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.

☺ [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 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 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 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. before the return statement. A function can have within it a record definition and should be available globally.

The constructs of the language are described below.

Keywords

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

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.

The pattern for the identifiers is a bit clumsy, but you may have to design such if you need different types ☺ of placeholders for some language. The purpose and actions of such value holders (variables or identifiers) may be different and a simple identifier pattern may not suit the needs. My purpose of using the above pattern for the identifiers is to give you an exposure of pattern matching through DFA.

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.

☺ You will have the comfort with a pattern of function identifiers separate from that of the variable identifiers. The ease will be felt while creating symbol tables for representing the scope of variables. Had the patterns for both been same, you would have used, in order to differentiate between them, the structural information of the rules that generated them in two different contexts.

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 float: 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];
        print(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 divides 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];
        print(c2d3); print(c2d4);
end
```

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;
```

```
        print (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
        print(c7);
else
        print (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_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

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> | eps
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 | eps
7. <parameter_list>===><dataType> TK_ID <remaining_list>
8. <dataType>===> <primitiveDatatype> |<constructedDatatype>
9. <primitiveDatatype>===> TK_INT | TK_REAL
10. <constructedDatatype>===>TK_RECORD TK_RECORDID
11. <remaining_list>===>TK_COMMA <parameter_list> | eps
12. <stmts>===><typeDefinitions> <declarations> <otherStmts><returnStmt>
13. <typeDefinitions>===><typeDefinition><typeDefinitions> |eps
14. <typeDefinition>===>TK_RECORD TK_RECORDID <fieldDefinitions> TK_ENDRECORD TK_SEM
15. <fieldDefinitions>===> <fieldDefinition><fieldDefinition><moreFields>
16. <fieldDefinition>===> TK_TYPE <primitiveDatatype> TK_COLON TK_FIELDID TK_SEM
17. <moreFields>===><fieldDefinition><moreFields> | eps
18. <declarations> ===> <declaration><declarations>|eps
19. <declaration>===> TK_TYPE <dataType> TK_COLON TK_ID TK_COLON <global_or_not> TK_SEM
20. <global_or_not>===>TK_GLOBAL| eps
21. <otherStmts>===> <stmt><otherStmts> | eps
22. <stmt>===> <assignmentStmt> | <iterativeStmt>|<conditionalStmt>|<ioStmt>| <funCallStmt>
23. <assignmentStmt>===><SingleOrRecId> TK_ASSIGNOP <arithmeticExpression> TK_SEM
24. <singleOrRecId>===>TK_ID | TK_RECORDID TK_DOT TK_FIELDID
25. <funCallStmt>===><outputParameters> TK_CALL TK_FUNID TK_WITH TK_PARAMETERS <inputParameters>
26. <outputParameters> ==> TK_SQL <idList> TK_SQR TK_ASSIGNOP | eps

27. <inputParameters>===> TK_SQL <idList> TK_SQR
28. <iterativeStmt>===> TK_WHILE TK_OP <booleanExpression> TK_CL <stmt><otherStmts> TK_ENDWHILE
29. <conditionalStmt>===> TK_IF <booleanExpression> TK_THEN <stmt><otherStmts> TK_ELSE <otherStmts> TK_ENDIF
30. <conditionalStmt>===> TK_IF <booleanExpression> TK_THEN <stmt><otherStmts> TK_ENDIF
31. <ioStmt>===>TK_READ TK_OP <allVar> TK_CL TK_SEM | TK_WRITE TK_OP <allVar> TK_CL TK_SEM
32. <allVar>===><var>| TK_RECORDID
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 | eps
44. <idList>===> TK_ID <more_ids>
45. <more_ids>===> TK_COMMA <idList> | eps

NOTE: ***The above grammar represents the language described in this document, but it is not LL(1). There are several rules which were left in the natural form of grammar and 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 and hand written modified grammar. Once you submit the modified grammar, you will be given the support of LL(1) compatible rules for the above language.***

More updates, test cases and errata will be regularly updated on the course website.