Analysing Given Original Grammar

(February 14, 2020)

COLORS in this document

Black: for original rules in the grammar in ERPLAG specification document

Blue: causes for trouble but do not need modifications in the rule directly

Red: specifies needs for modifications

Purple: rule needs modification due to discrepancy with token definitions and not due to violation of LL(1)

Green: rules do not require modification

1. <program> → <moduleDeclarations> <otherModules><driverModule><otherModules>

   Major discrepancy lies here.
   RULE 1 will remain same but a modification in token definition for DEF and ENDDEF for driver will be required.
   Observe the LL(1) property violation described for rule 4.
   There is a need to modify the token that forms the FIRST of <driverModule> so that the conflict can be resolved.
   I will modify rule 5 by defining the DEF of driver module as DRIVERDEF and END as DRIVERENDDEF respectively which will require us to have new pattern for start of driver. The three consecutive characters <<< and >>> are used to define the patterns in place of << and >> for representing DRIVERDEF and DRIVERENDDEF tokens as was mentioned in earlier updates.

2. <moduleDeclarations>→ <moduleDeclaration><moduleDeclarations> | ε

   FIRST(<moduleDeclarations>) = FIRST(ModuleDeclaration> = {DECLARE}
   FOLLOW(<moduleDeclarations>)=FIRST(<otherModules>) U FOLLOW(<otherModules>)
   =FIRST(<otherModules>) U FIRST(<driverModule>) U {$}
   ={DEF}U {DRIVERDEF} U {$}
   = {DEF, DRIVERDEF, $}

Since <moduleDeclarations>→ ε , we can see that FIRST(<moduleDeclarations>)∩ FOLLOW(<moduleDeclarations>)=ϕ
That is, LL(1) property that no element in FIRST(<moduleDeclaration><moduleDeclarations>) should be in FOLLOW(<moduleDeclarations>) holds good.

3. <moduleDeclaration> → DECLARE MODULE ID SEMICOL

As there is only one rule for the non terminal <moduleDeclaration>, and is not left recursive, LL(1) property hold good.
FIRST(<moduleDeclaration>) = {DECLARE}

4. <otherModules> → <module><otherModules>| ε

Using rule 6 to compute FIRST(<module><otherModules>) we get
FIRST(<module><otherModules>) = FIRST(<module>) = {DEF}
Since <otherModules>→ ε also, then LL(1) property is violated if any element in FIRST(<module><modules>) is also in
FOLLOW(<otherModules>) = FIRST(<driverModule>) U {$}
= {DRIVERDEF} U {$}
= {DRIVERDEF, $}
The FIRST(<module><otherModules>) ∩ FOLLOW(<otherModules>) = ϕ.

With the introduction of DRIVERDEF, the above two rules become LL(1) compatible

5. <driverModule> → DEF DRIVER PROGRAM ENDDEF <moduleDef>
Only one rule for the non terminal <driverModule>
FIRST(<moduleDeclaration>) = {DEF}
The rule in itself is not violating LL(1) property, but needs modification to incorporate changes required to make rule 4 LL(1) compatible.
The rule will be modified with new tokens as follows
<driverModule>→ DRIVERDEF DRIVER PROGRAM DRIVERENDDEF <moduleDef> ……..5a

6. <module> →DEF MODULE ID ENDDEF TAKES INPUT SQBO <input_plist> SQBC SEMICOL <ret><moduleDef>
Only one rule for the non terminal <module>
FIRST(<moduleDeclaration>) = {DEF}

7. <ret> → RETURNS SQBO <output_plist> SQBC SEMICOL | ε

Let α represent the RHS of the first rule and β represent the RHS of the second rule for the non terminal <ret>
FIRST(α) = {RETURNS}
From rule 12, we get
FIRST(<moduleDef>) = {START}
From rule 6, we get
FOLLOW(<ret>) = FIRST(<moduleDef>) ={START}
As rule 7 derives epsilon, we need to verify the following LL(1) property.
Observe that an element in the set FIRST(α) is not in FOLLOW(<ret>) as {RETURNS} and {START} are disjoint.
Hence epsilon production does not violate the LL(1) property.

8. <input_plist> → <input_plist> COMMA ID COLON <dataType> | ID COLON <dataType>
This rule involves left recursion, which violates the LL(1) property.
Hence needs left recursion elimination.
let
α represent COMMA ID COLON <dataType>
β represent ID COLON <dataType>
Then the rule
<input_plist>→<input_plist> α | β
is modified as follows

<input_plist> → β <N1>
<N1> → α <N1> | ε

[Note: I will introduce the non terminal symbols as N1, N2, N3, and so on wherever we will require for the modification of rules.]

Replacing α and β with the actual strings, we get the modified rules as
<input_plist> → ID COLON <dataType><N1> …………….8a
<N1> → COMMA ID COLON <dataType> <N1> | ε ……………..8b

Now let us analyze these new rules whether they conform to the LL(1) property or not.

Rule 8a :

Only one non recursive rule for the non terminal <input_plist>
FIRST(<input_plist>) = {ID}

Rule 8b :

FIRST(COMMA ID COLON <dataType> <N1>) = {COMMA}

As <N1>→ ε
we must look at the FOLLOW(<N1>).
Using rule 6 we find FOLLOW(<input_plist>) as {SQBC} and get
FOLLOW(<N1>) = FOLLOW(<input_plist>) = {SQBC}

This conforms both rules for the non terminal <N1> (as specified in 8b) to LL(1)

9. <output_plist> → <output_plist> COMMA ID COLON <type> | ID COLON <type>

This rule involves left recursion, which violates the LL(1) property.
Hence needs left recursion elimination.
let

α represent COMMA ID COLON <type>
β represent ID COLON <type>
Then the rule
<output_plist>→<output_plist> α | β
is modified as follows

<output_plist>→ β <N2>
<N2> → α <N2> | ε

Which becomes

<output_plist>→ ID COLON <type> <N2> …………..9a
<N2> → COMMA ID COLON <type><N2> | ε …………...9b

The new rules 9a and 9b conform to LL(1) (refer 8 for similar description)

10. <dataType> → INTEGER | REAL | BOOLEAN | ARRAY SQBO <range> SQBC OF <type>

Here one rule is modified as follows

<dataType>→ ARRAY SQBO <range_arrays> SQBC OF <type> ....10 a

where <range_arrays> accommodates variable identifiers in defining range of array type. The <range> was only deriving ranges using only the integer values (static constants, e.g. array[2..10] of integer) while the <range_arrays> can derive array types as array[a..b] of integers (e.g.) where a and b are variable identifiers.

<range_arrays> →<index> RANGEOP <index> .....10 b

The non-terminal <index> is already given in rule number 23.
The <range> non-terminal continues to be used in defining the iterative statement for the FOR loop.

Let the strings of grammar symbols on the right hand side of the production rules for <dataType> are represented by the greek letters α, β, γ, δ such that

α represents INTEGER
β represents REAL
γ represents BOOLEAN
δ represents ARRAY SQBO <range_arrays> SQBC OF <type>

FIRST(α) = {INTEGER}
FIRST(β) ={ REAL}
FIRST(γ)={BOOLEAN}
FIRST(δ)= {ARRAY}

We observe that
FIRST sets of the right hand sides of the 4 rules are disjoint.
The rule is not left recursive and does not need left factoring.
There is no nullable production, hence there no need to check the disjointness of the FOLLOW(<dataType>) and the first sets of RHS of non null productions.

11. <type> → INTEGER | REAL | BOOLEAN

Let

α represents INTEGER

β represents REAL

γ represents BOOLEAN

Then, first sets of the RHS are disjoint.

FIRST(α) = {INTEGER}

FIRST(β) ={ REAL}

FIRST(γ)={BOOLEAN}

The production rules for the non terminal <type> conform to the LL(1) property.

12. <moduleDef> → START <statements> END

Only one rule for the non terminal <moduleDef>

FIRST(<moduleDef>) = {START}

13. <statements> →<statement> <statements> | ε

FOLLOW(<statements>) = {END, BREAK} …….From RHS of rules 12, 42, 44 and 45

FIRST(<statements>) = FIRST(<statement>)

= { GET_VALUE, PRINT, ID, SQBO, USE, DECLARE, SWITCH, FOR, WHILE}

Both of these are disjoint, hence LL(1) compatible.

14. <statement> →<ioStmt>|<simpleStmt>|<declareStmt>|<condionalStmt>|<iterativeStmt>

Let us compute the set intersection of FIRST sets of the RHSs of the rule for <statement>.

We get,

FIRST(<ioStmt>) ……get from 15

∩ FIRST(<simpleStmt>) ……from 18

∩ FIRST(<declareStmt>)

∩ FIRST(<condionalStmt>

∩ FIRST(<iterativeStmt>)

={GET_VALUE, PRINT} ∩ {ID, SQBO, USE} ∩{DECLARE} ∩ {SWITCH} ∩ {FOR, WHILE}

= ϕ

Therefore, the RHSs of all five rules for <statement> above have disjoint FIRST sets . therefore the LL(1) compatibility is ensured.

Also, we compute FIRST(<statement>) for use in 13 above.

FIRST(<statement>) = FIRST(<ioStmt>) ……get from 15
U FIRST(<simpleStmt>) ……from 18
U FIRST(<declareStmt>)
U FIRST(<condionalStmt>
U FIRST(<iterativeStmt>)

={GET_VALUE, PRINT} U {ID, SQBO, USE} U{DECLARE} U {SWITCH} U {FOR, WHILE}
= { GET_VALUE, PRINT, ID, SQBO, USE, DECLARE, SWITCH, FOR, WHILE}

15. <ioStmt>→GET_VALUE BO ID BC SEMICOL | PRINT BO <var> BC SEMICOL

Let
α represents GET_VALUE BO ID BC SEMICOL
β represents PRINT BO <var> BC SEMICOL

Then, first sets of the RHS are disjoint.

FIRST(α) = { GET_VALUE }
FIRST(β) ={ PRINT }
There is no nullable production for <ioStmt>, therefore no need to look at the FOLLOW(ioStmt).
[Recall that the rule A→ε is used to populate the parsing table entry T(A,a) for all symbols 'a' in FOLLOW(A)]
Hence, FIRST(<ioStmt>) = union of sets FIRST(α) and FIRST(β) ={ GET_VALUE, PRINT }

This contributes to the first set of <statement> (rule 14), which in turn will be used to verify (rule 13's LL(1) compatibility) whether the FIRST(<statement>) and FOLLOW(<statements>) are disjoint.

16. <var> → ID <whichId> | NUM | RNUM

To facilitate printing of true and false values, let us introduce a new rule for deriving boolean constants true and false.
<boolConstt> →TRUE | FALSE ...... 16(a)
Also, we will introduce a new nonterminal <var_id_num> as follws
<var_id_num> → ID <whichId> | NUM | RNUM ....16(b)
There we modify the rules for <var> as follows

<var> → <var_id_num> | <boolConstt> ....16(c)

Also, we will need to modify rule number 17, to incorporate printing of an array element indexed by positive integer (say a[4]) as well.

FIRST sets are {ID}, {NUM}, {RNUM} and {TRUE, FALSE} respectively for all the FOUR production rules for the nonterminal symbol <var> . FIRST sets are disjoint as above.
There is no nullable production.
Hence the rules abouve conform to LL(1) property.
FIRST(<var>) = {ID, NUM, RNUM, TRUE, FALSE }

17. <whichId>→ SQBO ID SQBC | ε

Here, I am modifying the rule to access array elements by a statically available integer number as well. The new rules are
<whichId>→ SQBO ID SQBC | SQBO NUM SQBC | ε

This can be best written as

<whichId>→ SQBO <index> SQBC | ε

where <index> is defined in rule 23 as <index>→ NUM | ID

FIRST(α) ={SQBO} for α being the RHS of the first rule of <whichID>
As there is a rule <whichID>→ ε, we need to see if any terminal symbol in FIRST(α) is in FOLLOW(<whichId>).
Computing FOLLOW(<whichID>) = FOLLOW(<var>) …………from 16
= {BC} U …………from 15
Hence the rules for <whichID> conform to the LL(1) properties.
[Note: I am not specifying explicitly at each rule that LL(1) properties like no ambiguity, no left recursion, no left factoring needed, etc unless it is to be reported because of presence of these in the grammar rules.]

18. <simpleStmt> → <assignmentStmt> | <moduleReuseStmt>

FIRST(<assignmentStmt>) and FIRST(<moduleReuseStmt>) should be disjoint

{ID} ∩ {SQBO, USE} = φ ……………refer 19 and 24

Hence rules for the nonterminal <simpleStmt> conform to LL(1)
Also FIRST(<simpleStmt>) = {ID, SQBO, USE}

19. <assignmentStmt> → ID <whichStmt>

FIRST(<assignmentStmt>) = FIRST(ID <whichStmt>) = {ID}

20. <whichStmt> →<lvalueIDStmt> | <lvalueARRStmt>

FIRST(<lvalueIDStmt>) and FIRST(<lvalueARRStmt>) should be disjoint.
Refer 21 and 22 to see that the above rule conforms to LL(1) specifications

21. <lvalueIDStmt> → ASSIGNOP <expression> SEMICOL
FIRST(<lvalueIDStmt>) = {ASSIGNOP}

22. <lvalueARRStmt> → SQBO <index> SQBC ASSIGNOP <expression> SEMICOL
FIRST(<lvalueARRStmt>) = {SQBO}

23. <index> → NUM | ID
Conforms to LL(1) (trivial)

24. <moduleReuseStmt> →<optional> USE MODULE ID WITH PARAMETERS <idList>SEMICOL
FIRST(<moduleReuseStmt>) = FIRST(<optional>) U FOLLOW(<optional>)
={SQBO} U {USE} = {SQBO, USE}
Notice that the MODULE keyword remains here as per the updates.

25. <optional> → SQBO <idList> SQBC ASSIGNOP | ε
FIRST(SQBO <idList> SQBC ASSIGNOP) ∩ FOLLOW(<optional>) = φ
Therefore there is no need for modification.

26. <idList> → <idList> COMMA ID | ID
This requires left recursion elimination
<output_plist> → ID <N3>
<N3> → COMMA ID <N3> | ε

27. <expression> →<arithmeticExpr> | <booleanExpr>
This rule needs special care. With given original rules for <arithmeticExpr> and <booleanExpr>, we found that their FIRST sets were not disjoint and were same as { BO, ID, NUM, RNUM }. A special care is required to perform left factoring which is done by using a single non terminal for both expressions and the expressions are constructed by way of appropriate binding.

Let us redefine this rule (rule 27) as follows
<expression>→<arithmeticOrBooleanExpr> | <U> …………….new 27.1

Defining an expression <U> generated by using unary operators plus or minus (type 1)
<U> → MINUS BO <arithmeticExpr> BC | PLUS BO <arithmeticExpr> BC | MINUS <var_id_num> | PLUS <var_id_num>

which can be left factored as below
<U> → <unary_op> <new_NT>

The new non terminals are defined as follows

<new_NT> → BO <arithmeticExpr> BC | <var_id_num> …………….new 27.2
<unary_op> → PLUS | MINUS …………….new 27.3

Consider 27.1 and let us show that the rules for <expression> are LL(1) compatible.
FIRST(<arithmeticOrBooleanExpr>) ∩ FIRST(<U>) ....see the computation later
= **{ID, NUM, RNUM, BO, TRUE, FALSE}** ∩ {PLUS, MINUS}
because FIRST(<U>) = FIRST(<<unary_op>) = {PLUS,MINUS}

Consider 27.2 and let us show that the two rules are LL(1) compatible.
FIRST(BO <arithmeticExpr> BC) ∩ FIRST(<var_id_num>)
= {BO} ∩ {ID, NUM, RNUM} = ϕ
Rule 27.3 is trivially LL(1).

Now we generate a type 2 expression which can be either a
(i) simple arithmetic expression or
(ii) an expression containing one boolean expression having two arithmetic expressions combined using relational operators or
(iii) a combination of boolean expressions constructed by combining with AND and OR operators.
Let us observe the following grammar

<arithmeticOrBooleanExpr> → <arithmeticOrBooleanExpr> <logicalOp> <AnyTerm> | <AnyTerm>
<AnyTerm> → <AnyTerm> <relationalOp> <arithmeticExpr> | <arithmeticExpr>

While <AnyTerm> can either expand to generate a boolean expression in its most atomic form using relational operators or can generate an arithmetic expression. We try to introduce more atomicity in the construction of boolean expression using boolean constants true and false as below

<AnyTerm> → <boolConstt> **.....using 16 (a)**

The rules for <arithmeticOrBooleanExpr> and <AnyTerm> are left recursive, hence need modification.

We have now rules

<arithmeticOrBooleanExpr>→ <AnyTerm> <N7> **….27 a**
<N7> → <logicalOp> <AnyTerm> <N7> | ε **…27 b**
<AnyTerm>→ <arithmeticExpr> <N8> | <boolConstt> <N8> **…27 c**
<N8> → <relationalOp> <arithmeticExpr><N8> | ε **…27 d**

Rules **27 a** is not left recursive, and is LL(1) compatible.
The two Rules of **27 c** are having their RHSs FIRST sets disjoint as is described below.

FIRST(<arithmeticExpr>) ∩ FIRST(<boolConstt>)
= {ID, NUM, RNUM, BO} ∩ {TRUE, FALSE}
=ϕ

Verifying the LL(1) compatibility of **27 d**,

FOLLOW(<N8>) = FOLLOW(<AnyTerm>) = FIRST(<N7>) = FIRST(<logicalOps>)
= {AND, OR}
FIRST(**<relationalOp> <arithmeticExpr><N8>**) = {LE, LT, GE, GT, EQ, NE}
Both FOLLOW(<N8>) and FIRST(**<relationalOp> <arithmeticExpr><N8>**) are disjoint.
Hence 27 d is LL(1) compatible.

Now let us verify the LL(1) compatibility of 27 b,

FOLLOW(<N7>) = FOLLOW(**<arithmeticOrBooleanExpr>)**
= FOLLOW(<expression>)
= {SEMICOL} ...using rules 21 and 22

And FIRST(**<logicalOp> <AnyTerm> <N7>) = FIRST(<logicalOp>)**
= {AND, OR}
Both of these sets are disjoint.
Hence rules 27 a - d are LL(1) compatible.
We will remove all rules that start with <booleanExpr>. See below.

Now FIRST(<arithmeticOrBooleanExpr>) = FIRST(<AnyTerm>)
= FIRST(<arithmeticExpr>) U FIRST(<boolConstt>)
= {ID, NUM, RNUM, BO} U {TRUE, FALSE}
= {ID, NUM, RNUM, BO, TRUE, FALSE}

Consider the rule new 27.1 and check the FIRST sets of both of its RHS strings
FIRST(<arithmeticOrBooleanExpr>) and FIRST(<U>) are disjoint, hence new 27 rule is LL(1) compatible. It generates the expression which is either a simple arithmetic expression, or an expression that has negative expression, or generates a boolean expression as well without having the first sets creating trouble as was the case with the original rule 27.

28. <arithmeticExpr> → <arithmeticExpr> <op> <term>

29. <arithmeticExpr> → <term>

To facilitate the precedence of operators, we need to split the rule for <op> into two <op1> and <op2> (see new rules defined in 34)

Rule 28 becomes
<arithmeticExpr> → <arithmeticExpr> <op1> <term>

Above two rules(28 and 29) are used for left recursion elimination as <arithmeticExpr> is left recursive.
These become
<arithmeticExpr> → <term> <N4> ….29 a
<N4> → <op1> <term> <N4> | ε …29 b

FIRST(<arithmeticExpr>) = FIRST(<term>) = {ID, NUM, RNUM, BO } ….. from 31
FIRST(<op1> <term> <N4>) = FIRST(<op1>) = {PLUS, MINUS}

As <N4>→ ε is a production, we need to look at the FOLLOW(<N4>)
FOLLOW(<N4>) = FOLLOW(<arithmeticExpr>) = FIRST(<N8>) U {BC} ……from 27 c, 27 d and new 27.2
= FIRST(**<relationalOp> <arithmeticExpr><N8>**) U {BC}
= {LE, LT, GE, GT, EQ, NE, BC}
This shows that the rules 29 a and 29 b are LL(1) compatible

30. <term> →<term> <op> <factor>
31. <term> →<factor>

Define rule 30 as <term>→<term><op2><factor> …..new rule for replacing 30

Above two rules require left recursion elimination
<term> → <factor> <N5> ……31a
<N5> → <op2> <factor> <N5>| ε ……31b

FIRST(<term>) = FIRST(<factor>) = {ID, NUM, RNUM, BO } ……from 33
FIRST(<op2><factor><N5>) = FIRST(<op2>) = {MUL, DIV}
As <N5>→ ε is a production then we need to look at the FOLLOW(<N5>)
FOLLOW(<N5>) = FOLLOW(<term>
= FIRST(<N4>) ……..from 29a
= FIRST(<op1>)
= {PLUS, MINUS}
Hence 31a and 31b conform to LL(1).
32. <factor> →BO <arithmeticExpr> BC

33. <factor> →<var>
Here we will restrict atomic values only to those specified by new rule 16 (b) for <var_id_num>
Therefore, the new RHS of this rule becomes <var_id_num> as follows
<factor> →<var_id_num>
FIRST(<factor>) = FIRST(<var_id_num>) U (BO <arithmeticExpr> BC)
= {ID, NUM, RNUM, BO} …..from 16

34. <op> → PLUS | MINUS | MUL | DIV

This rule needs split to facilitate precedence of operators

Instead of <op> , we have two new non terminals defined as

<op1> and <op2>

<op1>→PLUS|MINUS ……34a

<op2>→MUL|DIV …….34b

Both <op1> and <op2> are LL(1)

35. ~~<booleanExpr> →<booleanExpr> <logicalOp> <booleanExpr>~~ See the description above in 27

36. <logicalOp> →AND | OR

FIRST sets of the RHS are disjoint.

~~37. <booleanExpr> →<arithmeticExpr> <relationalOp> <arithmeticExpr>~~ refer 27

~~38. <booleanExpr> → BO <booleanExpr> BC~~ to incorporate this we have 27.1

39. <relationalOp> → LT | LE | GT | GE | EQ | NE

FIRST sets of the RHS are disjoint.

40. <declareStmt> → DECLARE <idList> COLON <dataType> SEMICOL

FIRST(<declareStmt>) = {DECLARE}

41. <condionalStmt> →SWITCH BO ID BC START <caseStmts><default> END

FIRST(<conditionalStmt>) ={ SWITCH}

42. <caseStmt> →CASE <value> COLON <statements> BREAK SEMICOL <caseStmt>

This rule is modified to facilitate any number of case statements (essentially one or more). We introduce a new nonterminal <caseStmts>. We modify the nonterminal in rule 41, while change is reflected using red color in rule 41.

The rules become

<caseStmts> → CASE <value> COLON <statements> BREAK SEMICOL <N9>

<N9> → CASE <value> COLON <statements> BREAK SEMICOL <N9> | ε

Here FIRST(<caseStmts>)={CASE}

As <N9> is a nullable production

FIRST (CASE <value> COLON <statements> BREAK SEMICOL <N9>) and FOLOW(<N9>) should be disjoint.

FOLLOW(<N9>) = FOLLOW(<caseStmts>) = FIRST(<default>) = {DEFAULT}

The rules therefore are LL(1) compatible.

43. <value> →NUM | TRUE | FALSE

FIRST(<value>) = {NUM, TRUE, FALSE}

The FIRST sets of the RHS of the three rules are disjoint.

44. <default> →DEFAULT COLON <statements> BREAK SEMICOL | ∈

FIRST(DEFAULT COLON <statements> BREAK SEMICOL) = {DEFAULT}

FOLLOW(<default>) = {END}

Hence both rules are LL(1) compatible.

45. <iterativeStmt> →FOR BO ID IN <range> BC START <statements> END |
WHILE BO <arithmeticOrBooleanExpr> BC START <statements> END

In While construct, the boolean expression construct is replaced with the non-terminal <arithmeticOrBooleanExpr>. It is considered syntactically correct to receive even an arithmetic expression here, based on the newly defined grammar rules (27 a-d). Hence your parser will not report error on while(a+b-c) ...kind of statements. However, later the error of this type will be detected by the type checker module of semantic analysis phase. Similarly, an expression a+ (b<c) is also syntactically correct and will be reported as type error later.

FIRST(<iterativeStmt>) = {FOR, WHILE}

The FIRST sets of the RHS of both rules are disjoint, hence LL(1)

46. <range> →NUM RANGEOP NUM

FIRST(<range>) = {NUM}

****

Note: New nonterminal symbols used to modify the grammar are as follows

<N1>
<N2>
<N3>
<N4>
<N5>
<N7>
<N8>
<N9>
<caseStmts> (in place of <caseStmt>)

<op1>
<op2>
<arithmeticOrBooleanExpr>
<AnyTerm>

NOTE: Students are advised to verify the grammar on their own. This document is provided only as a support. Ensure that the features of the language are preserved even after the modifications. please inform me if any discrepancy exists in rules in this document.

****