UNIT 5: SCANNING AND PARSING

Scanning - A Scanner simply turns an input string (say a file) into a list of tokens. These tokens represents thing like identifiers, parentheses, operators etc.

Parsing - A parser converts this list of tokens into a Tree-like object to represent how the tokens fit together to form a cohesive whole (sometimes referred to as a sentence)

Two types of Parsing Method

1. Bottom-up Parsing Technique

- A parser can start with the input & attempt to rewrite it to the start symbol. Intvitively the parser attempts to locate the most basic elements, then the elements containing these & so on. Example : LR parsers.

Process:

- Constructs a parse tree for an input string beginning at the leaves & working upwards towards the root.
- To do so, bottom-up parsing tries to find a rightmost derivation of a given string backwards.

→ Bottom-up parsing is also called shift & reduce parsing, where shift means read the next token, and reduce means that a substring B matching the right side of a production A → B is replaced by A

→ Bottom-UP parsing reduces a string to the start symbol by inserting productions:

int * int * int	T → int
int * T + int	T → int * T
T + int	T → int
T + T	E → T
T + E	E → T + E
E	

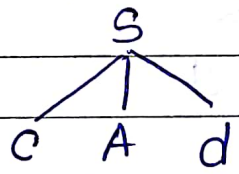
→ Bottom-Up Parsing recognizes 'the text's' lowest level small details first, before its mid-level structures, leaving the highest-level overall structure to last.

Top-Down Parsing

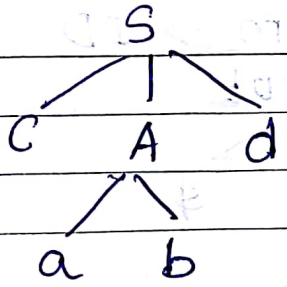
→ It is a parsing technique where the parser first looks at the highest level of the parse tree & works down the parse tree by using the rules of a formal grammar.

Process

Step 1: From Start Symbol

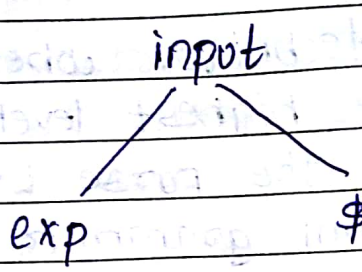


Step 2: Expand A using first alternate A → ab to obtain the foll tree

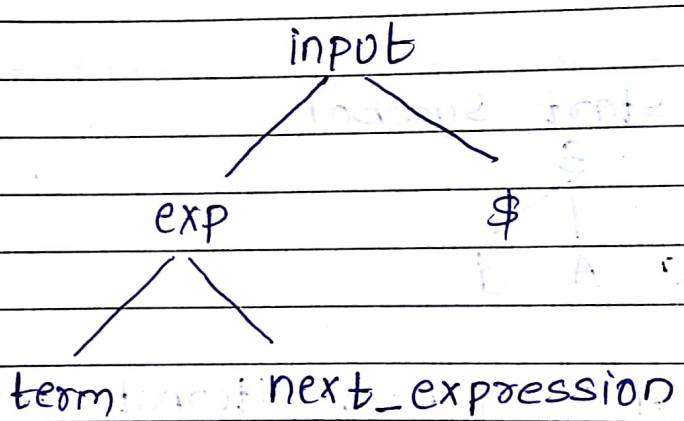


Build Parse Tree: Start from start symbol to invoke:

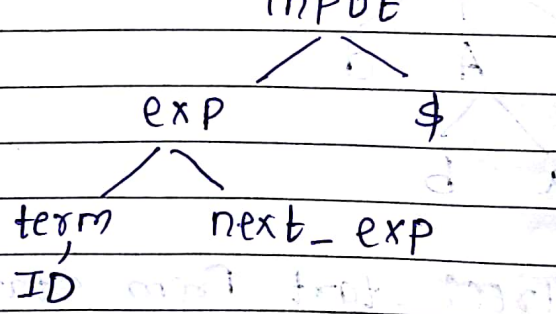
int input(void)



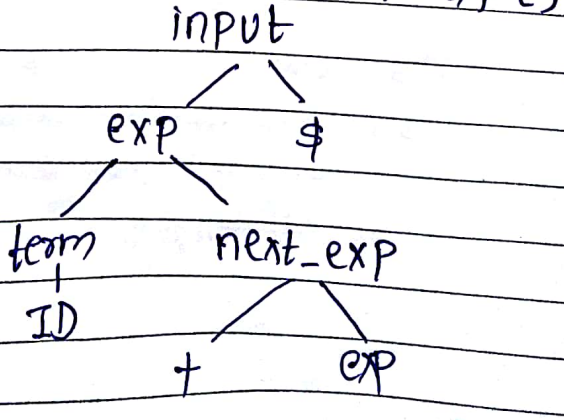
next, invoke expression



select term → ID



invoke next_exp()



Recursive-Descent Parsing

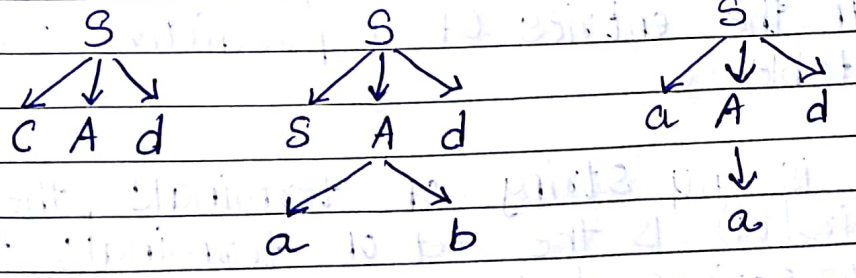
A recursive-Descent Parsing Program consits of a set of procedures, one for each nonterminal.
Execution begins with the procedure for the Start symbol, which halts and announces success if its procedure body scans the entire input string.

General recursive-descent may require backtracking; that is, it may require repeated scans over input.

Consider the input grammar with input string cAd

$S \rightarrow cAd$

$A \rightarrow ab \mid a$



Predictive Parsing

A parsing technique that used a lookhead symbol to determine if the current i/p arguments matches the lookhead symbol

1.] LL Grammar

The first L in LL means stands for scanning the i/p from left to right, the second L is for producing a leftmost derivation.

2.] First and Follow

First and Follow aids the construction of a predictive parser. They allow us to fill in the entries of a predictive parsing table.

a is any string of terminals, then First(a) is the set of terminals that begin the strings derived from a. If a is an empty string (ε) then ε is allow in First(a)

Follow(A), for a nonterminal A, to be the set of terminals a that can appear immediately to the right of A in a sentential form.

Rules for computing FIRST(X):

1.) If X is a terminal, then FIRST(x) = X.
2.) If X is ϵ, then FIRST(X) = ϵ
3.) If X is a non-terminal and Y and Z are non-terminals, with a production of
X → Y
Y → Z a
Z → b ; then FIRST(X) = b ;
where FIRST(nonterminal1) → FIRST(nonterminal2) or until you reach the first terminal of the production
4.) If X is a non-terminal and contains two productions. Ex X → a|b ; then FIRST(X) = {a, b}

Rules for computing FOLLOW(X)

1.) If X is a part of a production and is succeeded by a terminal, A → Xa ; then FOllow (X) = {a}
2.] If X is a start symbol for a grammar, for ex:
X → AB
A → a
B → b ; then add $ to FOLLOW(x) ;
FOLLOW (X) = {$}
3.] If X is a part of a production and followed by another non terminal, get the FIRST of that succeeding nonterminal.
A → XD D → aB ; then FOLLOW (X) = FOLLOW (D) = {a} ;

4] If X is the last symbol of a production, ex: $S \rightarrow abX$, then FOLLOW(X) = FOLLOW(S)

Examples:

1]

	First	Follow
$S \rightarrow aABb$	{a}	{$}
$A \rightarrow c/\epsilon$	{c, ε}	{d, b}
$B \rightarrow d/\epsilon$	{d, ε}	{b}

2]

	First	Follow
$S \rightarrow Bb/Cd$	{a, b, c, d}	{$}
$B \rightarrow aB/\epsilon$	{a, ε}	{b}
$C \rightarrow cC/\epsilon$	{c, ε}	{d}

3]

	First	Follow
$S \rightarrow ABCDE$	{a, b, c}	{$}
$A \rightarrow a/\epsilon$	{a, ε}	{b, c}
$B \rightarrow b/\epsilon$	{b, ε}	{c}
$C \rightarrow c$	{c}	{d, e, $}
$D \rightarrow d/\epsilon$	{d, ε}	{e, $}
$E \rightarrow e/\epsilon$	{e, ε}	{$}

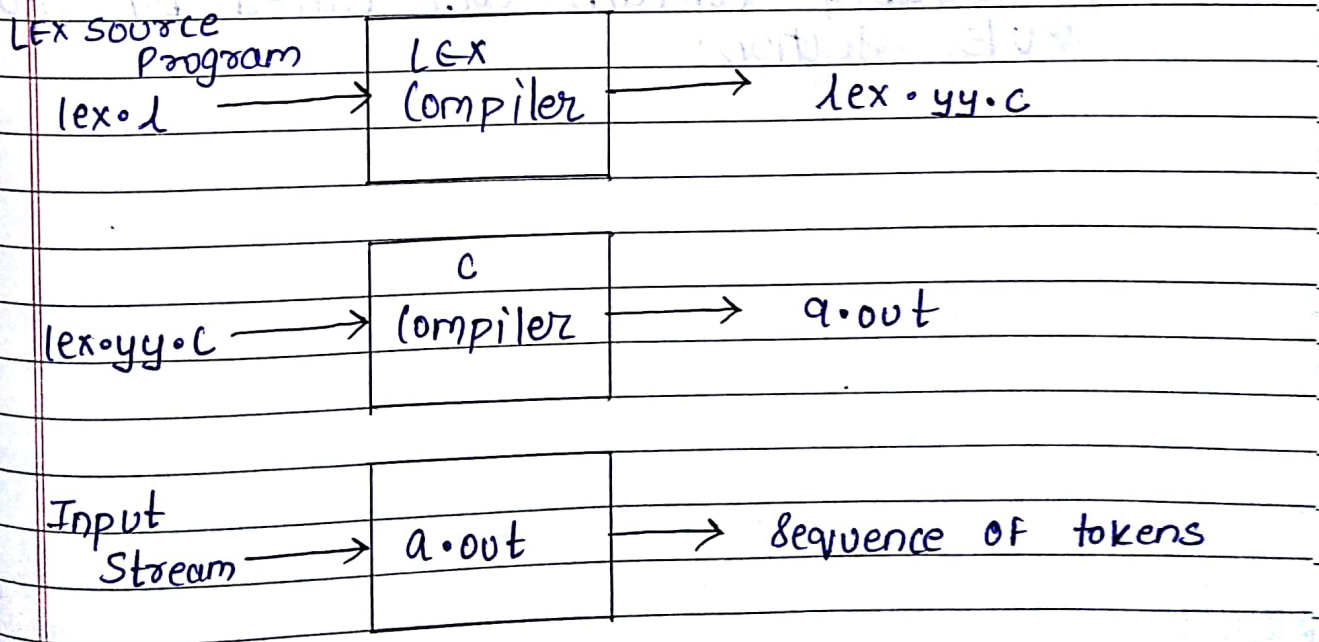
LEX - A Lexical Analyzer Generator

a. LEX helps write programs whose control flow is directed by instances of regular expressions in the i/p stream.

b. It is well suited for editor-script type transformations & for segmenting i/p in preparation for a parsing routine.

c. It is a table of regular expressions & corresponding program fragments.

d. The table is translated to a program which reads an i/p stream, copying it to an o/p stream & partitioning the i/p into strings which match given expressions.



Structure of LEX file:

1] Definition: The definition section defines macro & imports header file written in C. It is also possible to write any C code here, which will be copied over time into generated source file

2] Rules: The Rules Section associates regular expression patterns with C statements. When the lexer uses texts in the input matching a given pattern it will execute associated C code.

3] C code : It contains C statements & functions that are copied verbatim to the generated source file. These statements presumably contain code called by rules in rules section.

YACC - Yet Another Compiler Compiler

a. It is a computer program for UNIX operating system.

b. It is a look ahead left-to-Right (LALR) parser, the part of a compiler that tries to makes ~~semanti~~ ~~semtactic~~ Syntatic sense of the source code

c. YACC produces only a parser; for full syntactic analysis this requires an external lexical analyzer.

d. It provides a general tool for imposing structure in the input to a computer system

e. YACC user prepares a specification of the i/p process; this includes rules describing the i/p structure, code to be invoked when these rules are recognized, & a low-level routine (lex analyzer) to pick up the basic items. (tokens) from the i/p stream

f. These tokens are organized according to the i/p structure rules & called grammar rules, when one of these rules has been recognized, the user code applied for this rule, an action, is invoked.