Beyens Ziad
Nougba Hamza

Über-FORTRAN
Introduction to language theory and compiling
Project – Part 2

1.a) First, we check if there is any unreachable and unproductive variable.
Unproductive algorithm:

i	V(i)
0	
1	{Vars, VarList, Code, ExprArith, Op, BinOp, Comp}
2	{Vars, VarList, Code, ExprArith, Op, BinOp, Comp, Program, Assign, SimpleCond, Do, Read, ExpList}
3	{Vars, VarList, Code, ExprArith, Op, BinOp, Comp, Program, Assign, SimpleCond, Do, Read, ExpList, Instruction, Cond, Print}
4	{Vars, VarList, Code, ExprArith, Op, BinOp, Comp, Program, Assign, SimpleCond, Do, Read, ExpList, Instruction, Cond, Print}

Unreachable algorithm :

i	V(i)
0	{Program}
1	{Program, Vars, Code}
2	{Program, Vars, Code, VarList, Instruction}
3	{Program, Vars, Code, VarList, Instruction, Assign, If, Do, Print, Read}
4	{Program, Vars, Code, VarList, Instruction, Assign, If, Do, Print, Read, ExprArith, Cond, ExpList}
5	{Program, Vars, Code, VarList, Instruction, Assign, If, Do, Print, Read, ExprArith, Cond, ExpList, BinOp, SimpleCond, Op}
6	{Program, Vars, Code, VarList, Instruction, Assign, If, Do, Print, Read, ExprArith, Cond, ExpList, BinOp, SimpleCond, Op, Comp}
7	{Program, Vars, Code, VarList, Instruction, Assign, If, Do, Print, Read, ExprArith, Cond, ExpList, BinOp, SimpleCond, Op, Comp}

Belong to the previous algorithms, no variable has to be removed.

1.b) Next, we have to modify the grammar in order to remove its ambiguousness.
Rules 14-22 are replaced by :

P		
<ExprArith>	→	<ExprArith><Op+-><ExprArith_b>
	→	<ExprArith_b>

<ExprArith_b>	→	<ExprArith_b><Op*/> <ExprArith_c>
	→	<ExprArith_c>
<ExprArith_c>	→	- <ExprArith_c>
	→	[VarName]
	→	[Number]
	→	(<ExprArith>)
<Op+->	→	+
	→	-
<Op*/>	→	*
	→	/

Rules 25-27 are replaced by :

P		
<Cond>	→	<Cond> .OR. <Cond_b>
	→	<Cond_b>
<Cond_b>	→	<Cond_b> .AND. <Cond_c>
	→	<Cond_c>
<Cond_c>	→	.NOT. <SimpleCond>
	→	<SimpleCond>

1.c) Here is the final grammar after removing left-recursion and applying factorisation.

#	P		
[1]	<Program>	→	PROGRAM [ProgName] [EndLine] <Vars> <Code> END
[2]	<Vars>	→	INTEGER <VarList> [EndLine]
[3]		→	ε
[4]	<VarList>	→	[VarName] <VarList_next>
[5]	<VarList_next>	→	, <VarList>
[6]		→	ε
[7]	<Code>	→	<Instruction> [EndLine] <Code>
[8]		→	ε
[9]	<Instruction>	→	<Assign>
[10]		→	<If>
[11]		→	<Do>
[12]		→	<Print>
[13]		→	<Read>
[14]	<Assign>	→	[VarName] = <ExprArith>
[15]	<ExprArith>	→	<ExprArith_b><ExprArith'>
[16]	<ExprArith'>	→	<Op+-> <ExprArith_b><ExprArith'>
[17]		→	ε
[18]	<ExprArith_b>	→	<ExprArith_c><ExprArith_b'>
[19]	<ExprArith_b’>	→	<Op*/> <ExprArith_c> <ExprArith_b'>
[20]		→	ε
[21]	<ExprArith_c>	→	- <ExprArith_c>
[22]		→	[VarName]
[23]		→	[Number]
[24]		→	(<ExprArith>)
[25]	<Op+->	→	+

[26]		→	-
[27]	<Op*/>	→	*
[28]		→	/
[29]	<If>	→	IF (<Cond>) THEN [EndLine] <Code><If-next>
[30]	<If-next>	→	ENDIF
[31]		→	ELSE [EndLine] <Code> ENDIF
[32]	<Cond>	→	<Cond_b> <Cond'>
[33]	<Cond'>	→	.OR. <Cond_b> <Cond'>
[34]		→	ε
[35]	<Cond_b>	→	<Cond_c> <Cond_b'>
[36]	<Cond_b'>	→	.AND. <Cond_c> <Cond_b'>
[37]		→	ε
[38]	<Cond_c>	→	.NOT. <SimpleCond>
[39]		→	<SimpleCond>
[40]	<SimpleCond>	→	<ExprArith><Comp> <ExprArith>
[41]	<Comp>	→	.EQ.
[42]		→	.GE.
[43]		→	.GT.
[44]		→	.LE.
[45]		→	.LT.
[46]		→	.NE.
[47]	<Do>	→	DO [VarName] = [Number], [Number] [EndLine] <Code> ENDDO
[48]	<Print>	→	PRINT*, <ExpList>
[49]	<Read>	→	READ*, <VarList>
[50]	<ExpList>	→	<ExprArith><ExpList_next>
[51]	<ExpList_next>	→	, <ExpList>
[52]		→	ε

2) Next, we have to calculate the First of all the variables and the Follow of the variables for which the First contains ε.

V	First(V)	Follow(V)
<Program>	PROGRAM	
<Vars>	INTEGER ε	[VarName] IF DO PRINT* READ* END
<VarList>	[VarName]	[EndLine]
<VarList_next>	, ε	[EndLine]
<Code>	[VarName] IF DO PRINT* READ* ε	END ENDIF ELSE ENDDO
<Instruction>	[VarName] IF DO PRINT* READ*	[EndLine]
<Assign>	[VarName]	[EndLine]
<ExprArith>	- [VarName] [Number] (	[EndLine]) .AND. .OR. , .EQ. .GE. .GT. .LE. .LT. .NE. **?**
<ExprArith'>	+ - ε	[EndLine]) .AND. .OR. , .EQ. .GE. .GT. .LE. .LT. .NE.
<ExprArith_b>	- [VarName] [Number] (	+ - [EndLine]) .AND. .OR. , .EQ. .GE. .GT. .LE. .LT. .NE.
<ExprArith_b'>	* / ε	+ - [EndLine]) .AND. .OR. , .EQ. .GE. .GT. .LE. .LT. .NE.
<ExprArith_c>	- [VarName] [Number] (	* / + - [EndLine]) .AND. .OR. , .EQ. .GE. .GT. .LE. .LT. .NE.

<Op+->	+ -	- [VarName] [Number] (
<Op*/>	* /	- [VarName] [Number] (
<If>	IF	[EndLine]
<If-next>	ENDIF ELSE	[EndLine]
<Cond>	.NOT. - [VarName] [Number] (	)
<Cond'>	.OR. ε	)
<Cond_b>	.NOT. - [VarName] [Number] (	.OR.)
<Cond_b'>	.AND. ε	.OR.)
<Cond_c>	.NOT. - [VarName] [Number] (	.AND. .OR.)
<SimpleCond>	- [VarName] [Number] (	.AND. .OR.)
<Comp>	.EQ. .GE. .GT. .LE. .LT. .NE.	- [VarName] [Number] (
<Do>	DO	[EndLine]
<Print>	PRINT*	[EndLine]
<Read>	READ*	[EndLine]
<ExpList>	- [VarName] [Number] (	[EndLine]
<ExpList_next>	, ε	[EndLine]

Therefore, we have the action table (excel).
We can see that there is not conflict : LL(1) parser.

3) Finally, a recursive descent LL(1) parser for this grammar has been written.
The leftmost derivation of the input string is the output of the parser if the syntax is correct.
Otherwise, an error message appear with the number of rules causing the error.