_NE	TK_RP
<constVars>	TK_ID, TK_CCONST, TK_NUM, TK_RNUM, TK_STR,TK_BCONST	TK_RP, TK_COMMA, TK_SEMICOLON, TK_ASSIGN,TK_RS, TK_PLUS, TK_MINUS, TK_MUL, TK_DIV,TK_MOD, TK_POW, TK_LT, TK_LE, TK_EQ, TK_GT,TK_GE, TK_NE
<conditionalStmt>	TK_IF	TK_FUNCID, TK_LP, TK_ID, TK_INT, TK_REAL,TK_CHAR, TK_BOOL, TK_STRING, TK_TUPLE,TK_RC, TK_FOR, TK_WHILE, TK_IF, TK_BRK,TK_NEXT, TK_RETURN, TK_READ, TK_WRITE
<elseStmt>	TK_ELSE	TK_FUNCID, TK_LP, TK_ID, TK_INT, TK_REAL,TK_CHAR, TK_BOOL, TK_STRING, TK_TUPLE,TK_RC, TK_FOR, TK_WHILE, TK_IF, TK_BRK,TK_NEXT, TK_RETURN, TK_READ, TK_WRITE

<branchStmt>	TK_BRK, TK_NEXT, TK_RETURN	TK_FUNCID, TK_LP, TK_ID, TK_INT, TK_REAL,TK_CHAR, TK_BOOL, TK_STRING, TK_TUPLE,TK_RC, TK_FOR, TK_WHILE, TK_IF, TK_BRK,TK_NEXT, TK_RETURN, TK_READ, TK_WRITE
<retVars>	TK_ID, TK_CCONST, TK_NUM, TK_RNUM, TK_STR,TK_BCONST	TK_RP
<moreRet>	TK_COMMA	TK_RP
<ioStmt>	TK_READ, TK_WRITE	TK_FUNCID, TK_LP, TK_ID, TK_INT, TK_REAL,TK_CHAR, TK_BOOL, TK_STRING, TK_TUPLE,TK_RC, TK_FOR, TK_WHILE, TK_IF, TK_BRK,TK_NEXT, TK_RETURN, TK_READ, TK_WRITE
<functionCallStmt>	TK_FUNCID	TK_SEMICOLON
<inputParamList>	TK_ID, TK_CCONST, TK_NUM, TK_RNUM, TK_STR,TK_BCONST	TK_RP
<listVar>	TK_COMMA	TK_RP
<logicalOp>	TK_AND, TK_OR	TK_LP
<relationalOp>	TK_LT, TK_LE, TK_EQ, TK_GT, TK_GE, TK_NE	TK_ID, TK_CCONST, TK_NUM, TK_RNUM, TK_STR,TK_BCONST