Develop the lexical analyzer module of compiler for implementing the given language(given below in Language Specification)

Lexical Analyzer: This module takes as input the file containing user source code in the given language and produces the tokens. The lexical analyzer module scans the input only once and collects all relevant information required by the other modules of compiler. The lexical analyzer ignores comments and white spaces, while recognizes the useful lexemes as valid tokens The lexical errors are reported by this module when it sees any symbol or pattern not belonging to the language. Your lexical analyzer must

1. Tokenize lexemes appropriately
2. Maintain all information collected during a single pass of the source code
3. Be efficient with respect to time and space complexity
4. Report all lexical errors (with line number)

DO's and Don'ts

- *Read the language specifications document carefully.*
- *Understand patterns and tokens.*
- *Construct DFA based transition diagram (on paper) for recognizing the patterns in the source code.*
- *Design suitable and efficient data structure for representing and processing the language.*
- *Implement lexical analyzer first and test the tokens generated. Use a temporary driver to test lexical analyzer.*
- *Do not start coding randomly. First design the functionalities of your compiler front end and verify correctness of your DFA and grammar, then implement gradually your code.*
- *The Linux/ Ubuntu based GCC version will be specified for the C programming language you will use for your code within a few days. Do not use just any C compiler on your own. Eventually the versions of Linux/ Ubuntu and GCC will have to be used*
- *Revise your concepts regularly by reading your text book chapters.*
- *Feel free to fix up a time with me for a meeting in Contact hour session or any other convenient time whenever there is any doubt regarding design and implementation of the compiler.*

Language Specifications

The language is strongly typed and the primitive data types used are integers and real numbers. The language also supports record type and operations on records such as addition and subtraction can be applied for two operands of record type while scalar multiplication and division of record variables are also supported. Record type definitions are defined in any function but are available for any other function as well. The language supports modular code in terms of functions which uses call by value mechanism of parameter passing. The function may return many values of different types or may not return any value. The scope of the variables is local i.e. the variables are not visible outside the function where they are declared. The variables with prefix 'global' are visible outside the function and can be used within any function.

Sample code

```
% Program1.txt
_statistics
input parameter list [int c2dbc,int d7,int b2d567]
output parameter list [real d2, real c3bcd];
        type real: c3 : global;
        c3 <---3;
        d2 <--- (c2dbc + d7 + b2d567)/c3;
        c3bcd <--- d2*3.23;
        return [d2,c3bcd];
end
```

A semicolon is used as the separator and a % sign is used to start the comment. The white spaces and comments are non executable and should be cleaned as a preprocessing step of the lexical analyzer.
The function call is through the statements of following type

```
type real : c4;
type real : d3cd6 ;
[c4, d3cd6] <--- call _statistics with parameters [2,3,5] ;
```

If c4 was of type integer, then the semantic analyzer should have reported the type mismatch error for the use of c4 in the function call.

The infix expressions are used in assignment statements. The assignment operator is a bit unusual, a less than symbol followed by three continuous hyphen symbols.

The mathematical operations are many: addition, subtraction, multiplication and division which can be applied to both types of operands-integer and real, provided both the operands are of the same type. The operations + and – also add and subtract records, while multiplication and division can be used to perform scalar multiplication and scalar division of record variables.

The language deals with different lexical patterns, different for variable identifiers, function identifiers, record identifiers, record field identifiers, integer and real numbers. We incorporate a very small number of features in this language to make it simpler for you to implement. For example, we do not have a 'for' loop in our language. Also we are satisfied with a single conditional statement of if-then-else and if-then form, while we do not have switch case statements in our language.

The purpose of the lexical analyzer 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 program structure is modular such that all function definitions precede the main driver function. No function prototype declarations are required. Each function definition must have declaration statements first and the return statement only at the end. A return statement is a must for every function. All other statements such as assignment statements, conditional or iterative statements, input output statements etc. are placed before the return statement. A function can have within it a record definition and the same should be available globally.

The constructs of the language are described below.

Keywords

The language supports ***keywords*** ***while, return, main, if, type, read, write, call, input, output, parameter, list, record*** and so on. A list of all keywords is given towards the end of the document [Table 1].

Variable Identifiers

The identifiers are the names with the following pattern.

[b-d] [2-7][b-d] *[2-7] *

The identifier can be of any length of size varying in the range from 2 to 20.

A sample list of valid identifiers is d2bbbb54, b5cdbcdbcd7654, c6dcdcbcc7722.
The list of invalid identifiers is d2bdcbdcb5c, 2cdc765 and so on.
An identifier cannot be declared multiple times in the same scope and it should be declared before its use. Also, an identifier declared globally cannot be declared anywhere else in function definitions.

Function Identifiers

Function identifier name starts with an underscore and must have the following pattern

_[a-z|A-Z] [a-z|A-Z] *[0-9] *

i.e. a function name can have one or more number of English alphabet following the underscore. Also any number of digits can follow towards the trail. A function identifier is of maximum size of 30.

Data Types

The language supports the following 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 integer 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.

Record type: This is the constructed data type of the form of the Cartesian product of types of its constituent fields. For example the following record is defined to be of type 'finance' and its actual type is *int x real x int*

```
record  #finance
        type int: value;
        type real:rate;
        type int: interest;
endrecord
```

A record type must have *at least* two fields in it, while there can be any more fields as well.

The type information is fetched at the semantic analysis phase. A variable identifier of type finance is declared as follows

```
type record #finance : d5bb45;
```

The names of fields start with any alphabet and can have names as words of English alphabet (only small case). The fields are accessed using a dot in an expression as follows

```
d5bb45.value <--- 30;
d5bb45.rate     <--- 30.5;
```

and so on.

A test case handling addition operation on two records and use of record variables in parameters list is depicted below

```
_recordDemo1 input parameters [record #book d5cc34, record #book d2cd]
output parameters[record #book d3];
        d3<--- d5cc34 + d2cd;
        return [d3];
end
_main
        record  #book
                type int : edition;
```

```
            type real: price;
        endrecord;
        type record #book b2;
        type record #book c2;
        type record #book d2;
        b2.edition <--- 3;
        b2.price <--- 24.95;
        c2.edition <--- 2;
        c2.price <--- 98.80;
         [d2]<--- call _function1 with parameters [b2,c2];
        write(d2);
end
```

A variable of record type can only be multiplied or divided by a scalar (integer or real) i.e. two record type variables cannot be multiplied together nor can be divided by the other. Two operands (variables) of record type can be added, subtracted from one provided the types of the operands match and both the operands are of record type. Semantically an addition/subtraction means addition/subtraction of corresponding field values, for Example :

```
        type record #finance : d5;
        type record #finance : c4;
        type record #finance : c3;
        c3 <--- c4 + d5;
```

global: This defines the scope of the variable as global and the variable specified as global is visible anywhere in the code. The syntax for specifying a variable of any type to be global is as follows

```
        type int: c5d2: global;
```

Functions

There is a main function preceded by the keyword _main. The function definitions precede the function calls. Function names start with an underscore. For example

```
_function1
input parameters [int c2, int d2cd]
output parameters[int b5d, int d3];
        b5d<---c2+234-d2cd;
        d3<---b5d+20;
        return [b5d, d3];
end

_main
        type int: b4d333;
        type int : c3ddd34;
        type int:c2d3;
```

```
        type  int  c2d4;
        read(b4d333);
        read(c3ddd34);
        [c2d3, c2d4]<--- call _function1 with parameters [b4d333, c3ddd34];
        write(c2d3); write(c2d4);
end
```

The language does not support recursive function calls. Also, function overloading is not allowed in the language. Function's actual parameters types should match with those of formal parameters. Even if the type of a single actual parameter in a function call statement does not match with the type of the formal parameter in function definition, it is considered an error.

Statements:

The language supports following type of statements:

Assignment Statement: An expression to the right hand side assigned to an identifier is the form of these statements. Example

```
c2ddd2 <--- (4 + 3)*(d3bd -73);
```

Declaration Statement: Declaration statements precede any other statements and cannot be declared in between the function code. A declaration statement for example is

```
type int : b2cdb234;
```

Each variable is declared in a separate declaration (unlike C where a list of variables of similar type can be declared in one statement e.g. int a,b,c;)

Return Statement: A return statement is the last statement in any function definition. A function not returning any value simply causes the flow of execution control to return to the calling function using the following statement

```
return;
```

A function that returns the values; single or multiple, returns a list of in the following format

```
return [b5d, d3];
```

Iterative Statement: There is a single type of iterative statement. A while loop is designed for performing iterations. The example code is

```
while(c2d3 <=d2c3)
        c2d3 = c2d3+1;
        write (c2d3);
endwhile
```

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(c7>=d2dc)
then
        write(c7);
else
        write (d2dc);
endif
```

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

[c2d3, c2d4]<---call _function1 with parameters [b4d333, c3ddd34];

A function that does not return any value is invoked as below

call _function1 with parameters [b4d333, c3ddd34];

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

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 /. [You will have to modify the given grammar rules to impose precedence of operators]

(ii) Boolean: Conditional expressions control the flow of execution through the while loop. The logical AND and OR operators are &&& and @@@ respectively. An example conditional expression is (d3<=c5cd) &&& (b4>d2cd234). We do not use arithmetic expressions as arguments of boolean expressions, nor do we have record variables used in the boolean expressions.

Table 1: Lexical Units

Pattern	Token	Purpose
<---	TK_ASSIGNOP	Assignment operator
%	TK_COMMENT	Comment Beginning
[a-z][a-z] *	TK_FIELDID	Field name
[b-d] [2-7][b-d] *[2-7] *	TK_ID	Identifier (used as Variables)
[0-9][0-9]*	TK_NUM	Integer number
[0-9][0-9]* .[0-9][0-9]	TK_RNUM	Real number
_[a-z|A-Z][a-z|A-Z] *[0-9] *	TK_FUNID	Function identifier
#[a-z][a-z] *	TK_RECORDID	Identifier for the record type
with	TK_WITH	Keyword with
parameters	TK_PARAMETERS	Keyword parameters
end	TK_END	Keyword end
while	TK_WHILE	Keyword while
int	TK_INT	Keyword int
real	TK_REAL	Keyword real
type	TK_TYPE	Keyword type

_main	TK_MAIN	Keyword main
global	TK_GLOBAL	Keyword global
parameter	TK_PARAMETER	Keyword parameter
list	TK_LIST	Keyword list
[	TK_SQL	Left square bracket
]	TK_SQR	Right square bracket
input	TK_INPUT	Keyword input
output	TK_OUTPUT	Keyword output
int	TK_INT	Keyword int
real	TK_REAL	Keyword real
,	TK_COMMA	Comma
;	TK_SEM	Semicolon as separator
:	TK_COLON	Colon
.	TK_DOT	Used with record variable
endwhile	TK_ENDWHILE	Keyword endwhile
(	TK_OP	Open parenthesis
)	TK_CL	Closed parenthesis
if	TK_IF	Keyword if
then	TK_THEN	Keyword then
endif	TK_ENDIF	Keyword endif
read	TK_READ	Keyword read
write	TK_WRITE	Keyword write
return	TK_RETURN	Keyword return
+	TK_PLUS	Addition operator
-	TK_MINUS	Subtraction operator
*	TK_MUL	Multiplication operator
/	TK_DIV	Division operator
call	TK_CALL	Keyword call
record	TK_RECORD	Keyword record
endrecord	TK_ENDRECORD	Keyword endrecord
else	TK_ELSE	Keyword else
&&&	TK_AND	Logical and
@@@	TK_OR	Logical or
~	TK_NOT	Logical not
<	TK_LT	Relational operator less than
<=	TK_LE	Relational operator less than or equal to
==	TK_EQ	Relational operator equal to
>	TK_GT	Relational operator greater than
>=	TK_LE	Relational operator greater than or equal to
!=	TK_NE	Relational operator not equal to