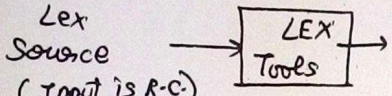
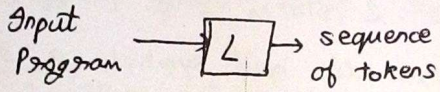
	a	b
A	B	A
B	B	D
D	B	E
E	B	A

LEX TOOLS

-Software to generate lexical analyser.

It is a tool for automatically generating the first phase of compiler i.e. lexical analysis.





The LEX source program consists of a set of R.E. together with an action for each R.E.

Action is a piece of code which is to be executed whenever the token specified by the corresponding R.E. is recognized in input program.

The LEX source program consists of 2 parts:-

1) Auxillary defination → R.E.
2) Translation Rules → Action

AUXILLARY DEFINITIONS :-

These are statements of the form :-

$D_1 = R_1$
$D_2 = R_2$
:
$D_n = R_n$

where each D_i is a distinct name & each R_i is R.E.

Ex :- letter = A|B|C| -- |Z
digit = 0|1| - - - |9
identifier = letter (letter | digit)*

TRANSITION RULES

These are statements of the form :-

P_1 $\{A_1\}$

P_2 $\{A_2\}$

⋮

P_n $\{A_n\}$

where each P_i is a R.E. called a pattern.
each A_i is a program fragment describing what action the lexical analysis LA should take when token P_i is fixed.

The working of lexical analysis L can be explained as follows.

L reads the input 1 character at a time until it has found the longest prefix of input that matches one of the regular expression, P_i then it executes the corresponding action A_i & returns the control to Parser.

Ex :- Auxillary Definations :-

letter = A | B | - - | z

digits = 0 | 1 | - - | 9

Translation Rules :-

BEGIN {return 1}

END {return 2}

IF {return 3}

THEN {return 4}

ELSE {return 5}

Letter (Letter|digit)* { LEXICAL = INSTALL();
return 6 }

digit+ { LEXICAL = INSTALL();
return 7 }

< { LEXICAL = 1;
return 8;
}

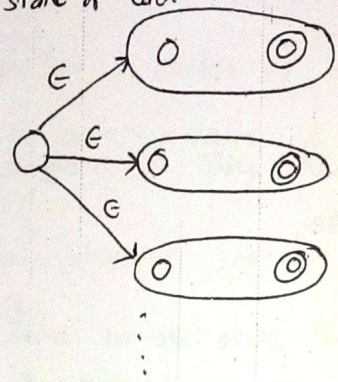
Suppose ZA is given the input BEGIN, both first & sixth pattern matches so it's a conflict. But since the pattern for keyword BEGIN comes before the pattern identifies. So the conflict is resolved in favours of keywords. So keywords are defined first & then the identifies.

P.T.O.

LEX TOOL :-

IMPLEMENTATION OF L.A. :-

The LEX tool helps to produce L-A. that behaves as finite automaton. We construct a NFA N_i for each token pattern P_i & then link all these NFA's together with a new start state. Then we convert this NFA to DFA. In the combined NFA several different accepting states are there. Accepting state of each NFA indicates that its token P_i has been found.



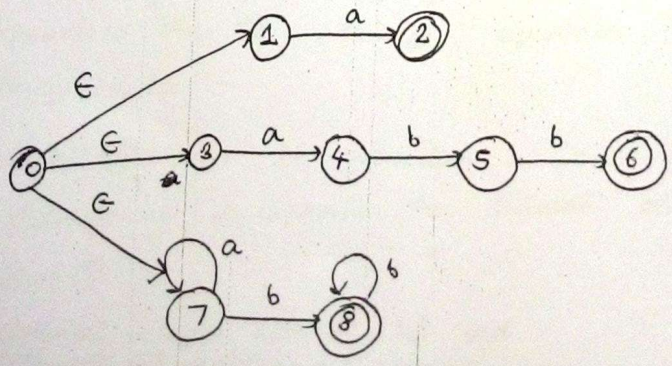
Ex :- Following is a LEX program :-

AD

T.R

a { }
abb { }
a*b+ { }

NFA



DFA

States	a	b	Token Found
0137	247	8	none
247	7	58	a
8	–	8	a^*b^+
7	7	8	none
58	–	68	a^*b^+
68	–	8	abb

The last column of DFA shows token that will be recognized if that state is the accepting state of that token.

In the first stage 0137 there is no final state so no token is found. In the second stage 247 state 2 is the final state of token 'a'. So 'a' is found & so on.

In the last stage 68 both 6 & 8 are final states but since the translation rules of token abb are mentioned before the token a^*b^+. So the token abb has priority over a^*b^+

GRAMMER

Q.) Construct a minimal state DFA for the following regular expression :-

$(a|b)^* / (ab)^*b | a^*(bb)^*$.

GRAMMER

CHAPTER-4

"SYNTACTIC SPECIFICATION OF PROGG. LANG"

→ Parser is build around a certain grammer.

We use a notation called as CFG or grammar or BNF (Backus Naur form) - for the syntatic specification of programming language.

- It is TYPE 2 in Chomsky classification.

This notation has a number of advantages :-

1) A grammar gives precise & easy to understand syntactic specifications of programs.

2) For properly designed grammar we can generate an automatic parser automatically.

CONTEXT FREE GRAMMAR :-

Before definition of CFG, let us look at full- examples :-

1) If S_1 & S_2 are statements & E is an expression.
then if E then S_1 else S_2 is a statement ... (1).

2) If $S_1, S_2, \ldots S_n$ are statements then

begin S_1 ; S_2 ; S_3 , ... S_n end is a statement ... (2)

3) If E_1 & E_2 is an expression then $E_1 + E_2$ is an expression .. (3)

4) If we use a syntactic category "statement" to represent all the statements & another category "expressions" to denote all the expressions then the following equations can be rewritten by following rules :-

→ Equation (1) can be rewritten as :-

statement → if expression then statement else statement .

Equation (2)

statement → begin statement_list end
statement_list → statement | statement ; statement_list

Equation (3)

Expression → expression + expression.

For equation 2. we define a new syntactic category i.e. "statement_list".

A CFG is defined as :-

1) Non-Terminals
2) Terminals
3) Start-symbol
4) Production rules

TERMINALS :- These are the tokens. These are the basic symbols of which strings in a language are composed of.
Ex :- Begin, end, for, +, etc.
(TOKENS).

NON-TERMINALS :- These are the special symbols that denotes sets of streams. These are syntactic categories or syntactic variables.
Ex :- statements, expressions etc.

→ Start symbol is a non-terminal that denotes the complete language. It is usually in first non-terminal used in first non-production rules.

PRODUCTION-RULES :- These rules defines the ways in which syntactic categories may be build up from 1 another & from T & NT. Each production consists of a NT followed by an arrow that is followed by string of terminals & non-terminals.

Ex :- Consider a grammar for simple arithmetic expressions :-

NT → expression, operator

S.S. → expression

T → id, +, −, *, /, ↑, (,)

Productions :-

expression → expression operator expression.
expression → (expression)
expression → − expression
expression → id
operator → + | − | * | / | ↑

There are some notational conventions that we follow while designing grammar.

1) NT ← Syntactic categories like expressions, statements etc.
Capital letters like A, B, C
Letter S is used to denote start symbols.

2) Terminals → a, b, c, ... near the alphabets

- Tokens
- operators like +, −, ...
- digits like 0, 1, ... 9
- punctuation symbols like ';', ',' etc.
- Strings like identifiers ('id'), begin, end etc.

3) Capital letters at the end of alphabets. like X, Y, Z are used to denote Grammer symbols i.e. either Terminals or non-terminals.

4) Small letter at the end of alphabet like u, v, x, y, z are used to denote strings of terminals.

5) Symbols like α, β, γ are used to denote strings of terminals & non-terminals.

6) If we have productions

$A \to \alpha_1$
$A \to \alpha_2$
⋮

These are called as A-productions.

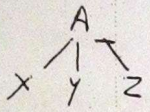
$A \to \alpha_1 | \alpha_2 | \alpha_3 | \ldots$

PARSE TREE

A parse tree is a graphical representation for the derivations. It is a hierarchical syntax structure.

Ex:- $A \rightarrow XYZ$

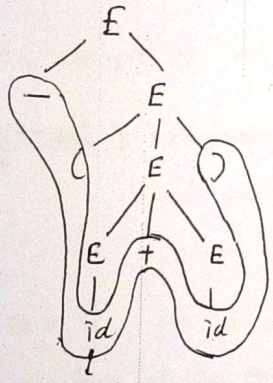
This production is used at some point in the derivation. Then its parse tree is shown as :-

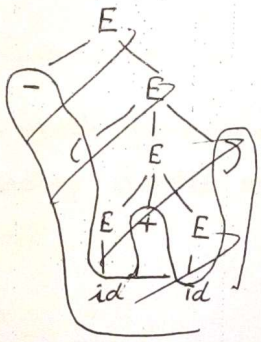


- In parse tree each interior node is labelled by some non-terminal. Childres are labelled from left to right by the symbols of right side of the derivations.
- The leaves of parse tree are labelled by terminals & read from left to right are called YIELD OR FRONTIER of the tree.

Ex:- Parse tree of the string $-(id+id)$ & the same of grammar :-

$E \rightarrow E+E \mid (E) \mid -E \mid id \mid E*E$





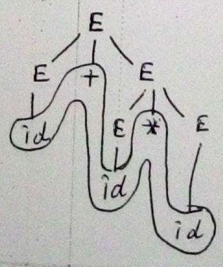
yield or frontier.

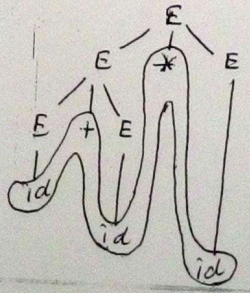
→ For the same grammar the following string has 2 leftmost derivations :-

$id+id*id$

$E \Rightarrow E+E$	$E \Rightarrow E*E$
$\Rightarrow id+E$	$\Rightarrow E+E*E$
$\Rightarrow id+E*E$	$\Rightarrow id+E*E$
$\Rightarrow id+id*E$	$\Rightarrow id+id*E$
$\Rightarrow id+id*id$	$\Rightarrow id+id*id$.

So now these are 2 parse trees.





Hence grammar is ambiguous.

The selection of the correct parse tree depends upon the precedence of operators.

AMBIGUITY

A grammar is said to be ambiguous if it produces more than 1 parse tree for the same string

Or

A grammar which produces more than 1 leftmost or rightmost derivation for the same string or sentence.

For an efficient parser the grammar should be unambiguous. To remove ambiguity from a grammar we can use DISAMBIGUITING RULES.

Ex:- The grammar

$$E \rightarrow E+E \mid (E) \mid E*E \mid E-E \mid E/E \mid E \uparrow E \mid -E \mid id$$

is an ambiguous grammar.

We can disambiguate this grammar by defining the precedence & associativity of the operators. Suppose we assume following precedence :-

1.) $-$ (Unary $-$)
2.) $\uparrow$ (right associative)
3.) $*$, $/$ (left associative)
4.) $+$, $-$ (left associative)

We can modify our grammar to include rules into it. For this we introduce 1 non-terminal for each precedence level. Following is the list of non-terminals introduced.

- Element
- Primary
- Factor
- Term
- Expressions

1.) Element → (expression) | id

An element is a sub-expression that is indivisible if it is either a single identifier or a parenthesized expression.

2.) primary → - primary | element

A primary is an element with 0 or more operators of highest precedence. (unary minus).

3.) factor → primary ↑ factor | primary

~~4.) term → term * term | +~~

4.) term → term * factor | term / factor | factor.

terms are sequences of 1 or more factors connected by * & /.

5.) expressions → expressions + term | expression - term | term.

Another example of an ambigious grammar is :-

statement → if condition then statement |
if condition then statement else statement |
other-statement.

For example for a string,

if c_1 then if c_2 then s_1 else s_2

it can have 2 parse trees.

→ The ~~this ambigui~~ disambiguity rule used in this gramar that each ~~then~~ ~~is~~ else is to be matched with closest previous unmatched ~~else~~ then then it will be an unambigious.

S-R PARSING

Shift-reduce parsing

Handle & Handle ~~Parsing~~ Pruning

Handle is a substring i.e. the right side of the production such that the replacement of that substring by the production on that side leads finally to a reduction to ~~the~~ start symbol.

The process of bottom-up parsing can be viewed as that of finding and reducing handles.

~~Handle Pruning :-~~

Handle Pruning :- It gives rightmost derivation in reverse i.e. we start with a string of terminals i.e. w which we wish to parse. If w belongs to the grammar then w can be written as $w = \gamma_n$ where γ_n is n^{th} right sentential form. So, we can write the following equation.

$$S = \gamma_0 \underset{\pi_1}{\Rightarrow} \gamma_1 \cdots \gamma_{n-2} \underset{\pi_n}{\Rightarrow} \gamma_{n-1} \underset{\pi_n}{\Rightarrow} \gamma_n = w$$

We locate the handle ~~β_n~~ β_n in γ_n & replace it by the left side of the production $A_n \rightarrow \beta_n$ to obtain $(n-1)^{th}$ right sentential form. This process is repeated till we reach S.

→ So, S-R parsing is right most derivation in reverse order.

STACK IMPLEMENTATION :-

We ~~use~~ use a stack & input buffer to implement S-R parsing. We use $ to mark the bottom of the stack & right end of input string. The parser operates by shifting the input symbols on to a stack until a handle β appears on the top of the stack. It is then reduced to A. This process is repeated till input string is empty or error.

Ex :- I/p string : $id_1 + id_2 * id_3$

Grammar :- $E \rightarrow E+E \mid E-E \mid E*E \mid id$

Stack	I/P	Action
$	$id_1 + id_2 * id_3$ $	Shift
id_1	$+ id_2 * id_3$ $	reduce by $E \to id$
$E	$+ id_2 * id_3$ $	~~reduce~~ shift
$E +	$id_2 * id_3$ $	shift
$E + id_2$	$* id_3$ $	reduce by $E \to id$
$E + E	$* id_3$ $	~~shift the~~ shift
$		~~At this po~~

At this point the parser gets the conflict, it can either shift the next symbol on the stack or it can ever reduce the handle. This has occured because grammar has ambiguous.
[SHIFT - REDUCE CONFLICT].

Suppose the parser shifts the next input symbol here.

$E + E *	id_3 $	shift
$E + E * id_3$	$	reduce by $E \to id$
$E + E * E$	$	reduce by $E \to E * E$
$E + E	$	reduce by $E \to E + E$
$E	$	accept

Suppose the parser reduce by $E \to E + E$.

$E + E	$* id_3$ $	reduce by $E \to E + E$
$E	$* id_3$ $	shift
$E *	id_3 $	shift
$E * id_3$	$	reduce by $E \to id$
$E * E	$	reduce by $E \to E * E$
$E	$	accept.

So, both the methods are correct to accept string.

ACTION/OPERATION OF PARSER :-

1.) **SHIFT** :- In this action the next input symbol is shifted to the top of the stack.

2.) **reduce** :- In this action the parser replaces the handle on the top of the stack with the left side of the production.

3.) **accept** :- In this operation parser announces successful completion of parsing.

4.) **error** :- In this action parser discovers that a syntax error has occured & calls the error recovery routine.

TOP-DOWN PARSING

In top-down parsing we construct a parse tree starting from the top & moving down to the leaves.

Ex:- Consider a grammar,

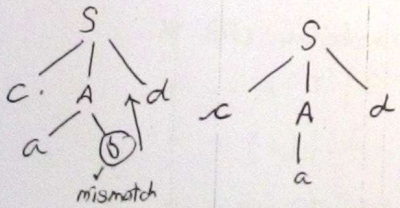
$S \rightarrow cAd$

$A \rightarrow ab|a$

&

the given input string, $w = cad$

We draw a parse tree. In the begining, it consists of a single node i.e. S. We keep an input pointer pointing to the first symbol w i.e. c. We expand S by its production.



The leftmost leaf i.e. c matches it with the first symbol of w. So we move the input pointer to the next symbol i.e. a & take the next leaf. ~~By~~ We expand A by using its first production.

The second input symbol matches with a. Now the next input symbol is d & the next leaf is b. There is a mismatch. In this case we go back to A to look for its another ~~pro~~ production. We move the input pointer back to position 2.

the next input symbol matches with the third leaf - So we have produced parse tree. To implement top-down parser we write procedures for each non-terminals.

→ PROCEDURES from book

PROBLEMS / DIFFICULTIES :-

1.) BACKTRACKING

2.) LEFT RECURSION

Ans:

→ BACKTRACKING :- When we make expansions of non-terminals & find a mismatch. This problem is called as Backtracking. In such a problem entries in the single problem needs to be removed. So it causes overhead.

→ LEFT RECURSION :- A grammer is left recursive if there is a production $A \rightarrow A\alpha$. In this case the top-down parser goes into infinite loop.

ELIMINATION OF LEFT RECURSION :-

If we have a production $A \rightarrow A\alpha \mid \beta$ where β does not begin with A. Then we can replace these productions with the following :-

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

Ex :- We have a grammar,

$$E \rightarrow E+T \mid T$$
$$T \rightarrow T*F \mid F$$
$$F \rightarrow (E) \mid id$$

$E \rightarrow TE'$

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

~~F →~~

$T \rightarrow FT'$

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

$F \rightarrow (E) | id$

→ Here this grammar is unambigious grammar where E means expression & T means term.

→ Now we have removed left recursive problem from it.

→ QUICK & DIRTY COMPILER → study.

RECURSIVE DESCENT PARSING :-

~~Top, D~~

- Free from backtracking

It is a top-down parser having no backtracking. This parser uses some recursive procedure in the implementation. Ex:-

Grammer is :-

$E \rightarrow TE'$

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

$T \rightarrow FT'$

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

$F \rightarrow (E) | id$

- In the implementation we write procedure for all non-terminals i.e. E, E', T, T', F.

- PROCEDURES → from book.

SNP :-

LEFT - FACTORING

Sometimes the grammar is not suitable for ~~these~~ recursive descent parsing even if there is no left recursion in it. There is a concept of LEFT FACTORING used.

LEFT - FACTORING :-

It is the process of factoring out of common prefix for right hand side of the production -

Ex:- If there is a grammar :-

$S \rightarrow iCtS \mid iCtSeS$

These productions can be replaced with the following :-

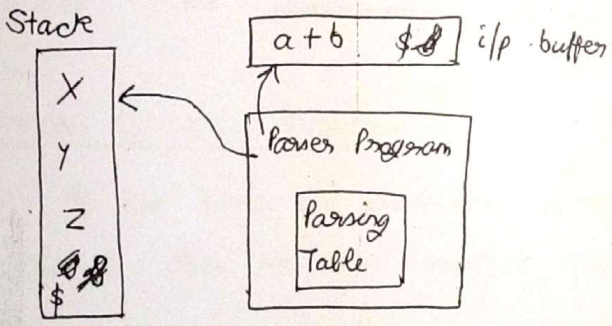
$S \rightarrow iCtSS'$

$S' \rightarrow eS \mid \epsilon$

PREDICTIVE PARSING — Parsing table .

It is an efficient top-down parser that makes use of a parsing table.

STRUCTURE OF PREDICTIVE PARSER :-



Initially the stack contains the start symbol of grammer preceded by $. The parsing table is 2D array M[A, a].

Rows → non-terminals

Columns → terminals

Entries → Either production or blank.

The parser works as follows :-

1.) It determine the 'X' i.e. symbol on the top of stack & the current i/p symbol 'a'. Then there are 3 possibilities :-

i.) If $X = a = \$$, then the parser stops & announces succesful completion of parsing.

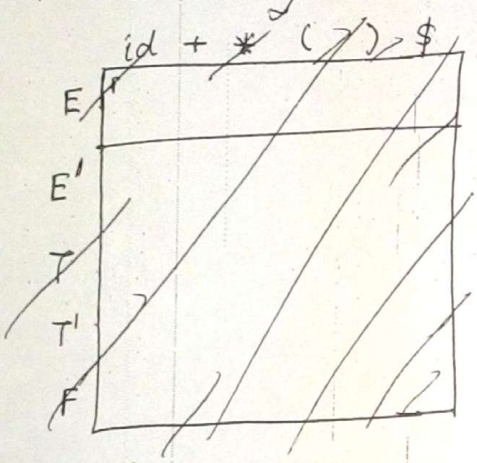
ii.) If $X = a \neq \$$, then parser pops X from the stack & advances the i/p pointer to the next symbol.

iii.) If X is a non-terminal then parser consults the entry $M[X, a]$ of the parsing table. This entry will be either an X production of a grammar or error. If parser is blank or error the parser calls error recovery routine.

- if $M[X, a] = \{X \rightarrow UVW\}$

the parser replaces X on the top of the stack by ~~WU~~ WVU.

Ex:- For same grammar



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