Language Specifications

The purpose of the compiler project is to make you learn the basic implementation of some of the modules. You gain the confidence of building a small compiler. The two key modules that are expected to be developed in this project are Lexical analyzer and Syntax analyzer. Though, the language also support several other features like (type checking, scope information, etc.) that are not specifically required in the implementation of these two modules. However, these features and their equivalent constructs are just included in the language specifications so that they can be implemented to identify them as valid tokens and verify the structure of tokens using the grammar rules. You are not supposed to implement these things as type checker (semantic analysis) and code generation phase. The evaluation of these phases will be evaluated through assignments/Quiz.

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 language supports primitive data types that are used as integers and real numbers. The language also supports record type and union type. The operations on variables of record type are addition and subtraction. These operations can be applied for two operands of record type. The 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 function, 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 between two statements 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];

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. [You will be required to modify expression grammar for operations with records, but definitely you will be given the complete LL(1) grammar after making you go through the LL(1) compatibility checks, completeness of rules and overall modifications which you will be asked to submit within a week or so]

The program structure is modular such that all function definitions precede the main driver function. Function prototype declarations are not required. Each function definition must have declaration statements first and the return statement only at the end. A return statement is necessary for every function. All other statements such as assignment statements, conditional or iterative statements, input output statements etc. appear 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 as below.

Keywords

The language supports ***keywords*** ***while, return, main, if, type, read, write, call, input, output, parameter, parameters, list, record, union, definetype, as*** 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 type information is fetched at the semantic analysis phase hence, not required to implement in this course project. These related features of type information have just retained in the language specification to give you a feel of exhaustive language specifications so that their constructs and their structure can be handled for the implementation of lexical and syntax analysis phases.

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. The language also supports exponent and mantissa form of real number representation. The regular expression for the same is [0-9][0-9]* [.][0-9][0-9] [E] [+|-| ∈] [0-9][0-9] restricting to exactly two digits in the exponent part.

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 , real , int>* preserving the types and sequence of fields appearing in the record type definition.

```
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.
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. The types of these constructed names using variable name and record fields are same as field types defined in the record type definition correspondingly. Nested record definitions are supported in this language. However, the definition of the record type in nested positions cannot be used as standalone definitions.
A test case handling addition operation on two records and use of record variables in parameters list is depicted below. The record type #book declared in _main function is also visible in function _recordDemo1.
The language supports name equivalence and not structural equivalence, which means that similar structured record definitions are treated different. For example, #new and #book are the two record types with similar structure (sequence and type of variables) but different names.

```
_recordDemo1 input parameter list [record #book d5cc34, record #book d2cd]
output parameter list[record #book d3];
        record  #new
                type int : value;
                type real: cost;
        endrecord

        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;
        type record #new:b7bc34;
        b2.edition <--- 3;
        b2.price <--- 24.95;
        c2.edition <--- 2;
        c2.price <--- 98.80;
        % following is a valid statement as the types of d2, b2 and c2 are same
        d2<--- b2+ c2;
        %  Note that d2<--- b2 + b7bc34; is not type correct as the language follows name equivalence.
        %  Note that the types of b2 and b7bc34 are not same.
        [d2]<--- call _recordDemo with parameters[b2,c2];
        write(d2);
end
```

An assignment statement with variables from two different record types is not allowed in the language. Also, once a record type is defined, its re-occurrence anywhere in the program is not allowed and is treated as an error.

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. 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;
```

Union type: A union type is similar to record structure in its lexical formations other than the union keyword used. For example

```
union #student
        type int: rollno;
        type real:marks;
        type int: age;
endunion
```

As usual, the union data type refers to maximum of all fields memory allocation to the variables. Based on your understanding of unions, it is understood that the static type checking is not possible and it leads to spurious data access. In order to prevent the users from this situation, tagged union is supported in this language where the tag is computed at run time. The tag is part of the variant record following the same syntax as that of the record defining the (a) variant field as the union data type and (b) the fixed field of the tag. The tag can be of any primitive type integer or real.

```
definetype union #student as #newname;
record  #taggedunionexample
        type int: tagvalue;
        type #newname: field
endrecord
```

The tagged union variable is defined in the similar way as other variables are. For example

```
type record  #taggedunionexample b7bc34;
```

The variable b7bc34 of type #taggedunionexample which is a variant record has a fixed field tagvalue of integer type and the variant field of union type newname. The tagvalue field is used as

```
b7bc34.tagvalue = 1;
b7bc34. field.rollno = 23;
write(b7bc34. field.rollno); //No type error
write(b7bc34. field.marks); // Compiler reports the type error – dynamic type checking
b7bc34.tagvalue = 2;
b7bc34. field.marks = 97.5;
b7bc34.tagvalue = 3;
b7bc34. field.age = 21;
```

Similarly,

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;
```

Type definition (Aliases): The language supports type redefinition using the keyword definetype for record and union data type. For example

```
definetype union #student as #newname;
definetype record  #book as #newbook;
```

Since record and union type definitions are visible anywhere in the program, their type definitions representing equivalent names are also visible anywhere in the program. Hence, the type definition for other records or unions cannot be type defined similar to the ones already defined.

This language also supports that the record or union type variable construction can be expanded using the dot operator for any number of times. For e.g. if a variable is declared as follows

```
definetype record #definitionone as #definitiontwo
    record #definitionone
        type int : z
        type #pqr :a;    %non-recursive nested using definetype
        type record #abc: b;    %non-recursive nested without using definetype
        type #definitiontwo : c; %recursive nested using definetype
endrecord
```

Consider the following declaration statements,

```
type record #definitionone: c2d4;
type #definitiontwo: c5;
```

and different variables constructed as c2d4.c, c5.b. c2cd4.c.c.z, c2cd4.c.c.c.c.c.c.z, and so on.

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 parameter list [int c2, int d2cd]
output parameter list [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);
```

Input output statements: These are without any format and can take only one variable at a time to read or write. Examples are

```
read(b4d333); read(c3ddd34);

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

If the type of the variable is a record then writing is challenging and writes the values of all fields but if the variable is of variant record type then the write statement only prints the relevant field's value validated by its tag value at run time. Type checking is fun while handling different constructs in this language.

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];

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
[0-9][0-9]* [.][0-9][0-9] [E] [+|-|∈] [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_RUID	Identifier for the record or union type
with	TK_WITH	Keyword with
parameters	TK_PARAMETERS	Keyword parameters
end	TK_END	Keyword end
while	TK_WHILE	Keyword while
union	TK_UNION	Keyword union

endunion	TK_ENDUNION	Keyword end union
definetype	TK_DEFINETYPE	Keyword definetype
as	TK_AS	Keyword as
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_GE	Relational operator greater than or equal to
!=	TK_NE	Relational operator not equal to

Your compiler should report the proper expected lexical errors. For. e.g., <--, 23.4, 23.45E-, !, @@, &&, etc. The programmer was intended to write an assignment operator (i.e. <---) however, the programmer has mistakenly skipped the third - character. Similarly, the programmer was expected to write a decimal number, but he skipped the second digit after . operator.

Also, the lexeme such as <- should not be reported as a lexical error. In such case, double retraction is required, hence, tokenize < as TK_LT and – as TK_MINUS separately. Similarly, 23.abc should not be reported as a lexical error; tokenize 23 as TK_NUM and . as TK_DOT and abc as TK_FIELDID. Hence, it is expected that your compiler should report it as a lexical error. The method of tokenization and error reporting for the implementation of lexical analyzer will be properly discussed in the lectures.

Grammar:

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

1. *<program>* ===> <otherFunctions> <mainFunction>
2. <mainFunction>===> TK_MAIN <stmts> TK_END
3. <otherFunctions>===> <function><otherFunctions> | ∈
4. <function>===>TK_FUNID <input_par> <output_par> TK_SEM <stmts> TK_END
5. <input_par>===>TK_INPUT TK_PARAMETER TK_LIST TK_SQL <parameter_list> TK_SQR
6. <output_par>===>TK_OUTPUT TK_PARAMETER TK_LIST TK_SQL <parameter_list> TK_SQR | ∈
7. <parameter_list>===><dataType> TK_ID <remaining_list>
8. <dataType>===> <primitiveDatatype> |<constructedDatatype>
9. <primitiveDatatype>===> TK_INT | TK_REAL
10. <constructedDatatype>===>TK_RECORD TK_RUID | TK_UNION TK_RUID
11. <remaining_list>===>TK_COMMA <parameter_list> | ∈
12. <stmts>===><typeDefinitions> <declarations> <otherStmts><returnStmt>
13. <typeDefinitions>===><typeDefinition><typeDefinitions> |∈
14. <typeDefinition>===>TK_RECORD TK_RUID <fieldDefinitions> TK_ENDRECORD
15. <typeDefinition>===>TK_UNION TK_RUID <fieldDefinitions> TK_ENDUNION
16. <fieldDefinitions>===> <fieldDefinition><fieldDefinition><moreFields>
17. <fieldDefinition>===> TK_TYPE <primitiveDatatype> TK_COLON TK_FIELDID TK_SEM
18. <moreFields>===><fieldDefinition><moreFields> | ∈
19. <declarations> ===> <declaration><declarations>|∈
20. <declaration>===> TK_TYPE <dataType> TK_COLON TK_ID TK_COLON <global_or_not> TK_SEM
21. <global_or_not>===>TK_GLOBAL| ∈
22. <otherStmts>===> <stmt><otherStmts> | ∈
23. <stmt>===> <assignmentStmt> | <iterativeStmt>|<conditionalStmt>|<ioStmt>| <funCallStmt>
24. <assignmentStmt>===><SingleOrRecId> TK_ASSIGNOP <arithmeticExpression> TK_SEM
25. <singleOrRecId>===>TK_ID | TK_ID TK_DOT TK_FIELDID
26. <funCallStmt>===><outputParameters> TK_CALL TK_FUNID TK_WITH TK_PARAMETERS <inputParameters>
27. <outputParameters> ==> TK_SQL <idList> TK_SQR TK_ASSIGNOP | ∈
28. <inputParameters>===> TK_SQL <idList> TK_SQR
29. <iterativeStmt>===> TK_WHILE TK_OP <booleanExpression> TK_CL <stmt><otherStmts> TK_ENDWHILE

30. <conditionalStmt>===> TK_IF <booleanExpression> TK_THEN <stmt><otherStmts> TK_ELSE <otherStmts> TK_ENDIF
31. <conditionalStmt>===> TK_IF <booleanExpression> TK_THEN <stmt><otherStmts> TK_ENDIF
32. <ioStmt>===>TK_READ TK_OP <var> TK_CL TK_SEM | TK_WRITE TK_OP <var> TK_CL TK_SEM
33. <arithmeticExpression>===><arithmeticExpression> <operator> <arithmeticExpression>
34. <arithmeticExpression> ====>TK_OP <arithmeticExpression> TK_CL | <var>
35. <operator> ===> TK_PLUS | TK_MUL |TK_MINUS|TK_DIV
36. <booleanExpression>===>TK_OP <booleanExpression> TK_CL <logicalOp> TK_OP <booleanExpression> TK_CL
37. <booleanExpression>===> <var> <relationalOp> <var>
38. <booleanExpression>===> TK_NOT <booleanExpression>
39. <var>===> TK_ID | TK_NUM | TK_RNUM
40. <logicalOp>===>TK_AND | TK_OR
41. <relationalOp>===> TK_LT | TK_LE | TK_EQ |TK_GT | TK_GE | TK_NE
42. <returnStmt>===>TK_RETURN <optionalReturn> TK_SEM
43. <optionalReturn>===>TK_SQL <idList> TK_SQR | ∈
44. <idList>===> TK_ID <more_ids>
45. <more_ids>===> TK_COMMA <idList> | eps

46. <definetypestmt>===>TK_DEFINETYPE <A> TK_RUID TK_AS TK_RUID
47. <A>==>TK_RECORD | TK_UNION

***NOTE:** The above rules (**color coded in red**) needs some modifications/additions. It is suggested that students must read the language specification document properly and identify how to correct/modify/add/delete rules. Also, the above grammar represents the language described in this document, but it is not LL(1). There are also several rules which were left in the natural form of grammar and they do not conform to LL(1) specifications. What is not making it LL(1) compatible is left for you to resolve. You will be asked to work out many things and submit on paper the hand drawn DFA/NFA for the lexical analysis part. You will be also given the support of LL(1) compatible rules for the above language however, it is expected that you first modify the grammar on your own. More updates, implementation details, testcases and errata (if any) will be regularly updated on the course website.*