Language Specifications

The language supports following features

1. Expressions and statements
2. Nested Function Definitions
3. Static Scope of variables
4. Global scope of variables
5. Typed variables
6. Integer, real, strings and matrices data types
7. Operations on strings- string concatenation, string size etc.
8. Operations on matrices include- matrix addition, subtraction, size etc.
9. Comments - not executable
10. Compilation- single file
11. etc.

A semicolon is used as separator and a # 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 should be cleaned as a preprocessing step of the lexical analyzer.

The language deals with different lexical patterns, different for variable identifiers, function identifiers, integer and real numbers. We incorporate a very small number of features in this language to make it simpler for you to implement. The purpose of the compiler project is to make you learn the basic implementation of all modules. You gain the confidence of building a small compiler. The entire hard work of yours will be appreciable if you put constant efforts in learning and grow constantly.

The constructs of the language are described below.

1. Identifiers

Variable Identifiers: The identifiers are the names over the alphabet [a-z,A-Z,0-9]. It also takes uppercase letters. The variable identifier does not have an underscore in it. An identifier can have at most one digit at the end (optional). The maximum length of an identifier is 20. The regular expression for the variable identifiers is [a-z|A-Z][a-z|A-Z]*[0-9] | [a-z|A-Z][a-z|A-Z]* . Valid variable identifiers are: computeSum, algorithm1, FIND2, algorithmNEW etc. The identifier names _calculate, find_sum, new23, jehrjwhhkhjerthejrjtektrtjetjetet, new2abc3, etc.are invalid.

Function identifiers: The pattern includes names over alphabet [a-z] that start with an underscore. Upper case letters are also part of the pattern. Underscore or any other special symbol cannot occur anywhere in between the name of the function identifier. Any number of digits can be there in the function identifier. The regular expression for function identifiers is [_][a-z|A-Z][a-z|A-Z|0-9]*

2. Data Types

The language supports following data types

Integer type: The keyword used for representing integer data type is int and will be supported by the underlying architecture. A statically available number of the pattern [0-9][0-9]* is of integer type.

Real type: The keyword used for representing real data type is real and will be supported by the underlying architecture. A statically available real number has the pattern [0-9][0-9]* .[0-9][0-9] and is of type real. A number 23.67 is valid but 23.675 is not. Also .565 is invalid.

String type: The keyword used for representing string data type is string and follows a pattern ["][a-z][a-z]*["], no upper case letter is valid in the pattern. Valid string type data are "compilers", "absolute" 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. A string variable is initialized as follows:

```
s = "compilers";
```

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.

Matrix type: The keyword used for representing matrix data type is matrix which is a two dimensional array. The matrix element is of integer type and is accessed as M[i,j] where M is the name of the variable identifier, i is the row index and j is the column index. The size of matrix type variable cannot be more than 10 by 10. The matrix is laid out at compile time in column major format. The matrix data is provided in the code itself as this is not readable at run time. Therefore the user must type in all the matrices which are intended to be processed by the program. The language does not support loops, therefore adding the matrices element wise is not possible. But any element of the matrix is accessible through its row and column indices.
A matrix variable is initialized with values as follows

```
m = [2,3,4,5 ; 1,6,7,4 ; 10,2,3,4];
```

which is read row wise i.e. the matrix is

```
2   3   4   5
1   6   7   4
10  2   3   4
```

The construction of the right hand side of the matrix element values is understood as rows separated by semicolons and column values in each row are separated by commas.

An inconsistent right hand side of the assignment statement that initializes the matrix is known at the semantic analysis phase. The type expression of a matrix of size m by n is (Matrix, <m,n>). Since the values are statically available, the size of the matrix is known at compile time. Remember that string and matrix are not the user defined data types as they can be in C language. Here the data types are implemented in the language processor.

3. Expressions

(i) Arithmetic: The language 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 /. [You will have to modify the given grammar rules to impose precedence of operators]

(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) and == (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 or matrix, then the expression is invalid (semantics!!). In order to make the language simple, we do not use arithmetic expressions as arguments of boolean expressions, nor do we have string or matrix variables used in the boolean expressions. Valid boolean expressions are a<b, 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 (+):

Consider the expression b + c

Plus 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
- string. Then the type of the expression b+c is string . The plus sign semantically performs string concatenation by appending the string c at the end of string b.
- matrix. Then the type of the expression b+c is matrix. The sizes of b and c must be consistent.

(ii) Minus (-)

Consider the expression b - c
Minus 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
- matrix. Then the type of the expression b - c is matrix. The expression is semantically correct if the sizes of both matrices are same.

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 or are of matrix type. Also the multiplication of an integer by a real number is not allowed. Then the compiler is expected to report an error for an assignment statement a = 2*34.78;

(iv) Division (/)

Division operation is valid if both the operands are

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

Multiplication operation is invalid if the operands are of string type or are of matrix type.

(v) Size operator (@)

This is applicable only for the string and matrix type of variables and retrieves the respective size of the variable. This is a unary operator. The operation is valid if there is only one operand in the expression. Consider the expression @a, then the operation @ is valid if the type of a is

- string. Then the type of the expression @a is *integer*.
- matrix. Then the type of the expression @a is *integer x integer*.

Size operator cannot have an argument other than the ID. Examples @m, @s etc are valid. ([1,2;3,4;4,5] + m) is invalid. The size operator can only be applied to a variable name and not on values of string (i.e. actual string data) or on matrix element values. Only @str or @a etc are valid but @"abcd" and @[1,2,3;4,5,6] etc are not. The size operators can appear as the right hand side expressions in assignment statements.

```
a = @s;
[a,b] = @m;
```

(vi) logical and, or, not

The operators 'and' and 'or' are applied between two boolean expressions. Patterns of 'and', 'or' and 'not' are .and. (preceded and succeeded by dots), .or. and .not. respectively.

Example: a<b.and.(c<10.or.b=/=c)

5. Functions

(a) Function Definition: Keyword function precedes the definition and is followed by the list of output parameters. An equal sign is followed by the name of the function. The list of input parameters follows the name of the function.

```
function [string x, int m] = _operationDemoA[string u,string v, int a, int b]
#This is a sample function definition
        x = u + v;
        m = a + b;
end
```

Note that string is the inbuilt data type (and is unlike user defined data types as in C language) A value is not returned explicitly by a return statement but the semantics is such that the output parameters (e.g. x and m in the above function) are properly returned only if the variable appears at least once at the left hand side of the assignment statement.

```
function [matrix c] = _operationDemoB[matrix a, matrix b]
        c = a - b;
end
```

There is no return statement in the function definition. The semantic correctness of the function's output parameter is established through appropriate semantic rules. The semantic checks also include type checking, number of input parameters, number of output parameters and their types, whether the variable is returned properly or not. A function may not return any value.

(b) Function definitions are nested: Functions are defined within other functions.

```
function [int x] = _sumOfSquares[int a, int b, int c]
#computes the sum of squares of a, b and c
        int p,q,r;
        function [int s] = _square[int u]
                s = u*u;
        end
        p = _square(a);
        q = _square(b);
        r =  _square(c);
        x = p +q  + r ;
end
```

(c) Visibility of variables

A variable is visible within the scope of the function definition. In function _visibilityDemo(), variables m1, m2 are visible inside the function _concat().

```
function [string x] = _visibilityDemo[int a, int b]
        real c;
        string m1, m2;
        m1 = "computer";
        m2 = "programming";
        function [string s] = _concat[ ]
                s = m1+m2;
        end
        x = _concat( );
        c = a/b;
end
```

Note that the input parameters a and b contribute to the value of c which is not the output parameter, therefore value of c will not be returned to the calling function. At the same time the value of x is returned as "computerprogramming". The semantic analysis module has the responsibility of checking whether types of x and the expression with the function call at the right hand side match. If the type of the output parameter x of the function _visibilityDemo() is int, then the semantic analyzer must report the type mismatch error.

(d) Main function : This is the only driver function and specifies the complete scope.
All functions defined within it are considered visible. A nested function is visible only within the scope of its parent function that contains its definition. All variables declared in the main function are visible in all other function definitions (global scope).

```
 _main[ ]
        string g, h;
        g = "compiler";
        function [string x] = _visibilityDemo[int a, int b]
                real c;
                string m1, m2;
                m1 = "computer";
                m2 = "programming";
                function [string  s] = _concat[ ]
                        s = m1+m2;
                end
                x = _concat( );
                c = a/b;
        end
        h = g + _visibilityDemo(2,3);
        print(h);
end
```

Note that the operator '+' is valid over strings but '-' is not.

(e) Function Call: A function which is not defined yet cannot be called.

(f) Recursion: The language does not support recursion. Following is an example of semantically incorrect code.

```
function[....] = _g1[........]
        ......
        _ g1(...);

end
```

The function _g1() cannot be invoked within its definition.

(g) Function Block: A function block is identified by the keywords function-end pair.

(h) Function Overloading: The language does not support function overloading. Function overloading refers to redundant function definition in the same static scope. This means a function name can be used for different purpose in different scopes. Example :

```
function[.....] = _f1[.......]
        .......
        function[....] = _g1[........]
        ......
                function[....] = _h1[........]
                ......
                end
                #Following is an invalid definition with function name _h1
                #as more than one function with the same name is not permissible in the scope of _g1
                # This is not a syntax error but is a semantic error
                function[....] = _h1[........]
                ......
                end
```

```
end
function[....] = _g2[........]
......
        function[....] = _P1[........]
        ......
        end
        #Following is a valid definition with function name _h1
        # as there is no conflict within the scope of function _g2
        function[....] = _h1[........]
        ......
        end
end
```

6. Statements:

The language supports following type of statements:

Assignment Statement: Examples are as follows

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

Function call returning more than one value is a form of assignment statement with a list of variables on the left hand side

```
[a, b] =_demo(c,d);
```

Function call returning one value is a form of assignment statement with only one variable on the left hand side

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

Matrix elements can be accessed by respective indices and used in the expression on the right hand side

```
a = m[1,2]+m[1,3];
```

Matrix variables in the expression are also used directly. The type checking module verifies the type consistency and the compiler produces assembly language code with instructions for element wise addition of elements of matrix1 and matrix2 and storage in corresponding elements of c.

```
c = matrix1 + matrix2;
```

Assignment of values to variables is given below

```
a = 10;
b ="programming";
c = 12.34;
```

```
m = [2,3,4,5 ; 1,6,7,4 ; 10,2,3,4];
```

Expression on the right hand side can have numbers, string constants and matrix constants. Following are syntactically valid expressions in assignment statements. Their semantic correctness will be verified later and errors will be reported by the semantic analyzer appropriately.

```
s = "abc"+m;
m1 = [1,2;2,3;3,4] + m2;
s = "hello"+"world";
s = c + "hello";
```

An assignment statement does not have the matrix element (m[i,j]) at the left hand side of the assignment operator (...making language less complex for your project:)

Declaration Statement: Declaration statements can be anywhere in the code. Declaration statements, for example, are

```
string s1,s2;
matrix m;
int a,b;
real u, v;
```

Conditional Statements: Only one type of conditional statement is provided in this language. The 'if' conditional statement is of two forms; 'if-then' and 'if-then-else'. Example code is as follows

```
if  (i<10)
        m = n + o;
else
        m = n - o;
endif
```

else part may or may not be the part of the if statement. Each of the then and else parts must have at least one statement (semantics!).

Iterative Statements: Iterative statements are not supported in this languages.

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 print(x). If x is a simple integer or real number, the reading and printing are straight forward. When x is of string or matrix type, the semantic analyzer reports an error for read(x), but prints the values for print(x) appropriately for string and matrix. Print and read can have only an ID as the argument. print(x), read(x2) are valid. But print(5), print("abc"),etc are invlid. Also read(1) or read(m[i,j]) are invalid. Print and read do not operate on the list of identifiers. e.g. print (a,b) is invalid, while print(a); print(b) are valid.

Function Call Statement: Function Call Statements are used to invoke the function with the given actual input parameters. The returned values are copied in a list of variables as given below

```
[a, b] =_demo(c,d);
```

A function returning a single output value

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

The above two statements are of the form of assignment statements
A function that does not return any value is invoked as below

```
_demo(c,d);
```

The semantic analyzer verifies the type and the total number of output or input actual parameters matching with those used in function definition.

Miscellaneous:

Assignment statements of the following type must be handled properly. It is invalid syntactically.

```
[a,b] = 5;
```

The logical operator not is a unary operator and precedes the boolean expression

```
.not.(a<b)
```

Expressions have the maximum precedence of the parenthesis pair.
The keyword 'function' must be included in the language.
The language does not include typecasting. An expression with one type cannot be converted to another type.
At least one statement is a must in the function definition.
The open and closed parentheses pair around each boolean expression is required as per the syntax. So

```
(a<b).and.((c<10).or.(b=/=c))
```

should be correct.
Arithmetic operators are left associative.
Syntactically, all logical operators (and, or, not) must be applied to the boolean expressions and not on a number or arithmetic expressions.

Table 1: Lexical Units

Pattern	Token	Purpose
=	ASSIGNOP	Assignment operator
#	COMMENT	Comment Beginning
_[a-z|A-Z][a-z|A-Z|0-9] *	FUNID	Function identifier
[a-z|A-Z][a-z|A-Z]*[0-9] | [a-z|A-Z][a-z|A-Z]*	ID	Identifier (used as Variables)

[0-9][0-9]*	NUM	Integer number
[0-9][0-9]* .[0-9][0-9]	RNUM	Real number
["][a-z][a-z]*["]	STR	String Instance (value)
end	END	Keyword end
int	INT	Keyword int
real	REAL	Keyword real
string	STRING	Keyword string
matrix	MATRIX	Keyword matrix
_main	MAIN	Keyword main
[	SQO	Left square bracket
]	SQC	Right square bracket
(	OP	Open parenthesis
)	CL	Closed parenthesis
;	SEMICOLON	Semicolon as separator
,	COMMA	Comma
if	IF	Keyword if
else	ELSE	Keyword else
endif	ENDIF	Keyword endif
read	READ	Keyword read
print	PRINT	Keyword print
function	FUNCTION	Keyword function
+	PLUS	Addition operator
-	MINUS	Subtraction operator
*	MUL	Multiplication operator
/	DIV	Division operator
@	SIZE	Size Operator
.and.	AND	Logical and
.or.	OR	Logical or
.not.	NOT	Logical not
<	LT	Relational operator less than
<=	LE	Relational operator less than or equal to
==	EQ	Relational operator equal to
>	GT	Relational operator greater than
>=	LE	Relational operator greater than or equal to
=/=	NE	Relational operator not equal to

Tentative Outline of the Grammar:

The nonterminal <mainFunction> is the start symbol of the given grammar.

1. **<mainFunction>** ===> MAIN SQO SQC <stmtsAndFunctionDefs> END
2. <stmtsAndFuntionDefs>===> <stmtOrFunctionDef><stmtAndFunctionDefs>|<stmtOrFunctionDef>

3. <stmtOrFunctionDef>===> <stmt> | <functionDef>
4. <stmt>===> <declarationStmt> | <assignmentStmt> |<ifStmt>|<ioStmt>| <funCallStmt>
5. <functionDef>===>FUNCTION SQO <parameter_list> ASSIGNOP FUNID SQO <parameter_list> SQC <stmtsAndFunctionDefs> END
6. <parameter_list>===><type> ID <remainingList>
7. <type>===> INT | REAL | STRING | MATRIX
8. <remainingList>===>COMMA <parameter_list> | ∊
9. <declarationStmt>===> <type> <var_list> SEMICOLON
10. <assignmentStmt>===> <leftHandSide> ASSIGNOP <rightHandSide> SEMICOLON
11. <leftHandSide> ===> ID | SQO <var_list> SQC
12. <rightHandSide> ===> <arithmeticExpression> | <functionCall>
13. <conditionalStmt>===> IF <booleanExpression> <stmt><otherStmts> ELSE <otherStmts> ENDIF
14. <conditionalStmt>===> IF <booleanExpression> <stmt><otherStmts> ENDIF
15. <ioStmt>===> READ OP <var> CL SEMICOLON | PRINT OP <var> CL SEMICOLON
16. <arithmeticExpression>===><arithmeticExpression> <operator> <arithmeticExpression>
17. <arithmeticExpression> ====> OP <arithmeticExpression> CL | <var>
18. <operator> ===> PLUS | MUL | MINUS| DIV
19. <booleanExpression>===> OP <booleanExpression> CL <logicalOp> OP <booleanExpression> CL
20. <booleanExpression>===> <var> <relationalOp> <var>
21. <var>===> ID | NUM | RNUM | <matrixElement>
22. <matrixElement>===>ID SQO NUM COMMA NUM SQC
23. <logicalOp>===> AND | OR | NOT
24. <relationalOp>===> LT | LE | EQ | GT | GE | NE
25. <var_list>===> ID <more_ids>
26. <more_ids>===> COMMA <var_list> | ∊

***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. Problem description for both stage 1 and stage 2 will be uploaded separately. More updates, test cases and errata will be regularly updated on the course website.**

Vandana
January 21, 2018