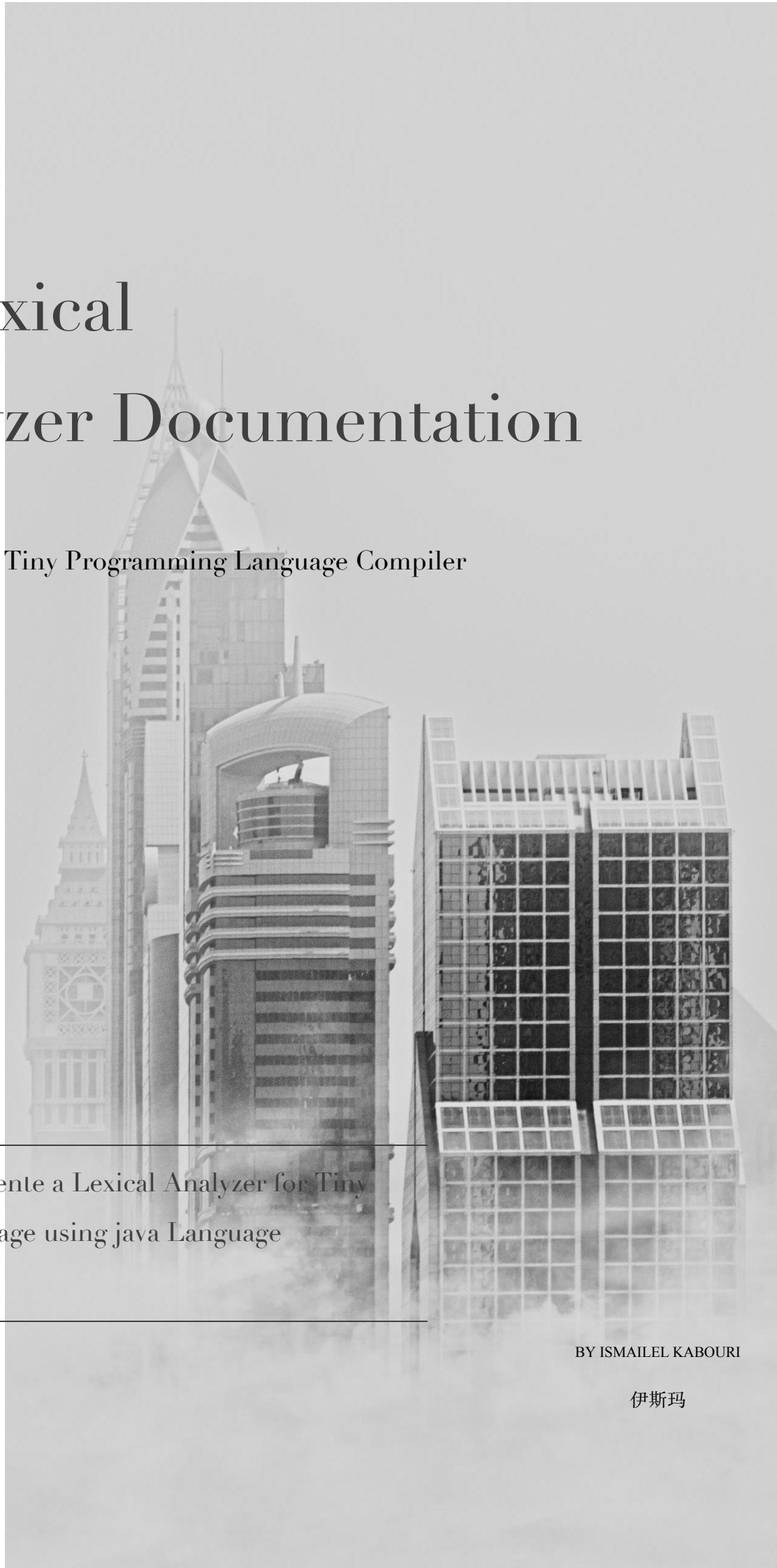
Tiny Lexical Analyzer Documentation

Tiny Programming Language Compiler



Design And Implemente a Lexical Analyzer for Tiny programming Language using java Language

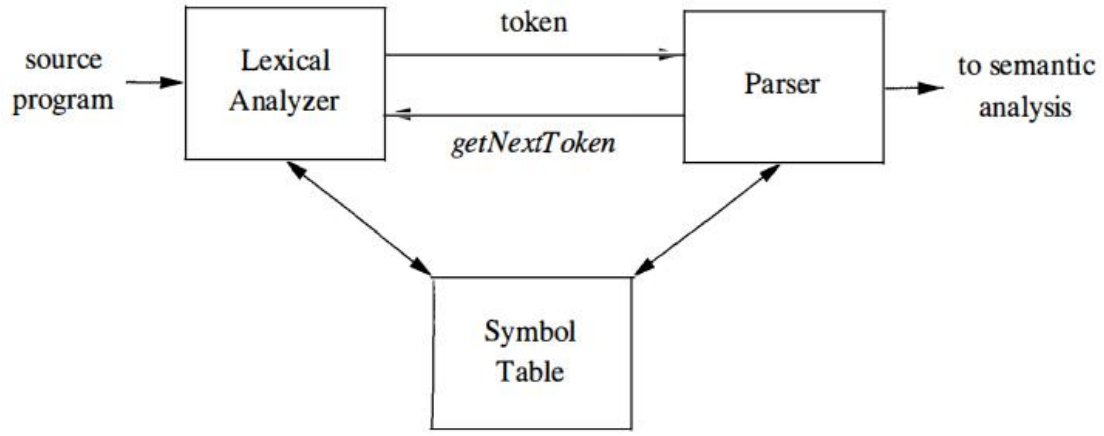
BY ISMAILEL KABOURI

伊斯玛

Catalog

1 - Introduction

At the heart of every compiler lies the crucial role of the lexical analyzer which is read the input characters of the source program, group them into lexemes, and produce as output a sequence of tokens for each lexeme in the source program. The stream of tokens is sent to the parser for syntax analysis. It is common for the lexical analyzer to interact with the symbol table as well. When the lexical analyzer discovers a lexeme constituting an identifier, it needs to enter that lexeme into the symbol table



Since the lexical analyzer is the part of the compiler that reads the source text, it may perform certain other tasks besides identification of lexemes. One such task is stripping out comments and whitespace (blank, newline, tab, and perhaps other characters that are used to separate tokens in the input). Another task is correlating error messages generated by the compiler with the source program.

Therefor lexical analyzers consist of

-*Scanning* consists of the simple processes that do not require tokenization of the input, such as deletion of comments and compaction of consecutive whitespace characters into one.

- *Lexical analysis* proper is the more complex portion, where the scanner produces the sequence of tokens as output.

2 - Tiny Programing Language

1 - Introduction

TINY is a simplified programming language often used for educational purposes and compiler construction projects. Its minimalist syntax and structure make it an ideal platform for learning fundamental concepts such as lexical analysis, parsing, and code generation

A program in TINY consists of a set of functions (any number of functions and ends with a main function), each function is a sequence of statements including (declaration, assignment, write, read, if, repeat, function, comment, …) each statement consists of (number, string, identifier, expression, condition, …).

2 - Tiny Language Structure and specifications

- *Number*: any sequence of digits (e.g. 123 | 554 | 0.23 | …)
- ***String*:** starts with double quotes followed by any combination of characters and digits then ends with double quotes (e.g. “Hello” | “2nd + 3rd” | …)
- ***Reserved*_Keywords:** int | float | string | read | write | repeat | until | if | elseif | else | then | return | endl
- ***Comment_Statement:*** starts with /* followed by any combination of characters and digits then ends with */ (e.g. /* comment*/ | …)
- ***Identifiers:*** starts with letter then any combination of letters and digits. (e.g. x | val | counter1 | str1 | s2 | …)
- ***Function_Call:*** *s*tarts with Identifier then left bracket “(“ followed by zero or more Identifier separated by “,” and ends with right bracket “)”. (e.g. sum(a,b) | factorial(c) | …)

- **Arithmetic_Operator:** any arithmetic operation (+ | - | * | /)
- **Assignment_Statement:** starts with Identifier then assignment operator ":=" followed by Expression (e.g. x := 1 | y:= 2+3 | …)
- **Datatype:** set of reserved keywords (int, float, string)
- **Declaration_Statement:** starts with Datatype then one or more identifiers (assignment statement might exist) separated by coma and ends with semi-colon. (e.g. int x; | …)
- ***Write_Statement:*** starts with reserved keyword "write" followed by an Expression or endl and ends with semi-colon (e.g. write x; | write 5; | write 3+5; | write "Hello World"; | …)
- ***Read_Statement:*** starts with reserved keyword "read" followed by an Identifier and ends with semi-colon (e.g. read x; | …)
- ***Return_Statement:*** starts with reserved keyword "return" followed by Expression then ends with semi-colon (e.g. return a+b; | return 5; | return "Hi"; | …)
- ***Condition_Operator:*** (less than "<" | greater than ">" | is equal "=" | not equal "<>")
- ***Condition:*** starts with Identifier then Condition_Operator then Term (e.g. z1 <> 10)
- ***Boolean_Operator:*** AND operator "&&" and OR operator "||"
- ***Condition_Statement:*** starts with Condition followed by zero or more Boolean_Operator and Condition (e.g. x < 5 && x > 1)
- ***If_Statement:*** starts with reserved keyword "if" followed by Condition_Statement then reserved keyword "then" followed by set of Statements (i.e. any type of statement: write, read, assignment, declaration, …) then Else_If_Statment or Else_Statment or reserved keyword "end"
- ***Else_If_Statement:*** same as if statement but starts with reserved keyword *"elseif"*
- ***Else_Statement:*** starts with reserved keyword "else" followed by a set of Statements then ends with reserved keyword "end"

- ***Repeat_Statement:*** starts with reserved keyword "repeat" followed by a set of Statements then reserved keyword "until" followed by Condition_Statement

- ***Parameter:*** starts with Datatype followed by Identifier (e.g. int x)

- ***Function_Declaration:*** starts with Datatype followed by FunctionName followed by "(" then zero or more Parameter separated by "," then ")" (e.g. int sum(int a, int b) | …)

- ***Function_Body:*** starts with curly bracket "{" then a set of Statements followed by Return_Statement and ends with "}"

- ***Function_Statement:*** starts with Function_Declaration followed by Function_Body

- ***Main_Function:*** starts with Datatype followed by reserved keyword "main" then "()" followed by Function_Body

- ***Program:*** has zero or more Function_Statement followed by Main_Function

3 -Tiny Language Token

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 }	-	-

4 - TINY Language Regular Expressions

```
digit ::= 0|1|2|3|4……|9
letter ::= [a-z][A-Z]
Number ::= digit.?digit
String ::= “(letter|digit)*”
Reserved_Keywords ::= int | float | string | read | write | repeat
| until | if | elseif | else | then | return | endl
Comment ::= /*String*\/
Identifier ::= letter (letter | digit)*
Arithmatic_Operator ::= + | - | * | /
Datatype ::= int | float | string
Condition_Operator ::= < | > | = | <>
Boolean_Operator ::= && | \|\|
```

3 - Design the Lexical Analyzer

3 -1 Regular Expressions to NFA

Each regular expression is first converted into an NFA (Non-Detirministic Finite Automaton) . This step involves creating states and transitions for each character in the regular expression, we can do that by hand or use some tools which can generate the automaton for us like “Gragpize “ withe “Dot Language” which accept states and thier transitions, and we can add some more attributes for good appearence like “color” or “ style”.Even we get the NFA representation it still a littile dificule during implementation, coz the NFA can make the “Epsilon” transitions to navigate between states, and that not effient at all we need to defined unique pathe from start state to accepting state for each regex, to avoid confusion during scanning and analysing of lexemes .

3-2 NFA to DFA

The NFA is then converted into a DFA (Detirministic Finite Automaton)which has a unique transition for each character in the alphabet from each state. and again we can do that by hand by following subset construction algorithm, which consider state of the constructed DFA corresponds to a set of NFA states. And remove “epsilon” tarnsition, in DFA each token has a unique path from start state to accepting state, then minimize DFA to make it more simple and easy to implement. Or by using some tools like “Graphize “ and “Dot language “ which generate well done DFA representation, with nice appearence.

3-3 DFA for Tiny Language

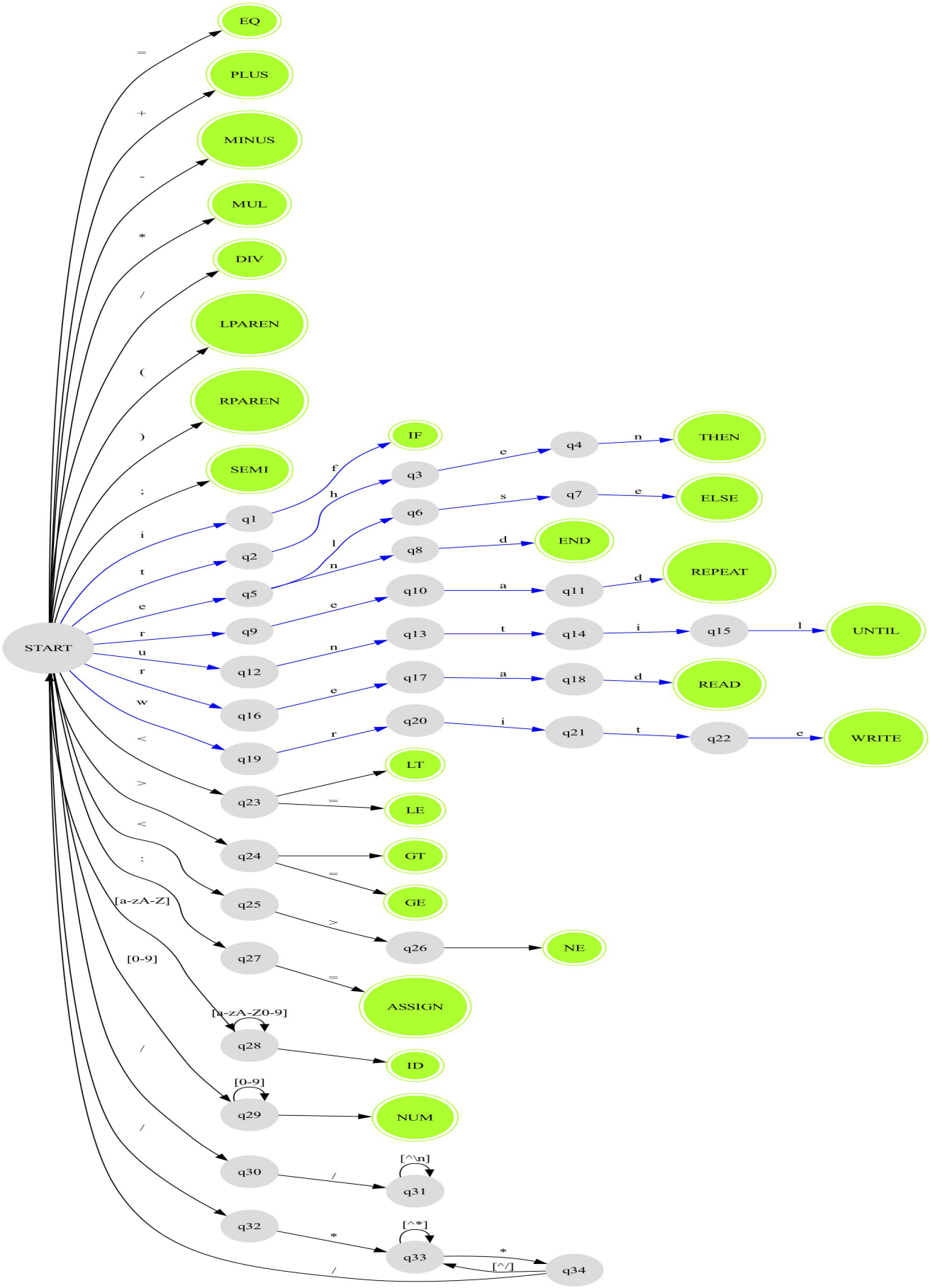
We use “Graphize” and Dot Language to generate the DFA for comman Tiny lexemes.

Dot Language Structure

```
 digraph DFA {

size = 10.8;
node["Attributes"] // which represent the final states .
node["Attributes"] // which represent other states.

s0["Attributes"] // initail state

// Transitions goeas here

// Keywords
.
.
.

// Identifiers
.
.
.
```

To generate the DFA graph, use Graphviz with the following command:

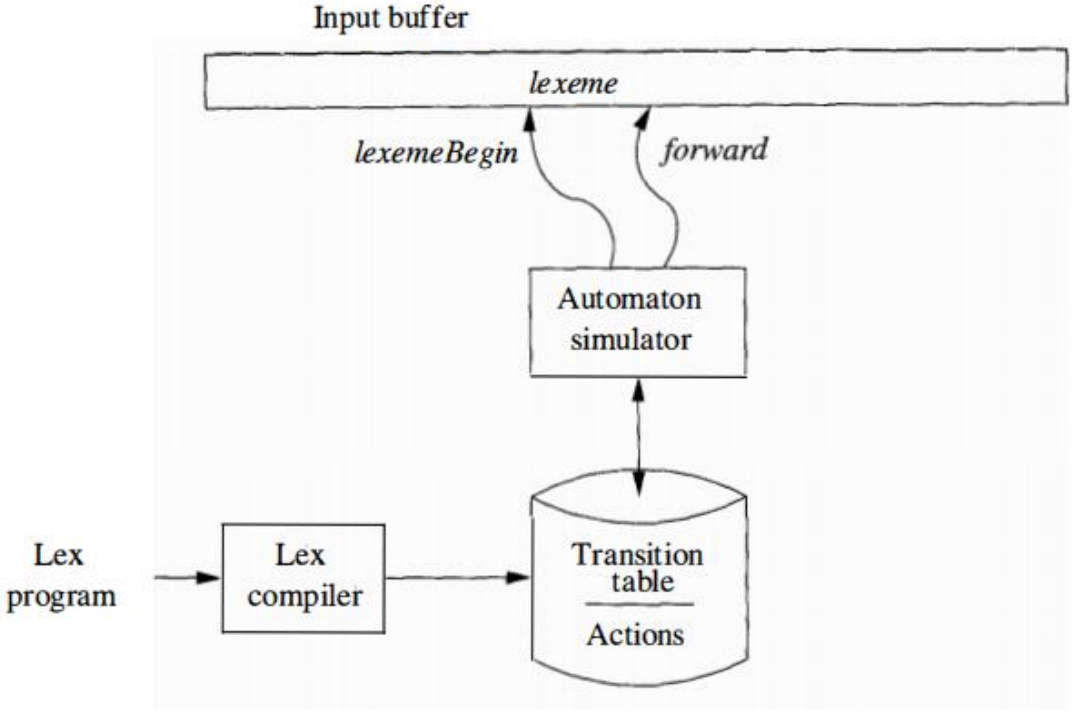
```
$$  -->  dot -Tpng -Gdpi=1000 dfa.dot -o dfa.png
```


START
EQ
=
PLUS
+
MINUS
-
MUL
*
DIV
/
LPAREN
(
RPAREN
)
SEMI
;
q1
i
q2
t
q5
e
q9
r
q12
u
r
q16
w
q19
<
q23
>
<
q24
:
q25
[a-zA-Z]
q27
[0-9]
[a-zA-Z0-9]
q28
/
ID
[0-9]
q29
/
NUM
q30
/
q31
[^\n]
q32
*
q33
[^*]
*
/
q34
[^/]
IF
f
q3
h
q6
l
q8
n
d
END
e
q4
n
THEN
s
q7
e
ELSE
q10
e
a
q11
d
REPEAT
q13
n
t
q14
i
q15
l
UNTIL
q17
e
a
q18
d
READ
q20
r
i
q21
t
q22
e
WRITE
LT
=
LE
GT
=
GE
>
q26
NE
=
ASSIGN

3-4 DFA implementation with Jflax.

Implementing DFA by hand can be chalenging specailly when we dail with complex language with wide range of tokens and specifications.To avoid complexity we can use a program to automatically transform a set of regular expressions into code for a scanner. Such a program is known as a scanner generator. Us u know I choose java language to do the implementation of that lexer. The appropriate generator for that is “Jflex” which is a lexical analyser generator for Java written in Java.It takes as input a specification with a set of regular expressions and corresponding actions. It generates a program (a lexer) that reads input, matches the input against the regular expressions in the spec file, and runs the corresponding action if a regular expression matched.

- **The Structure of the Generated Analyzer**



Jflex structure

```
// some import stm...
 %%------------------------------------------------------------------------------------------
 %class className
  %unicode
  %public
 %type Token
 %{-----------------------------------------------------
                                 // java code
 %}-----------------------------------------------------
 %state COMMENT                 // decleration section
 %% ------------------------------------------------------------------------------------------
<YYINITIAL> {
 "if"                           { return new Token(sym.IF, yytext()); }
 "else"                         { return new Token(sym.ELSE, yytext()); }
 "read"                         { return new Token(sym.READ, yytext()); }
 "write"                        { return new Token(sym.WRITE, yytext()); }
 "<"                            { return new Token(sym.LT, yytext()); }
 ">"                            { return new Token(sym.GT,  yytext()); }
 "+"                            { return new Token(sym.PLUS, yytext()); }
 "-"                            { return new Token(sym.MINUS, yytext()); }
  [a-zA-Z][a-zA-Z0-9]*          { return new Token(sym.ID, yytext()); }
 "//"                           { yybegin(COMMENT); } { /* Ignore whitespaces */ }
<COMMENT> {
 \n                             { yybegin(YYINITIAL); newline(); }
  .                             { /* Ignore characters in single-line comments */ }
-------------------------------------------------------------------------------------------------------------------------------------------
```

4 Tiny language sample

4 - 1 Sample one

INPUTE	OUTPUT	
if x < 10 then read y; write y + 1; end	Token: <READ, 'read'> |Token: <ID, 'x'> |Token: <SEMI, ';'> |Token: <IF, 'if'> Token: <ID, 'x'> Token: <GT, '>'> Token: <NUM, '0'> Token: <THEN, 'then'> Token: <ID, 'fact'> Token: <ASSIGN, ':='> Token: <NUM, '1'> Token: <SEMI, ';'> Token: <REPEAT, 'repeat'> Token: <ID, 'fact'> Token: <ASSIGN, ':='> Token: <ID, 'fact'> Token: <MUL, '*'>	| Token: <ID, 'x'> | Token: <SEMI, ';'> | Token: <ID, 'x'> | Token: <ASSIGN, ':='> | Token: <ID, 'x'> | Token: <MINUS, '-'> | Token: <NUM, '1'> | Token: <UNTIL, 'until'> | Token: <ID, 'x'> | Token: <EQ, '='> | Token: <NUM, '0'> | Token: <SEMI, ';'> | Token: <WRITE, 'write'> | Token: <ID, 'fact'> | Token: <END, 'end'>

4-2 Sample tow

INPUTE	OUTPUT	

int main()
{
int x;
read x;
if x > 0
then
int fact := 1;
 	
repeat
fact := fact * x;
 	
x := x - 1;
until x = 0
 	
write fact;
end
return 0;
 	
}
	_____
	| Token: <ID, 'int'>
	|_____|
	| Token: <ID, 'main'>
	|_____|
	| Token: <LPAREN, '('>
	|_____|
	| Token: <RPAREN, ')'>
	|_____|
	| Token: <LBRACE, '{'>
	|_____|
	| Token: <ID, 'int'>
	|_____|
	| Token: <ID, 'x'>
	|_____|
	| Token: <SEMI, ';'>
	|_____|
	| Token: <READ, 'read'>
	|_____|
	| Token: <ID, 'x'>
	|_____|
	| Token: <SEMI, ';'>
	|_____|
	| Token: <IF, 'if'>
	|_____|
	| Token: <ID, 'x'>
	|_____|
	| Token: <GT, '>'>
	|_____|
	| Token: <NUM, '0'>
	|_____|
	| Token: <THEN, 'then'>
	|_____|
	| Token: <ID, 'int'>
	|_____|
	| Token: <ID, 'fact'>
	|_____|
	| Token: <ASSIGN, ':='>
	|_____|
	| Token: <NUM, '1'>
	|_____|
	| Token: <SEMI, ';'>
	|_____|
	| Token: <REPEAT, 'repeat'>
	|_____|
	| Token: <ID, 'fact'>
		| Token: <ASSIGN, ':='>
		|_____|
		| Token: <ID, 'fact'>
		|_____|
		| Token: <MUL, '*'>
		|_____|
		| Token: <ID, 'x'>
		|_____|
		| Token: <SEMI, ';'>
		|_____|
		| Token: <ID, 'x'>
		|_____|
		| Token: <ASSIGN, ':='>
		|_____|
		| Token: <ID, 'x'>
		|_____|
		| Token: <MINUS, '-'>
		|_____|
		| Token: <NUM, '1'>
		|_____|
		| Token: <SEMI, ';'>
		|_____|
		| Token: <UNTIL, 'until'>
		|_____|
		| Token: <ID, 'x'>
		|_____|
		| Token: <EQ, '='>
		|_____|
		| Token: <NUM, '0'>
		|_____|
		| Token: <WRITE, 'write'>
		|_____|
		| Token: <ID, 'fact'>
		|_____|
		| Token: <SEMI, ';'>
		|_____|
		| Token: <END, 'end'>
		|_____|
		| Token: <RETURN, 'return'>
		|_____|
		| Token: <NUM, '0'>
		|_____|
		| Token: <SEMI, ';'>
		|_____|
		| Token: <RBRACE, '}'>
		|_____|

5 Acknowlegement

I would like to express my gratitude to everyone who supported and guided me throughout the development of this project.

Special thanks to:

- Professors and Mentors: For their invaluable guidance and insightful feedback.
- Open Source Community: For providing tools like JFlex and resources that were crucial in building this lexical analyzer.

This project would not have been possible without the contributions and support from these individuals and communities.

6 - References

- https://www3.nd.edu/~dthain/compilerbook/

- https://www.jflex.de/manual.html

- https://a7medayman6.github.io/Tiny-Compiler/

- https://www.graphviz.org/

- Compiler Principles Techniques and Tools (2nd Edition) Book.

Made by : ISMAIL EL KABOURI
ID : 92115855E124
University: NJUST
Contructor: Kedong Yan
Location : China