Analysis of the Given Grammar

(February 13, 2026)

COLORS in this document
Black: for original rules in the grammar in Language Specification document
Red: specifies needs for modifications
Green: rules do not require modification
Brown: Not valid syntactically

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

1. **<program>** ===> <otherFunctions> <mainFunction>
   The LL(1) property for the nonterminal <program> is satisfied trivially due to single production for it. The following FOLLOW set will be required for verifying LL(1) compatibility for the nonterminal <otherFunctions> in (3).
   FOLLOW(<otherFunctions>) = FIRST(<mainFunction>) = {TK_MAIN}
   Note that all function definitions precede the main function definition and the language does not have constructs for function prototype declarations.
2. <mainFunction>===> TK_MAIN <stmts> TK_END
   The LL(1) property for the nonterminal <mainFunction> is satisfied trivially due to single production for it.
   FIRST(<mainFunction>) = FIRST(α) = {TK_MAIN}
   where α represents the right hand side of the production i.e. TK_MAIN <stmts> TK_END.
3. <otherFunctions>===> <function><otherFunctions> | eps
   The nonterminal <otherFunctions> need special care for verifying the LL(1) compatibility due to a nullable production. Let us first verify that the sets FIRST(<function><otherFunctions>) and FOLLOW<otherFunctions>) are disjoint.
   FIRST(<function><otherFunctions>) = FIRST(<function>) = {TK_FUNID}
   Note that <function> has no nullable production.
   Also from 1, we have FOLLOW(<otherFunctions>) = {TK_MAIN}

i.e.

FIRST(<function><otherFunctions) ∩ FOLLOW(<otherFunctions>) = ϕ

The given productions for <otherFunctions> are LL(1) compatible.

NOTE: Other properties such as ambiguity, left recursiveness, left factoring needs etc. causing violation of LL(1) compatibility of the rules will be discussed only if one or more of them exist. Otherwise I am focussing on violations due to epsilon productions. Also I will highlight the introduction of new nonterminals to incorporate the precedence of arithmetic operators and handling operations on variables of type record and union.

4. <function>===>TK_FUNID <input_par> <output_par> TK_SEM <stmts> TK_END

   This has no issue of violations in LL(1) compatibility. The nonterminal <input_par> is an essential construct to be part of the function definition while a function may or may not return values. Hence the <output_par> can have a syntax of (6) .

   FIRST(<function>) = {TK_FUNID}

5. <input_par>===>TK_INPUT TK_PARAMETER TK_LIST TK_SQL <parameter_list> TK_SQR

   Single production having no conflict has its FIRST set as given follows:

   FIRST(<input_par>) = {TK_INPUT}

6. <output_par>===>TK_OUTPUT TK_PARAMETER TK_LIST TK_SQL <parameter_list> TK_SQR | eps

   Presence of epsilon production makes us verify whether FIRST(α) ∩ FOLLOW(<output_par>) = ϕ or not, where α represents the right hand side

   TK_OUTPUT TK_PARAMETER TK_LIST TK_SQL <parameter_list> TK_SQR

   FIRST(α) = {TK_OUTPUT}

   Refer rule 4 to compute the FOLLOW(<output_par>) as below

   FOLLOW(<output_par>) = {TK_SEM}

   This implies that FIRST(α) ∩ FOLLOW(<output_par>) = ϕ

   Hence the given rules for <output_par> conform to the LL(1) specifications.

7. <parameter_list>===><dataType> TK_ID <remaining_list>

   The single production for <parameter_list> is trivially LL(1) compatible.

   FIRST(<parameter_list>) = FIRST(<dataType>)

   = FIRST(<primitiveDatatype>) ∪ FIRST(<constructedDatatype>)

   = {TK_INT, TK_REAL, TK_RECORD, TK_UNION}

Here we will allow the usage of record as well as union as parameters within the grammar as we did in this rule. This means that a function with following parameters will be considered as syntactically correct.
_recordDemo1 input parameter list [record #book d5cc34, union #newbook d2cd, int d3, real d5]
output parameter list[record #book d3];

But, a union variable passed as parameter or returned as a parameter will not make any sense due to the obvious reasons. However, this semantics will be captured in semantic analysis phase and you will have more clarity on type checking and semantic rules when your stage 2 work starts.

8. <dataType>===> <primitiveDatatype> |<constructedDatatype>
   FIRST(<primitiveDatatype>) ∩ FIRST(<constructedDatatype>)
   = {TK_INT, TK_REAL} ∩ { TK_RECORD, TK_UNION}
   = ϕ
   Since there is no nullable production for <dataType>, there is no need for computing FOLLOW(<dataType>).

9. <primitiveDatatype>===> TK_INT | TK_REAL
   FIRST(TK_INT) ∩ FIRST(TK_REAL) = {TK_INT} ∩ {TK_REAL} = ϕ
   Hence the rules for <primitiveDataType> are LL(1) compatible.
   FIRST(<primitiveDatatype>) = {TK_INT, TK_REAL}

10. <constructedDatatype>===> TK_RECORD TK_RUID | TK_UNION TK_RUID
    This non-terminal supports the parameter list or the declaration statements, which use the record and union datatype. If these are redefined using the <definetype> or alias method as given below

```
definetype record #abc as #pqr
record #abc
        type int : x;
```

```
        type real:y;
endrecord
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;    %using the alias of redefinition as above.
```

Let us look into the requirement of modification in rules for <constructedDatatype> from the perspective of two usages of data type-one in the declaration statement and the other in the parameter list. Both can use record and union data types to declare type of their variables/ parameters. Also both can use the alias or the type definition. The newly defined type is represented by the token TK_RUID.

Therefore the rules for <datatype> must include RUID in its first set also. The example in parameter list using both type names is as follows

```
_recordDemo1 input parameter list [ record #definitionone c2d4, #definitiontwo c5, record #abc d5cc34, #pqr d2cd]
output parameter list[record #abc d3];
```

Therefore, modify the rules for <constructedDatatype> as follows

<constructedDatatype>===> TK_RECORD TK_RUID | TK_UNION TK_RUID | TK_RUID ……10-a

FIRST(<constructedDatatype>) = {TK_RECORD, TK_UNION, TK_RUID}
Also since the first sets of RHS of the three rules for <constructedDatatype> are disjoint, the grammar is LL(1).
That is FIRST(TK_RECORD TK_RUID) ∩ FIRST(TK_UNION TK_RUID) ∩ FIRST(TK_RUID)
= {TK_RECORD}∩{TK_UNION}∩ {TK_RUID}
= ϕ

11. <remaining_list>===>TK_COMMA <parameter_list> | eps
There exist a nullable production for the non-terminal <remaining_list>. The RHS (right hand side) of the other production is such that the null string eps (epsilon) is not derivable from it.
Also
FIRST(TK_COMMA <parameter_list>) ∩ FOLLOW(<remaining_list>) should be empty.
FIRST(TK_COMMA <parameter_list>) = {TK_COMMA}
and
FOLLOW(<remaining_list>) = FOLLOW(parameter_list)..... from (7)
= {TK_SQR}
Hence,

FIRST(TK_COMMA <parameter_list>) ∩ FOLLOW(<remaining_list>)
= {TK_COMMA}∩ {TK_SQR}
= ϕ
Therefore both the given rules for the non-terminal <remaining_list> conform to LL(1) specifications.

12. <stmts>===><typeDefinitions> <declarations> <otherStmts><returnStmt>

The nonterminal <stmts> specifies the grammar for the body of the function. The ordering of other nonterminals such as <typeDefinitions> , <declarations> and <returnStmt> have fixed positions within the body of the function code.
There is no epsilon production <in <stmts> and a single non nullable production for <stmts> is LL(1) compatible.
In order to find the first set of <stmts>, we need to union the first sets as follows. The non-terminals <typeDefinitions>m <declarations> and <otherStmts> are nullable and have reachability through the application of their null productions.
FIRST(<stmts>) = FIRST(<typeDefinitions>) ∪ FIRST(<declarations>) ∪ FIRST(<otherStmts>) ∪ FIRST(<returnStmt>)
. .....using 13, 19,22 and 43
= {TK_RECORD, TK_UNION} ∪ {TK_TYPE} ∪ { TK_ID, TK_WHILE, TK_IF, TK_READ, TK_WRITE, TK_SQL,TK_CALL}∪ {TK_RETURN}
={ TK_RECORD, TK_UNION, TK_TYPE, TK_ID, TK_RUID, TK_WHILE, TK_IF, TK_READ, TK_WRITE, TK_SQL, TK_CALL,TK_RETURN}
Since the nonterminal <stmts> does not have a nullable production, we need not compute FOLLOW(<stmts>) for populating the parsing table. Recall the construction of the parsing table. The columns corresponding to the tokens in FIRST(<stmts>) corresponding to the row<stmts> are populated with the rule number 12.

13. <typeDefinitions>===><typeDefinition><typeDefinitions> | ∈
This is the place where the <definetypestmt> finds its existence. As it is used for record and union types redefinitions (or alias). Let us have this construct appear anywhere within the piece of code reserved for <typeDefinitions>

Redefine the rule 13
<typeDefinitions>===> <actualOrRedefined> <typeDefinitions> | ∈ ….13-a
Where
<actualOrRedefined> ===> <typeDefinition> | <definetypestmt> ……13-b

With this, the constructs <typeDefinition> and <definetypestmt> can appear in any order before the <stmts> construct within a function.

The semantic analyzer will be able to catch the relevance of the presence of a type redefinition and type equivalence later. If your test case has the following code

```
definetype union #old   as #new                    % line 1- derived using <definetypestmt>.
% following record type definition is derived using <typeDefinition>
 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;   % line 2
  endrecord
definetype record #definitionone   as #definitiontwo                % derived using <definetypestmt>.
```

Note that here the record #old in line 1 is not defined using <typeDefinition> grammar but will still be syntactically correct. The semantic analyzer will be able to catch this error. Also note that the record #definitionone is redefined as #definitiontwo even after its use above in line - 2. The semantic analyzer will require two pass mechanism to collect all information.

Let us analyze 13-b first.

Since FIRST(<typeDefinition>) ∩ FIRST(<definetypestmt>)

= {TK_RECORD, TK_UNION} ∩ {TK_DEFINETYPE} ..... using 14, 15 and 46

Therefore rule 13-b is LL(1) compatible.

And FIRST(<actualOrRedefined>) = {TK_RECORD, TK_UNION, TK_DEFINETYPE}

Now 13-a

FIRST(<actualOrRedefined>) ∩ FOLLOW(<typeDefinitions>)

= {TK_RECORD, TK_UNION, TK_DEFINETYPE} ∩ {TK_ID, TK_WHILE, TK_SQL, TK_IF, TK_READ, TK_WRITE, TK_CALL, TK_RETURN, TK_TYPE}

= {TK_RECORD, TK_UNION, TK_DEFINETYPE} ∩ {TK_ID, TK_WHILE, TK_SQL, TK_IF, TK_READ, TK_WRITE, TK_CALL, TK_RETURN, TK_TYPE}= {TK_RECORD, TK_UNION, TK_DEFINETYPE} ∩ {TK_ID, TK_WHILE, TK_SQL, TK_IF, TK_READ, TK_WRITE, TK_CALL, TK_RETURN, TK_TYPE}
= ϕ
Therefore, rule 13-a is LL(1) compatible

14. <typeDefinition>===>TK_RECORD TK_RUID <fieldDefinitions> TK_ENDRECORD
15. <typeDefinition>===>TK_UNION TK_RUID <fieldDefinitions> TK_ENDUNION

The rule 14 reaches to the
FIRST(TK_RECORD TK_RUID <fieldDefinitions> TK_ENDRECORD) = {TK_RECORD}
Similarly, rule 15 reaches the first of its RHS
FIRST(TK_UNION TK_RUID <fieldDefinitions> TK_ENDUNION) = {TK_UNION}
Also, the Firsts sets of the RHS of the two rules 14 and 15 for the non-terminal <typeDefinition> are disjoint.
{TK_RECORD}∪ {TK_UNION} = ϕ
Therefore rules 14 and 15 conform to the LL(1) specifications and do not require any modification.
Also, FIRST(<typeDefinition>) = {TK_RECORD, TK_UNION}

16. <fieldDefinitions>===> <fieldDefinition><fieldDefinition><moreFields>
As two consecutive occurrences of <fieldDefinition> are there at the RHS of the above production, this imposes a requirement of at least two fields in a record. There can be one or more fields afterwards. As the above rule is single and non nullable, it is LL(1) compatible.
FIRST(<fieldDefinitions>) = FIRST(<fieldDefinition>) = {TK_TYPE}

17. <fieldDefinition>===> TK_TYPE <primitiveDatatype> TK_COLON TK_FIELDID TK_SEM
Here, the type of the field needs major change to support the nested record and union data types. As per the descriptions given in the language specification document, a field name can also be of type union or record. However, only the alias of the union or record type is allowed in the type

definition of these fields. A minor correction in the example given in the language specifications document that uses type newname: field should be updated using a # as prefix. The updated example is given below.

```
definetype union #student as #newname
record #taggedunionexample
        type int : tagvalue;
        type #newname : field; %add # before newname which was missing in the document earlier.
 endrecord
```

Therefore the rule for <fielddefinition> is modified as given below.

<fieldDefinition>===> TK_TYPE <fieldType > TK_COLON TK_FIELDID TK_SEM

<fieldtype>===> <primitiveDatatype> | <constructedDatatype> …… Add this rule in the grammar 17(a)

Note that the record or union data type can be made nested by using the alias. This grammar will not support the usage of usual type definition syntax of union and record to be nested here. To explain this, the following example **will not be syntactically valid.**

```
record #taggedunionexample
        type int : tagvalue;
        type union #student                    %NOT CORRECT SYNTACTICALLY
                type int:rollno;
                type real marks;
                type int age;
              endunion : field;  %add # before newname which was missing in the document earlier.
       endunion
 endrecord
```

Now check the compatibility of the two rules in 17(a)

FIRST(<primitiveDatatype>) ∩ FIRST(TK_RUID) = {TK_INT, TK_REAL} ∩ {TK_RUID} = ϕ

And FIRST(<fieldType>)={TK_INT, TK_REAL, TK_RUID}

FIRST(<fieldDefinition>) = {TK_TYPE}

18. <moreFields>===><fieldDefinition><moreFields> | eps

Let us find

FOLLOW(<moreFields>) = ϕ ∪ FOLLOW(<fieldDefinitions>) = {TK_ENDRECORD, TK_ENDUNION} ..... using rules 14, 15 and 16

FIRST(<fieldDefinition><moreFields>) ∩ FOLLOW(<moreFields>)

= {TK_TYPE} ∩ {TK_ENDRECORD, TK_ENDUNION}

= ϕ

Hence, the above two rules for the non terminal <moreFields> are LL(1) compatible.

19. <declarations> ===> <declaration><declarations>| ∈

FOLLOW(<declarations>) = { TK_ID, TK_WHILE, TK_SQL, TK_IF, TK_READ, TK_WRITE, TK_CALL, TK_RETURN}

FIRST(<declaration><declarations>) = FIRST(<declaration>) = {TK_TYPE}

We observe that both of the above sets are disjoint. Therefore both the above productions for the non terminal <declarations> conform to LL(1) specifications.

20. <declaration>===> TK_TYPE <dataType> TK_COLON TK_ID TK_COLON <global_or_not> TK_SEM

<declaration>===> TK_TYPE <dataType> TK_COLON TK_ID <global_or_not> TK_SEM

Also to incorporate the optional existence of global keyword prefixed with a colon is reflected as modification in rule 21.

FIRST(<declaration>) = {TK_TYPE}

Rule conforms to LL(1) specifications.

21. <global_or_not>===>TK_GLOBAL| eps

This rule is also modified as follows

<global_or_not>===>TK_COLON TK_GLOBAL| eps
FOLLOW(<global_or_not>) = {TK_SEM} ..... using 20
and
FIRST(TK_COLON TK_GLOBAL) = {TK_COLON}
As
FOLLOW(<global_or_not>) ∩ FIRST(TK_COLON TK_GLOBAL) = {TK_SEM} ∩ {TK_COLON} = ϕ
Therefore the new rule for <global_or_not> is LL(1) compatible.

22. <otherStmts>===> <stmt><otherStmts> | eps
FOLLOW(<otherStmts>) =
FOLLOW(<otherStmts >)
FOLLOW(<otherStmts>) = {TK_RETURN, TK_ENDWHILE, TK_ENDIF, TK_ELSE} [Note: <otherStmts> also derive an empty string]

.....using 23

23. <stmt>===> <assignmentStmt> | <iterativeStmt>|<conditionalStmt>|<ioStmt>| <funCallStmt>
Refering rules 24, 29, 30, 32 and 26, we conclude that the five productions given above for the nonterminal <stmt> are not nullable.
We must check the LL(1) property that the FIRST sets of the RHSs of all productions above are disjoint.
FIRST(<assignmentStmt>) ∩ FIRST(<iterativeStmt>) ∩ FIRST(<conditionalStmt>) ∩ FIRST(<ioStmt>) ∩ FIRST(<funCallStmt>)
= {TK_ID} ∩ {TK_WHILE} ∩ {TK_IF} ∩ {TK_READ, TK_WRITE} ∩ {TK_SQL, TK_CALL}
= ϕ
Hence, the above five productions for the nonterminal <stmt> conform to LL(1) specifications.
and ,
FIRST(<stmt>) = {TK_ID, TK_WHILE, TK_IF, TK_READ, TK_WRITE, TK_SQL, TK_CALL}

24. <assignmentStmt>===><SingleOrRecId> TK_ASSIGNOP <arithmeticExpression> TK_SEM
FIRST(<assignmentStmt>) = FIRST(<singleOrRecId>) = {TK_ID} ..... using 25 e

25. <singleOrRecId>===>TK_ID | TK_ID TK_DOT TK_FIELDID

The second rule for <singleOrRecId> does not support the record or union type variable construction expanded using the dot operator for any number of times. As you have seen in the examples given under rules for <constructedDatatype>, 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.
Then the following rules for <singleOrRecId>will not help as the token stream would then be TK_ID TK_DOT TK_FIELDID TK_DOT TK_FIELDID TK_DOT TK_FIELDID TK_DOT TK_FIELDID TK_DOT TK_FIELDID TK_DOT TK_FIELDID TK_DOT TK_FIELDID TK_DOT TK_FIELDID for a variable c2cd4.c.c.c.c.c.c.z (for example)

<singleOrRecId>===>TK_ID | TK_ID TK_DOT TK_FIELDID

So we need to redesign the rule
<singleOrRecId>===>TK_ID TK_DOT TK_FIELDID

As
<singleOrRecId> ===><constructedVariable>

by introducing a new nonterminal say <constructedVariable> which will represent the variables as described above.

Therefore the rules for <singleOrRecId> will become
<singleOrRecId>===>TK_ID | <constructedVariable> ..... 25 a

We can define a recursive rule for this as follows.

<constructedVariable> ===> TK_ID <oneExpansion><moreExpansions> ----------25 b
<oneExpansion>===> TK_DOT TK_FIELDID ----------25 c
<moreExpansions>===> <oneExpansion> <moreExpansions> | ∈ ---------- 25 d

Here two more non-terminals are introduced to address nested record types for the variables constructed using them. It is clear that the nesting level of zero, i.e. with no record type field, the nonterminal <oneExpansion> derives TK_DOT TK_FIELDID and in this case the null production for <moreExpansions> is used. For nested definitions, a number of times <moreExpansions> ===> <oneExpansion> <moreExpansions> rule is applied as required..

If you wish you can replace the name <singleOrRecId> with <singleOrRecOrUnionID> everywhere to make it self-explanatory.

Before going for verifying the LL(1) compatibility of the rule for < singleOrRecId>, let us examine the newly introduced rules.

Rule 25 b is a single rule and is non nullable.
FIRST(<constructedVariable>) = {TK_ID}
FOLLOW(<constructedVariable>) = FOLLOW(<singleOrRecId>) = {TK_ASSIGNOP}

Rule 25 c is LL(1) trivially.
FIRST(<oneExpansion>) = {TK_DOT}

Rule 25 d requires that FIRST (<oneExpansion> <moreExpansions>) and FOLLOW(<moreExpansions>) should be disjoint.
FIRST (<oneExpansion> <moreExpansions>) ∩ FOLLOW(<moreExpansions>)
= {TK_DOT} ∩ FOLLOW(<constructedVariable>)
= {TK_DOT} ∩ FOLLOW(<singleOrRecId >)
= {TK_DOT} ∩ {TK_ASSIGNOP, TK_CL}
= ϕ

Using 25 a and 25 b

<singleOrRecId>===>TK_ID | TK_ID <oneExpansion><moreExpansions>

Both of the above productions for the nonterminal <singleOrRecId> are non nullable and their first sets are not disjoint. This requires left factoring
The above two rules are left factored using new non terminal <new_24>
<singleOrRecId>===>TK_ID <option_single_constructed> ……25 e
<option_single_constructed>===> ∈ | <oneExpansion><moreExpansions> .....25 f

25 e is trivially LL(1) compatible.
For both rules for <option_single_constructed> in 25 f, FIRST (<oneExpansion> <moreExpansions>) and FOLLOW(<option_single_constructed>) should be disjoint.
FIRST (<oneExpansion> <moreExpansions>) ∩ FOLLOW(<option_single_constructed>)
= {TK_DOT} ∩ FOLLOW(<singleOrRecId>)
= {TK_DOT} ∩ {TK_ASSIGNOP, TK_CL}

= ϕ.

Therefore, the above productions are LL(1) compatible.

The final rules for <singleOrRecId> include 25 e, f, c and d.
Using 25 e, FIRST(<singleOrRecId>) = {TK_ID}

26. <funCallStmt>===><outputParameters> TK_CALL TK_FUNID TK_WITH TK_PARAMETERS <inputParameters>
The rule must have at its end TK_SEM to represent semicolon
<funCallStmt>===><outputParameters> TK_CALL TK_FUNID TK_WITH TK_PARAMETERS <inputParameters> TK_SEM
The rule is trivially LL(1) compatible.
FIRST(<funCallStmt>) = FIRST(<outputParameters>) ∪ FOLLOW(<outputParameters>)
= {TK_SQL} ∪ {TK_CALL} = {TK_SQL, TK_CALL}

27. <outputParameters> ==> TK_SQL <idList> TK_SQR TK_ASSIGNOP | ∈
FOLLOW(<outputParameters>) = {TK_CALL}
Due to the presence of the nullable production for the non terminal <outputParameters>, we check the following LL(1) property
FIRST(TK_SQL <idList> TK_SQR TK_ASSIGNOP) ∩ FOLLOW(<outputParameters>)
= {TK_SQL} ∩ {TK_CALL} = ϕ
Hence, the productions in 27 are LL(1) compatible.
FIRST(<outputParameters>) = {TK_SQL}

28. <inputParameters>===> TK_SQL <idList> TK_SQR
LL(1) compatible with FIRST(<inputParameters>) = {TK_SQL}
29. <iterativeStmt>===> TK_WHILE TK_OP <booleanExpression> TK_CL <stmt><otherStmts> TK_ENDWHILE
LL(1) compatible with FIRST(<iterativeStmt>) = {TK_WHILE}

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

The two rules (30) and (31) for the nonterminal <conditionalStmt> need modifications as they originally form ambigous grammar. We need to perform the left factoring as follows.

<conditionalStmt>===> TK_IF TK_OP <booleanExpression> TK_CL TK_THEN <stmt><otherStmts> <elsePart>..... number rule as 30-a
<elsePart>===>TK_ELSE <stmt><otherStmts> TK_ENDIF | TK_ENDIF ..... number rules as 31-a

Notice that I also added TK_OP and TK_CL in new rule 30-a to enclose boolean expression . These two tokens were missing earlier.
Also I am putting a constraint of at least one statement within else part.
Now there is a single rule for <conditionalStmt> which is non nullable and its first set is {TK_IF}
But the newly introduced nonterminal <elsePart> has two non nullable productions and their corresponding FIRST sets are disjoint.
FIRST(TK_ELSE <otherStmts> TK_ENDIF) ∩ FIRST(TK_ENDIF) = {TK_ELSE} ∩ {TK_ENDIF} = ϕ
Therefore the new set of rules for <elsePart> is also LL(1) compatible.

And,
FIRST(<conditionalStmt>) = {TK_IF}
FIRST(<elsePart>) = {TK_ELSE, TK_ENDIF}

32. <ioStmt>===>TK_READ TK_OP <var> TK_CL TK_SEM | TK_WRITE TK_OP <var> TK_CL TK_SEM

Both the above productions are non nullable.
and,
FIRST(TK_READ TK_OP <var > TK_CL TK_SEM) ∩ FIRST(TK_WRITE TK_OP <var> TK_CL TK_SEM)

= {TK_READ} ∩ {TK_WRITE} = ϕ
Therefore, the above productions are LL(1) compatible.
and,
FIRST(<ioStmt>) = {TK_READ, TK_WRITE}

33. <arithmeticExpression>===><arithmeticExpression> <operator> <arithmeticExpression>
I had left a major correction for you to do in the arithmetic expression grammar. You were required to impose precedence of operators and the operations on record variables.
34. <arithmeticExpression> ====>TK_OP <arithmeticExpression> TK_CL | <var>
An arithmetic expression is expected to take as argument an identifier, statically available integer number, real number, a record identifier and a record variable identifier (constructed by expanding the name using dot with a field name). These are represented in the non-terminal <var> in rule number 39-a.

Now we reformulate the rules 33, 34 and 35 to impose precedence.
<arithmeticExpression> ===> <arithmeticExpression> <lowPrecedenceOperators> <term> | <term> ..... (A1)
<term> ===> <term> <highPrecedenceOperators> <factor> | <factor> ..... (A2)
<factor> ===> TK_OP <arithmeticExpression> TK_CL | <var> ..... (A3)
<highPrecedenceOperator>===> TK_MUL | TK_DIV..... (A4)
<lowPrecedenceOperators> ===> TK_PLUS | TK_MINUS ..... (A5)

[Note : I have given new numbers for referring to the rules, but you can have your own numbering scheme for the rules]
Now let us verify the rules A1-5 for the nonterminals <arithmeticExpression>, <term>, <factor>, <lowPrecedenceOperators>, <highPrecedenceOperators>,
The rule A1 needs left recursion elimination and the new set of productions are obtained by introducing new nonterminal, say <expPrime>
<arithmeticExpression> ===> <term> <expPrime>
<expPrime> ===> <lowPrecedenceOperators> <term> <expPrime> | ∈

Similarly rule A2 also needs left recursion elimination. The new set of productions are obtained by introducing new nonterminal, say, <termPrime>
<term>===> <factor> <termPrime>
<termPrime> ===> <highPrecedenceOperators><factor> <termPrime> | ∈

Now we have two more nonterminals <expPrime> and <termPrime>. The complete set of non left recursive, unambigous arithmetic expression grammar then becomes with new numbers B1-B7

B1) <arithmeticExpression> ===> <term> <expPrime>
B2) <expPrime> ===> <lowPrecedenceOperators> <term> <expPrime> | ∈
B3) <term>===> <factor> <termPrime>
B4) <termPrime> ===> <highPrecedenceOperators><factor> <termPrime> | ∈
B5) <factor> ===> TK_OP <arithmeticExpression> TK_CL | <var>
B6) <highPrecedenceOperator>===> TK_MUL | TK_DIV
B7) <lowPrecedenceOperators> ===> TK_PLUS | TK_MINUS

Analysis of B1: As there is only single rule available for the nonterminal <arithmeticExpression>, we must see that is not nullable
FIRST(<arithmeticExpression>) = FIRST(<term>) = FIRST(<factor>) = {TK_ID, TK_NUM, TK_RNUM, TK_OP}
There is no need to go further to check LL(1) compatibility. The grammar for <arithmeticExpression> is LL(1) compatible.
Analysis of B2: One of the productions for the nonterminal <expPrime> is nullable.
Let us verify FIRST(<lowPrecedenceOperators> <term> <expPrime>) and FOLLOW(<expPrime>) are disjoint.

FIRST(<lowPrecedenceOperators> <term> <expPrime>) = FIRST(<lowPrecedenceOperators>)
= {TK_PLUS, TK_MINUS} ..... using analysis of B7
and,
FOLLOW(<expPrime>) = FOLLOW(<arithmeticExpression>) ..... using B1
= {TK_SEM} ..... using 24

This implies that

FIRST(<lowPrecedenceOperators> <term> <expPrime>) ∩ FOLLOW(<expPrime>)

= {TK_PLUS, TK_MINUS}∩{TK_SEM}

= ϕ

Therefore, B2 is also LL(1) compatible.

Analysis of B3:

As there is only single rule available for the nonterminal <term>, we must see that is not nullable

FIRST(<term>)

= FIRST(<factor>)

= {TK_ID, TK_NUM, TK_RNUM, TK_OP} ..... using analysis of B5

There is no need to go further to check LL(1) compatibility. The grammar for <term> is LL(1) compatible.

Analysis of B4:

One of the productions for the nonterminal <termPrime> is nullable.

Let us verify FIRST(<highPrecedenceOperators><factor> <termPrime>) and FOLLOW(<termPrime>) are disjoint.

FIRST(<highPrecedenceOperators><factor> <termPrime>) = FIRST(<highPrecedenceOperators>)

={ TK_MUL, TK_DIV} ..... using analysis of B6

And,

FOLLOW(<termPrime>)

= FOLLOW(<term>)..... using B3

= FIRST(<expPrime>) ∪ FOLLOW(<expPrime>) ..... using B2

= FIRST(<lowPrecedenceOperators>) ∪ FOLLOW(<expPrime>) ..... using B2

= {TK_PLUS, TK_MINUS} ∪ FOLLOW(<expPrime>) ..... using analysis of B7

= {TK_PLUS, TK_MINUS} ∪ FOLLOW(<arithmeticExpression>)..... using B1
= {TK_PLUS, TK_MINUS} ∪ {TK_SEM} ..... using 24
= {TK_PLUS, TK_MINUS, TK_SEM}

Therefore,

FIRST(<highPrecedenceOperators><factor> <termPrime>) ∩ FOLLOW(<termPrime>)

= { TK_MUL, TK_DIV} ∩ {TK_PLUS, TK_MINUS, TK_SEM}

= ϕ

Hence the rules defined by B4 conform to LL(1) specifications.

Analysis of B5: There are two productions for the non terminal <factor>.

FIRST(TK_OP <arithmeticExpression> TK_CL) = {TK_OP}

and,

FIRST(<var>) = {TK_ID, TK_NUM, TK_RNUM}

It is now obvious that no production for <factor> is a nullable production and,

FIRST(TK_OP <arithmeticExpression> TK_CL) ∩ FIRST(<var>) = ϕ

Therefore the grammar for <factor> is LL(1) compatible.

and,

FIRST(<factor>) = {TK_ID, TK_NUM, TK_RNUM, TK_OP}

Analysis of B6: The two productions of the nonterminal <highPrecedenceOperators> are trivially LL(1) compatible

FIRST(<highPrecedenceOperators>) = { TK_MUL, TK_DIV}

Analysis of B7: The two productions of the nonterminal <lowPrecedenceOperators> are also trivially LL(1) compatible

FIRST(<lowPrecedenceOperators>) = { TK_PLUS, TK_MINUS}

35. <operator> ===> TK_PLUS | TK_MUL |TK_MINUS|TK_DIV

This rule is discarded now.

36. <booleanExpression>===>TK_OP <booleanExpression> TK_CL <logicalOp> TK_OP <booleanExpression> TK_CL
FIRST(TK_OP <booleanExpression> TK_CL <logicalOp> TK_OP <booleanExpression> TK_CL) = { TK_OP}
and the production is not nullable.

37. <booleanExpression>===> <var> <relationalOp> <var>
FIRST(<var> <relationalOp> <var>) = FIRST(<var>)
= {TK_ID, TK_NUM, TK_RNUM}
and the production is not nullable.

38. <booleanExpression>===> TK_NOT <booleanExpression>
Introduce the parentheses pait around the <booleanExpression> as follows
<booleanExpression>===> TK_NOT TK_OP <booleanExpression> TK_CL ….38-a

Considering the above rules 36-38 for analysis, we observe that none of the three productions for the nonterminal <booleanExpression> is nullable (using First set of <var> from 39)
and, The first sets of the right hand sides of the three productions are disjoint.
FIRST(TK_OP <booleanExpression> TK_CL <logicalOp> TK_OP <booleanExpression> TK_CL)
∩ FIRST(<var> <relationalOp> <var>)
∩ FIRST(TK_NOT TK_OP <booleanExpression> TK_CL)
= {TK_OP} ∩ {TK_ID, TK_NUM, TK_RNUM}∩ {TK_NOT}
= ϕ
Hence, the rules 36, 37 and 38-a for the nonterminal <booleanExpression> are LL(1) compatible.

39. <var>===> TK_ID | TK_NUM | TK_RNUM
This rule needs modifications. The non-terminal <var> participates in read and write statements and in arithmetic and Boolean expressions. It is expected to represent the variable identifier, the record and union variables constructed by expanding the identifier name with dot operator followedby the field name. Also <var> represents the statically available integer and real numbers.

The rule for <var> is modified as follows

<var>===> <singleOrRecId> | TK_NUM | TK_RNUM .....39 a.

The three rules on the RHS are LL(1) compatible as

FIRST(<singleOrRecId>) ∩ FIRST(TK_NUM) ∩ FIRST(TK_RNUM) = ϕ

And

FIRST(<var>) = {TK_ID, TK_NUM, TK_RNUM}

40. <logicalOp>===>TK_AND | TK_OR

I have not defined the precedence of AND over OR or vice versa. We have the support to enclose the boolean expression in parentheses.

Trivially the productions for <logicalOp> conform to LL(1) specifications

41. <relationalOp>===> TK_LT | TK_LE | TK_EQ |TK_GT | TK_GE | TK_NE

All six productions above for the nonterminal <relationalOp> are trivially LL(1) compatible and

FIRST(<relationalOp>) = { TK_LT, TK_LE, TK_EQ , TK_GT , TK_GE , TK_NE}

42. <returnStmt>===>TK_RETURN <optionalReturn> TK_SEM

FIRST(<returnStmt>) = {TK_RETURN}

and the production is not nullable, hence the rule is LL(1) compatible.

43. <optionalReturn>===>TK_SQL <idList> TK_SQR | eps

FIRST(TK_SQL <idList> TK_SQR) = { TK_SQL}

and,

FOLLOW(<optionalReturn>) = {TK_SEM}..... using 42

Since both of the above sets are disjoint, the grammar for the nonterminal <optionalReturn> is LL(1) compatible.

44. <idList>===> TK_ID <more_ids>

FIRST(<idList>) = {TK_ID} and the production is not nullable, hence the rule is LL(1) compatible.

45. <more_ids>===> TK_COMMA <idList> | eps

FIRST(TK_COMMA <idList>) = {TK_COMMA}

and,

FOLLOW(<more_ids>) = FOLLOW(<idList>)..... using 44

= {TK_SQR} ..... using 43

This implies that

FIRST(TK_COMMA <idList>) ∩ FOLLOW(<more_ids>)

= {TK_COMMA} ∩ {TK_SQR}

= ϕ

Therefore, the rules for the nonterminal <more_ids> are LL(1) compatible.

46. <definetypestmt>===>TK_DEFINETYPE <A> TK_RUID TK_AS TK_RUID

FIRST(<definetypestmt>) = {TK_DEFINETYPE} and the production is not nullable, hence the rule is LL(1) compatible.

47. <A>==>TK_RECORD | TK_UNION

FIRST(TK_RECORD) ∩ FIRST(TK_UNION)

= { TK_RECORD } ∩ {TK_UNION} LL(1) Compatible

= ϕ (LL(1) Compatible)