3/2/24

LEX Tool

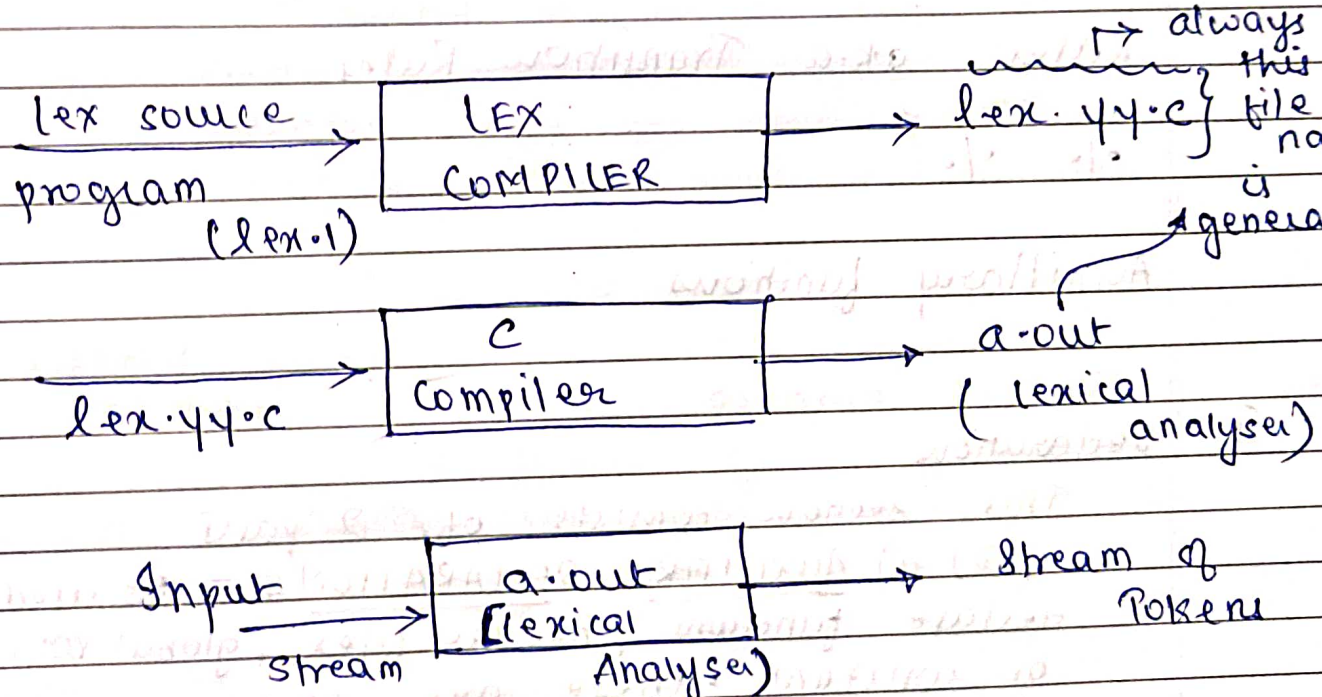
DATE 03/02/24

1) lexical Analyzer : scans the input stream & converts sequences of characters into TOKENS.

2) LEX is a tool used to generate a LEXICAL ANALYSER. [Lex Tool]

3) LEX Tool was initially developed by ERIC SCHMIDT & MIKE LESK.

4) The LEX Tool translates a SET OF REGULAR EXPRESSIONS specification [given as an input to any .l file] into transition diagram / finite automata & generates a code in a file called lex.yy.c



STRUCTURE OF LEX PROGRAMS

A LEX PROGRAM consists of three sections
a) Declarations
b) Rules
c) Auxillary functions

Each of the above section is separated by %.% delimiter

Format of program

Declarations

%.%

Rules a.k.a Transition Rules

%.%

Auxillary functions

I Declarations

This section consists of 2 parts

(AD) a) AUXILLARY DECLARATIONS - Is used to declare function, header files, global variables or constant. These are copied as such by LEX to the output of a lex.yy.c file. These instructions to C compiler & are not processed by the LEX Tool.

These instructions are enclosed within

%.{ .... %.}

(RD)

b) REGULAR DEFINITIONS :- lex allows use of short hands & extensions to regular expressions for the regular definitions

→ A Regular Definition in LEX is of the form.

D R { D is symbol representing Regular symbol }

Eg

```
%{
    #include <stdio.h>
    int global_variable;
%}
```

} AD.

number [0-9]+
op [-|+|*|/|^|=] } RD

II RULES

Rules in a LEX program consists of 2 parts
1) The pattern to be matched.
2) The corresponding action to be executed.

Eg

```
%%
{number}    {printf("Number");}
{op}        {printf("operator");}
%%
```

iii) AUXILIARY FUNCTIONS.

lex generates C code for the rules specified in RULES section & places this code into a single function called yylex()

yylex() :- Is a function of return type int.
* When invoked, it starts reading the input

yylex() continues scanning the input till one of the actions corresponding to a matched pattern executes a return statement

OR

Till the end of the input has been encountered.

S.O.

* In the case of console input, yylex() would wait for more input through the console. The user will have to input <ctrl+c> in the terminal to terminate yylex()

6/2/24

Syntax Analysis

Left Recursion

A grammar $G(V, T, P, S)$ is left recursive if it has a production in the form;

$$A \rightarrow A\alpha \mid \beta$$

Here, the leftmost variable of RHS is same as variable of its LHS.

A grammar containing a production having Left Recursion is called LEFT RECURSIVE GRAMMAR

Left Recursion is considered to be problematic for TOP DOWN PARSER

RULE TO REMOVE LEFT RECURSION

$$A \rightarrow A\alpha \mid \beta$$

Then Take a New variabe A', append it with each production (i.e β).

$$A \rightarrow \beta A'$$

Append it with 'α' part of A. & add ϵ

$$A' \rightarrow \alpha A' \mid \epsilon$$

Thus

$$A \rightarrow A\alpha \mid \beta \xrightarrow{\text{After removing left Recursion}} \begin{matrix} A \rightarrow \beta A' \\ A' \rightarrow \alpha A' \mid \epsilon \end{matrix}$$

This type of Recursion is also known as Immediate Left Recursion

eg: Remove left recursion for foll. grammar

1) $E \rightarrow E+T \mid T \mid F$

($E+T$: α; T, F: β)

Thus

$E \rightarrow TE' \mid FE'$

$E' \rightarrow +TE' \mid \epsilon$

2) $A \rightarrow AB \mid AC \mid aB \mid cd$ { Q. on Multiple Left Recursion }

$A \rightarrow aBA' \mid cdA'$

$A' \rightarrow BA' \mid CA' \mid \epsilon$

3) Consider the Left Recursion from the grammar

$E \rightarrow E+T \mid T$

$T \rightarrow T*F \mid F$

$F \rightarrow (E) \mid id$

Soln

$E \rightarrow TE'$

$E' \rightarrow +TE' \mid \epsilon$

$T \rightarrow FT'$

$T' \rightarrow *FT' \mid \epsilon$

$F \rightarrow (E) \mid id$

4) Eliminate left Recursion for the foll. grammar.

$S \rightarrow a \mid \wedge \mid (T)$

$T \rightarrow T, S \mid S$

Soln

$T \rightarrow ST'$

$T' \rightarrow ,ST' \mid \epsilon$

$S \rightarrow a \mid \wedge \mid (T)$

5) $E \rightarrow E(T) \mid T$

$T \rightarrow T(F) \mid F$

$F \rightarrow id$

Soln:

$E \rightarrow TE'$

$E' \rightarrow (T)E' \mid \epsilon$

$T \rightarrow FT'$

$T' \rightarrow (F)T' \mid \epsilon$

$F \rightarrow id$

6) $A \rightarrow Ac \mid Aad \mid bd \mid c$

$A \rightarrow bdA' \mid cA'$

$A' \rightarrow cA' \mid adA' \mid \epsilon$

LEFT FACTORING

→ **Left Factored Grammar.**

Grammar in the following form will be considered to be having left factored

$$S \longrightarrow aX \mid aY \mid aZ$$

Left-factored Grammar is having multiple productions with same starting symbol on the above eg. its 'a'

→ Problems with Left Factored Grammar

1) Creates an ambiguous situation for Top-Down Parsers

2) Top-down Parsers cannot choose the production to derive the given string since multiple productions will have a common prefix. This creates an ambiguous situation for the top down parser

3) Thus, we need to convert the left factored grammar into an equivalent grammar w/o any productions having common prefix.

4) LEFT FACTORING is the solution for this

* Left Factoring is a grammar transformation that is useful for producing a grammar suitable for PREDICTIVE or TOP-DOWN PARSING.

Rule for left factoring.

If $A \to \alpha\beta_1 \mid \alpha\beta_2 \mid \alpha\beta_3 \mid .. \mid \gamma_1 \mid \gamma_2$

then we replace wing foll. rules

$A \to \alpha A' \mid \gamma_1 \mid \gamma_2$

$A' \to \beta_1 \mid \beta_2 \mid \beta_3$.

Example

1) $S \to C+E \mid C*E \mid C/E$

Soln

$S \to CS'$

$S' \to +E \mid *E \mid /E$

2) $S \to \underset{\alpha}{iEtS} \mid \underset{\alpha}{iEtSeS} \mid \underset{\gamma}{a}$ (α: iEtS, α: iEtSeS, β: eS, γ: a)

$S \to iEtSS' \mid a$

$S' \to \epsilon \mid eS$

3) $S \to aAd^{\beta} \mid aB^{\beta}$

$A \to a \mid ab^{\beta}$

$B \to ccd \mid ddc$

$S \to aS'$

$S' \to Ad \mid B$.

$A \to aA'$

$A' \to \epsilon \mid b$

$B \to ccd \mid ddc$

* TERMINALS — Small letters.

Non-Terminals — Capital letters.

* To find FIRST & FOLLOW.

~~Rule~~

1) The construction of both Top-Down & Bottom-Up Parsers is aided by Two functions; FIRST and FOLLOW, associated with a grammar G.

2) During Top-Down Parsing, First & follow allow us to choose which production to apply based on the next input symbol.

RULES FOR FIRST

To compute FIRST of (X) for all grammar symbols, apply the following rules until no more terminals or ϵ can be added to any FIRST set

Small letter

1) If X is a terminal
then FIRST (X) = {X}

* for any small letter, its FIRST is that small letter itself *

2) If X → ϵ, then add ϵ to FIRST (X)

3) If X is Non-Terminal & X → $Y_1 Y_2 \ldots Y_k$ is a production then,
add FIRST (Y_1) to FIRST (X)
and if Y_1 → ϵ then
add FIRST (Y_2) to First(X)

Q1 Consider the following Non-left-Recursive grammar

$E \rightarrow TE'$
$E' \rightarrow +TE' \mid \epsilon$
$T \rightarrow FT'$
$T' \rightarrow *FT' \mid \epsilon$
$F \rightarrow (E) \mid id$

Soln

GRAMMAR	FIRST	FOLLOW
$E \rightarrow TE'$	{ (, id }	{ $,) }
$E' \rightarrow +TE' \mid \epsilon$	{ +, ε }	{ $,) }
$T \rightarrow FT'$	{ (, id }	{ +, $,) }
$T' \rightarrow *FT' \mid \epsilon$	{ *, ε }	{ +, $,) }
$F \rightarrow (E) \mid id$	{ (, id }	{ *, +, $,) }

starting point

Eg2 $S \rightarrow aBDh$
$B \rightarrow cC$
$C \rightarrow bc \mid \epsilon$
$D \rightarrow EF$
$E \rightarrow g \mid \epsilon$
$F \rightarrow f \mid \epsilon$

Soln:-

GRAMMAR	FIRST	FOLLOW
$S \rightarrow aBDh$	{a}	$ { $ }
$B \rightarrow cC$	{c}	{ g, f, h }
$C \rightarrow bc \mid \epsilon$	{ b, ε }	{ g, f, h }
$D \rightarrow EF$	{ g, f, ε }	{ h }
$E \rightarrow g \mid \epsilon$	{ g, ε }	{ f, h }
$F \rightarrow f \mid \epsilon$	{ f, ε }	{ h }

RULES FOR FOLLOW

To compute FOLLOW(A) for all non-terminals A apply the following rules until nothing can be added to any FOLLOW set

1) Place $ in FOLLOW(S) where S is the start symbol ($ is the input right end marker)

2) If there is production $A \to \alpha B \beta$ then everything in FIRST(β) is in FOLLOW(B)

* If a capital letter is followed by CAPITAL LETTER then write FIRST of that Capital letter.

3) If there is a production $A \to \alpha B$ or $A \to \alpha B \beta$ where FIRST(β) contains ϵ then everything in FOLLOW(A) is in FOLLOW(B).

* ϵ is never included in FOLLOW.

Eg Find Follow for Eg (1) & (2)

— Solved Behind

* If a capital letter appears at last symbol in a production Rule, then find FOLLOW of variable which is on the RHS of production

eg. $A \to aB$

↳ Follow(B) = Follow(A)

✱ LL(1) GRAMMAR

1) Predichive Pausers, i.e Recursive - Descent Pausers needing No Backtracking can be constructed for a class of grammars called LL(1).

LL(1)

- Stands for scanning the input from Left to Right
- for using one i/p symbol & lookahead at each step to make parsing action decision

Producing Leftmost Derivation

→ LL(1) Pauser or PREDICTIVE PARSER.

Steps to follow :-

1) Remove Left Recursion (if any)
2) Remove Left Factoring
3) Find FIRST() & FOLLOW()
4) Construct Pause Table
5) Stack Implementation
6) Pause Tree Generation

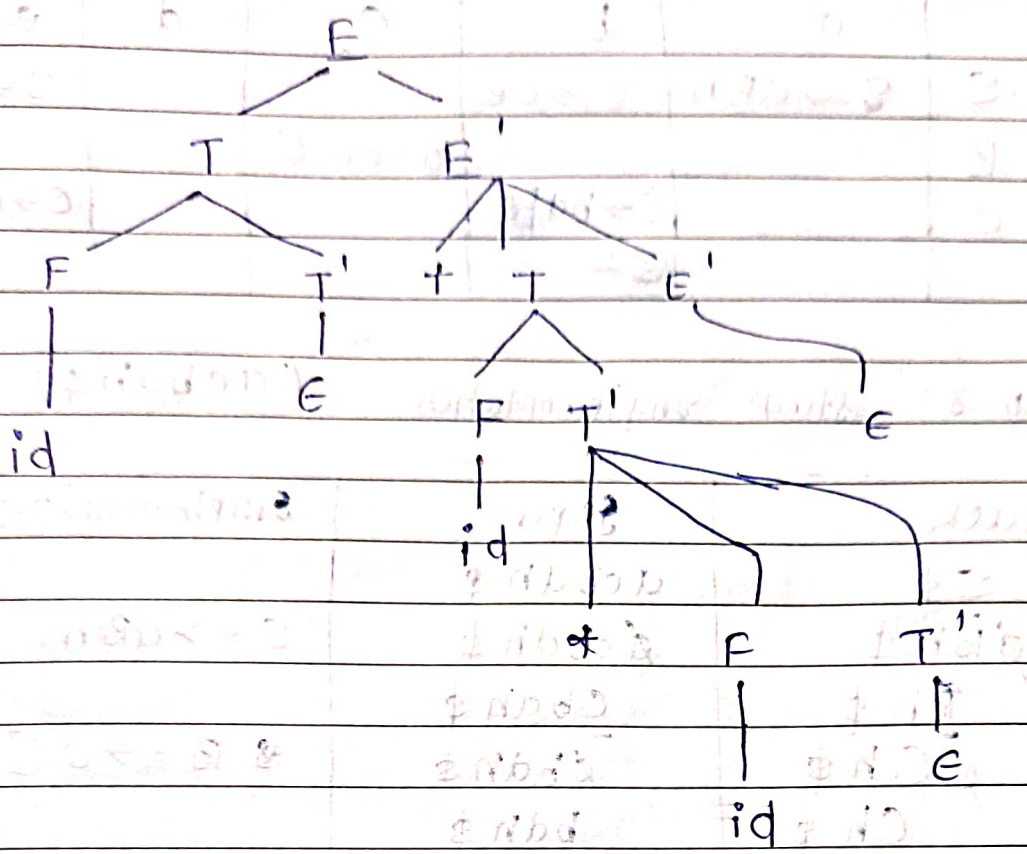
PARSING TABLE for Q1

@	+	*	id	(	)	$
E	~~E→TE'~~		E→TE'	E→TE'		
E'	E'→+TE'				E'→ϵ	E'→ϵ
T			T→FT'	T→FT'		
T'	T'→ϵ	T'→*FT'			T'→ϵ	T'→ϵ
F			F→id	F→(E)		

STACK IMPLEMENTATION (id + id * id $)

Stack	Input	Implementation
E $	id + id * id $	
TE' $	id + id * id $	E → TE'
FT'E'$	id + id * id $	T → FT'
~~id~~ T'E'$	~~id~~ + id * id $	F → id
T'E' $	+ id * id $	
E'$	+ id * id $	T' → ϵ
~~+~~TE' $	~~+~~ id * id $	E' → +TE'
TE'$	id * id $	
FT'E'$	id * id $	T → FT'
~~id~~ T'E'$	~~id~~ * id $	F → id
T'E' $	* id $	
~~*~~ FTE'$	~~*~~ id $	T' → *FT'
FT'E' ~~FT'$~~	id $	
~~id~~ T'E'$	~~id~~ $	F → id
T'E'$	$	
E'	$	T' → ϵ
$	$	E' → ϵ

PARSE TREE



Q2 $S \rightarrow aBh \mid Ce$ String :- acbdh$

$B \rightarrow cC$

$C \rightarrow bd \mid \epsilon$

Sol^n:- The given grammar does not require Left Recursion & Left Factoring

Step 1:-

Step 2

GRAMMAR	FIRST	FOLLOW
$S \rightarrow aBh \mid Ce$	{ a, b, e }	{ $ }
$B \rightarrow cC$	{ c }	{ h }
$C \rightarrow bd \mid \epsilon$	{ b, ϵ }	{ h, e }

Note :- If you encounter 'ϵ' in FIRST then select terminals from FOLLOW to fill table.

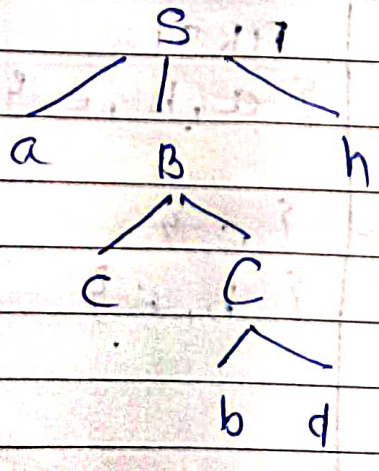
Step 3:- PARSING TABLE

	a	b	c	d	e	h	$
S	S→aBh	S→Ce			S→Ce		
B			B→cC				
C		C→bd			C→ϵ	C→ϵ	

Step 3: Stack Implementation (acbdh$)

Stack	Input	Implementation
S$	acbdh$	
aBh$	acbdh$	S→aBh.
Bh$	cbdh$	
Ch$	bdh$	B→cC
bdh$	bdh$	C→bd
dh$	dh$	
h$	h$	
$	$	

Step 4:- Parse Tree



Q3

$S \rightarrow A$

$A \rightarrow Bb \mid Cd$

$B \rightarrow aB \mid \epsilon$

$C \rightarrow cC \mid \epsilon$

Q2/4

The given grammar does not require.
Left Recursion &
Left R.Factoring

Step 2:- FIRST() & FOLLOW()

Grammar	FIRST	FOLLOW
~~cccfe~~		
$S \rightarrow A$	{a, b, c, d}	{$}
$A \rightarrow Bb \mid Cd$	{a, b, c, d}	{$}
$B \rightarrow aB \mid \epsilon$	{a, ∈}	{b}
$C \rightarrow cC \mid \epsilon$	{c, ∈}	{d}

Step 3:- PARSING TABLE

	a	b	c	d	$
S	$S \rightarrow A$	$S \rightarrow A$	$S \rightarrow A$	$S \rightarrow A$	
A	$A \rightarrow Bb$	$A \rightarrow Bb$	$A \rightarrow Cd$	$A \rightarrow Cd$	
B	$B \rightarrow aB$	$B \rightarrow \epsilon$			
C			$C \rightarrow cC$	$C \rightarrow \epsilon$	

Q Check if the following grammar is LL(1) or not

$S \rightarrow iCtSS_1 \mid a$
$S_1 \rightarrow eS \mid e$
$C \rightarrow b$

Step 1:- The given grammar does not require Left Recursion & Left factoring

Step 2 :- FIRST() & FOLLOW()

Grammar	FIRST	FOLLOW
$S \rightarrow iCtSS_1 \mid a$	{i, a}	{e, $}
$S_1 \rightarrow eS \mid \epsilon$	{e, ∈}	{e, $}
$C \rightarrow b$	{b}	{t}

~~Step 3:- PARSING~~

Step 3 :- PARSING TABLE

	a	i	e	b	t	$
S	$S \rightarrow a$	$S \rightarrow iCtSS_1$				
S_1			$S_1 \rightarrow \epsilon$ $S_1 \rightarrow eS$			
C						

Grammar is not in LL(1)