Projects for Compilers

1. Project One: Lexical Analysis (Required)

(1) Directions

Implement a transition-diagram-based lexical analysis for the programming language TINY.

(2) Outputs

- Source code (implemented in Java or C or C++ programming languages)
- A project report

Structure of the project report

1. Objectives
   (1) Regular expressions
   (2) Transition diagram based lexical analyzer
   (3) Error handling

2. Program Designing
   (1) Lexical specification for the programming language
   (2) Token scheme
   (3) Structure of the program

3. Test Cases and Test results
   (1) For correct input, the program should print the token sequence
   (2) For error input, the program should report the errors
   Test cases should cover the ranges as large as possible.

(3) Lexical Specification of Programming Language TINY

- **Tokens**

type	lexeme	Token Name (code)	Attribute
keywords	if	**if** (261)	-
	then	**then** (262)	-
	else	**else** (263)	-
	end	**end** (264)	-
	repeat	**repeat** (265)	-
	until	**until** (266)	-
	read	**read** (267)	-
	write	**write** (268)	-
relation operators	<	**relop** (270)	LT (271)
	<=	**relop**	LE (272)
	==	**relop**	EQ (273)
	<>	**relop**	NE (274)
	>	**relop**	GT (275)
	>=	**relop**	GE (276)
arithmetical operators	+	**addop**(280)	ADD (281)
	-	**addop**(280)	MINUS (282)
	*	**mulop**(285)	MUL(286)
	/	**mulop**(285)	DIV(287)
	(	**(** (294)	-
	)	**)** (295)	-
assignment	:=	**:=** (296)	-
Segment	;	**;** (297)	-
numbers	Such as *12342*	**num** (298)	Symbol Table Entry
identifiers	Such as *student1*	**id** (299)	Symbol Table Entry
white spaces (ws)	*blank*, *tab*, and *newline*	-	-
Comments	{bla, bala, bala }	-	-

Note: all white spaces and comments should be removed.

- **Regular Definitions**

```
delim→  [ \t\n]
ws    →  delim+
letter→ [A-Za-z]
digit →  [0-9]
id    →  letter(letter|digit)*
num   →  digit+
```

- **Example code of TINY**

```
{ Sample program in TINY language - computes factorial}
read x; { input an integer }
if x > 0 then { do not compute if x <= 0 }
   fact := 1;
   repeat
      fact := fact * x;
      x := x - 1
   until x = 0;
   write fact { output factorial of x }
end
```

2. Project Two: Syntax Analysis (Required)

(1) Directions

Use either LL(1) or LR(1) techniques to implement a syntax analyzer for the programming language TINY. LL (1) can implemented as either Recursive-Descent parsing or nonrecursive Predictive Parsing.

(2) Outputs

- Source code (implemented in Java or C or C++ programming languages)
- A project report

Structure of the project report

1. Objectives
   (1) Context-free grammar
   (2) LL(1) or LR(1) parsing
   (3) Error handling

2. Program Designing
   (1) Context-free grammar for the programming language
   (2) FIRST and FOLLOW for LL(1) (Sets of Items for LR(1))
   (3) Structure of the program

3. Test Cases and Test results
   (1) For correct input, the program should print a syntax tree
   (2) For error input, the program should report the errors
   Test cases should cover the ranges as large as possible.

(3) Context-free of Programming Language TINY

1) *program* → *stmt-sequence*
2) *stmt-sequence* → *stmt-sequence* **;** *statement* | *statement*
3) *statement* → *if-stmt* | *repeat-stmt* | *assign-stmt* | *read-stmt* | *write-stmt*
4) *if-stmt* → **if** *exp* **then** *stmt-sequence* **end**
   | **if** *exp* **then** *stmt-sequence* **else** *stmt-sequence* **end**
5) *repeat-stmt* → **repeat** *stmt-sequence* **until** *exp*
6) *assign-stmt* → **id :=** *exp*
7) *read-stmt* → **read id**
8) *write-stmt* → **write** *exp*
9) *exp* → *simple-exp* **relop** *simple-exp* | *simple-exp*
10) *simple-exp* → *simple-exp* **addop** *term* | *term*
11) *term* → *term* **mulop** *factor* | *factor*
12) *factor* → **(***exp***)** | **num** | **id**

LL(1) Grammar of TINY

1) *program* → *stmt-sequence*
2) *stmt-sequence* → *statement stmt-sequence-p*
3) *stmt-sequence-p* → **;** *statement stmt-sequence-p* | ε
4) *statement* → *if-stmt* | *repeat-stmt* | *assign-stmt* | *read-stmt* | *write-stmt*
5) *if-stmt* → **if** *exp* **then** *stmt-sequence if-rear*
6) *if-rear* → **end | else** *stmt-sequence* **end**
7) *repeat-stmt* → **repeat** *stmt-sequence* **until** *exp*
8) *assign-stmt* → **id :=** *exp*
9) *read-stmt* → **read id**
10) *write-stmt* → **write** *exp*
11) *exp* → *simple-exp exp-rear*
12) *exp-rear* → **relop** *simple-exp* | ε
13) *simple-exp* → *term simple-exp-p*
14) *simple-exp-p* →**addop** *term simple-exp-p* | ε
15) *term* → *factor term-p*
16) *term-p* → **mulop** *factor term-p* | ε
17) *factor* → **(***exp***)** | **num** | **id**

FIRST(*program*) = FRIST (*stmt-sequence*) = {**if repeat id read write**}

FIRST(*stmt-sequence*) = FIRST(*statement stmt-sequence-p*) = {**if repeat id read write**}

FIRST(*stmt-sequence-p*) = { **;** ε}

FIRST(*statement*) = FIRST (*if-stmt*) U FIRST (*repeat-stmt*) U FIRST (*assign-stmt*) U FIRST (*read-stmt*) U FIRST (*write-stmt*) = {**if repeat id read write**}

FIRST(*if-stmt*) = {**if**}

FIRST(*if-rear*) = FIRST(**end**) U FIRST(**else** *stmt-sequence* **end**) ={**end else**}

FIRST(*repeat-stmt*) = {**repeat**}

FIRST(*assign-stmt*) = {**id**}

FIRST(*read-stmt*) = {**read**}

FIRST(*write-stmt*) = {**write**}

FIRST(*exp*) = FIRST(*simple-exp exp-rear*)={**(num id**}

FIRST(*exp-rear*) = {**relop** ε}

FIRST(*simple-exp*) = FIRST(*term simple-exp-p*) = {**(num id**}

FIRST(*simple-exp-p*) = {**addop** ε}

FIRST(*term*) = FIRST(*factor term-p*) = {**(num id**}

FIRST(*term-p*) = {**mulop** ε}

FIRST(*factor*) = FIRST (**(***exp***)**) U FIRST (**num**) U FIRST (**id**) ={**(num id**}

FOLLOW(*program*) ={$}

FOLLOW (*stmt-sequence*)={$ **end else until**}

FOLLOW (*stmt-sequence-p*)={$ **end else until**}

FOLLOW (*statement*) ={$ **; end else until**}

FOLLOW (*if-stmt*)={$ **; end else until**}

FOLLOW (*if-rear*)= {$ **; end else until**}

FOLLOW (*repeat-stmt*)={$ **; end else until**}

FOLLOW (*assign-stmt*)={$ **; end else until**}

FOLLOW (*read-stmt*)={$ **; end else until**}

FOLLOW (*write-stmt*)={$ **; end else until**}

FOLLOW (*exp*)={ $ **;) end else until then**}

FOLLOW (*exp-rear*)={ $ **;) end else until then**}

FOLLOW (*simple-exp*)={ **relop** $ **;) end else until then** }

FOLLOW (*simple-exp-p*)={ **relop** $ **;) end else until then** }

FOLLOW (*term*)={ **addop relop** $ **;) end else until then** }

FOLLOW (*term-p*)={**addop relop** $ **;) end else until then**}

FOLLOW (*factor*)={**addop mulop relop** $ **;) end else until then**}

FIRST(*stmt-sequence-p*)∩FOLLOW (*stmt-sequence-p*)={}

FIRST(*exp-rear*)∩FOLLOW(*exp-rear*)={}

FIRST(*simple-exp-p*) ∩FOLLOW (*simple-exp-p*)={}

FIRST(*term-p*)∩FOLLOW (*term-p*) ={}

The grammar is an LL(1) grammar.