Compiler design (4)

Syntax Analysis :- In our compiler model, the parse obtains a string of tokens from the lexical analyz as shown in figure (1), and verifies that the string can be generated by the grammar for the source language. We expect the parser to report any synt errors in an intelligible fashion. It should also recover from commonly occurring errors so that it can continue processing the remainder of its input.

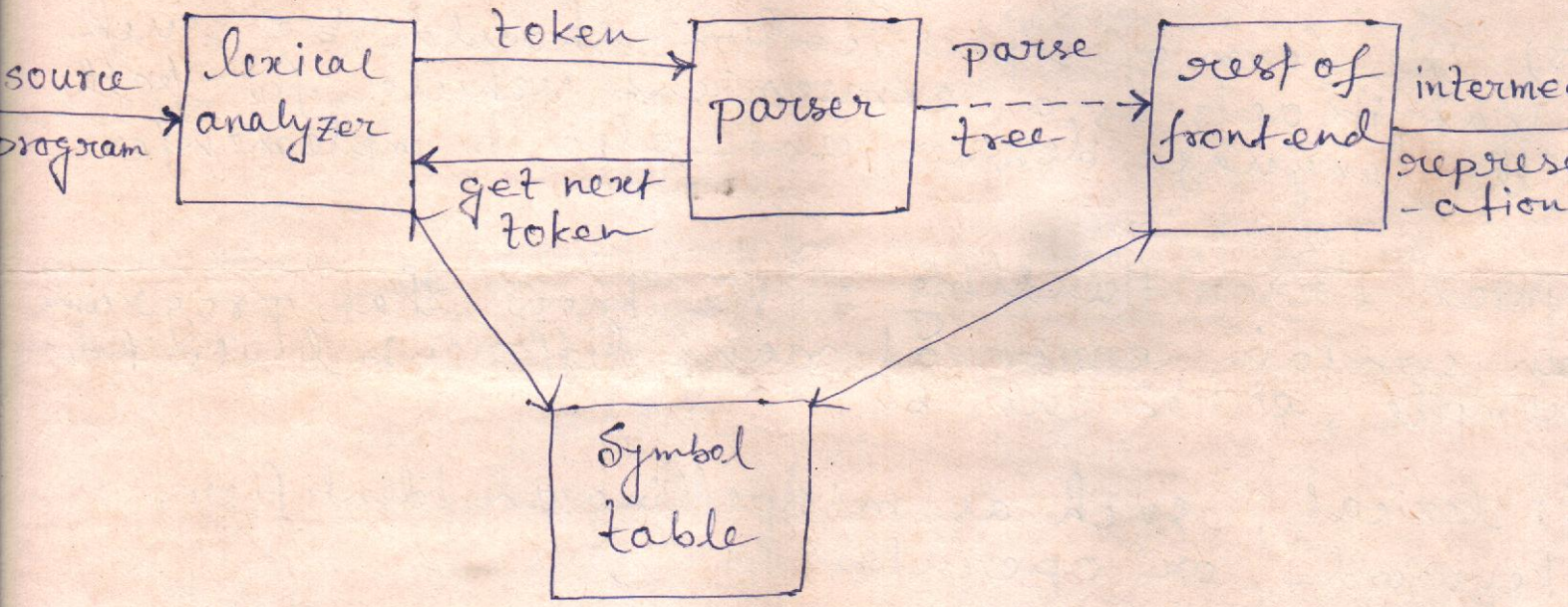


Fig 1 :- Position of parser in compiler model.

Basic Issues In Parsing :- A very basic issue in parsing is related to the specification of syntax of a programming language. It is obvious that a

parser must be provided with a description of language syntax. How to describe the same is an important question and the related issues are given in the following.

(1) The specification should be precise and unambiguous.

(2) The specification should be ~~such that~~ complete, that is, it must cover all the syntax details of the language.

(3) The form of specification should be such that it acts as a convenient vehicle for both the language designer and also its implementer.

Syntax Error Handling :- We know that programs can contain errors at many different levels. For example, error can be,

(i) lexical, such as misspelling an identifier, keyword, or operator.

(iii) ~~syntactic~~ semantic such as an operator applied to an incompatible operand.

(ii) syntactic, such as an arithmetic expression with unbalanced parentheses.

(iv) logical, such as an infinitely recursive call.

The error handler in a parser has simple-to-state goals:-

i) It should report the presence of errors clearly and accurately.

ii) It should recover from each error quickly enough to be able to detect subsequent errors.

iii) It should not significantly slow down the processing of correct programs.

Error-Recovery Strategies :- Here we introduce the following strategies :-

(i) panic mode
(ii) ~~pht~~ phrase level
(iii) error productions
(iv) global correction

Compiler Design (5)

Ambiguity :- A context free grammar G is said to be ambiguous if for some sentence $w \in L(G)$, there exist more than one parse tree whose frontiers are w. In otherwords, there will be at least two leftmost derivat or two rightmost derivations.

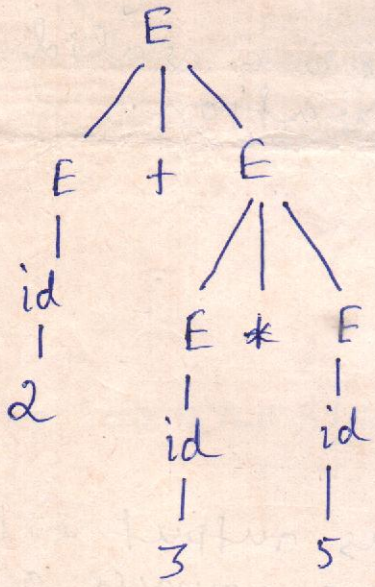
Ex :- $G = \langle V_N, \Sigma, P, S \rangle$

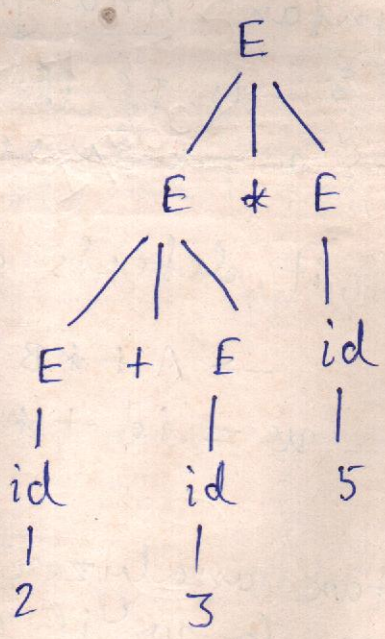
$V_N = \{ E \}$

$\Sigma = \{ +, *, (,), id \}$

$P : E \to E+E \mid E*E \mid (E) \mid id$

Consider the string $2+3*5$





$E \to \underline{E}+E \to id+E \to id+\underline{E}*E \to id+id*E$

$\to id+id*id$

$E \to E*E \to \underline{E}+E*E \to id+\underline{E}*E \to id+id*\underline{E}$
$\to id+id*id$

Similarly, there exist two rightmost derivations.

Disambiguation :-

G': $E \to E+T \mid T$
$T \to T*F \mid F$
$F \to (E) \mid id$

where T and F are new nonterminals. Verify the
$L(G) = L(G')$. Also derive $2+3*5$.

Parsing or Syntax Analyzer :- (To check for synt
specification) :- (i) If the given string is a valid on
then it discovers or produces the derivation.

(ii) Otherwise, it detects an error.

e.g., $\to A+*B$
$w = id+*id$.

(iii) A syntax analyzer produces as output either
a parse tree for w if $w \in L(G)$ or announces a
syntax error if $w \notin L(G)$.

Note :- A parse tree may not be explicitely generated equivalent sequence of production rules may be generated.

Top-down Approach :- Start from the start symbol S of G and check whether $S \overset{*}{\Rightarrow} w$. (S derives w).

Bottom-up Approach :- Start from the given string w and check whether w can be reduced to S.

Example :- $G = \langle \{S\}, \{a, b, c\}, P, S \rangle$

P : 1. $S \to aSb$

2. $S \to c$

Let $w = aaacbbb$

Then, $S \xrightarrow{(1)} aSb \xrightarrow{(1)} aaSbb \xrightarrow{(1)} aaaSbbb \xrightarrow{(2)} aaacb$

Top-down approach

$aacbbb \Rightarrow aaaSbbb \Rightarrow aaSbb \Rightarrow aSb \Rightarrow S$

Bottom-up approach

Eliminating Ambiguity :- Sometimes an ambiguous grammar can be rewritten to eliminate the ambiguity. As an example, we shall eliminate the ambiguity from the following "dangling-else" grammar :-

stmt → if expr then stmt
| if expr then stmt else stmt ..... (1)
| other

Here "other" stands for any other statement. According to this grammar, the compound conditional statement

if E_1 then S_1 else if E_2 then S_2 else S_3

has the parse tree shown in figure (2). Grammar (1) is ambiguous since the string

if E_1 then if E_2 then S_1 else S_2 ..... (2

has two parse trees shown in figure (3).

all programming languages with conditional statemen this form, the first parse tree is preferred. The eneral rule is, "Match each else with the closest previous unmatched then." This disambiguating rule can e incorporated directly into the grammar.

For example, we can rewrite grammar (1) as the following unambiguous

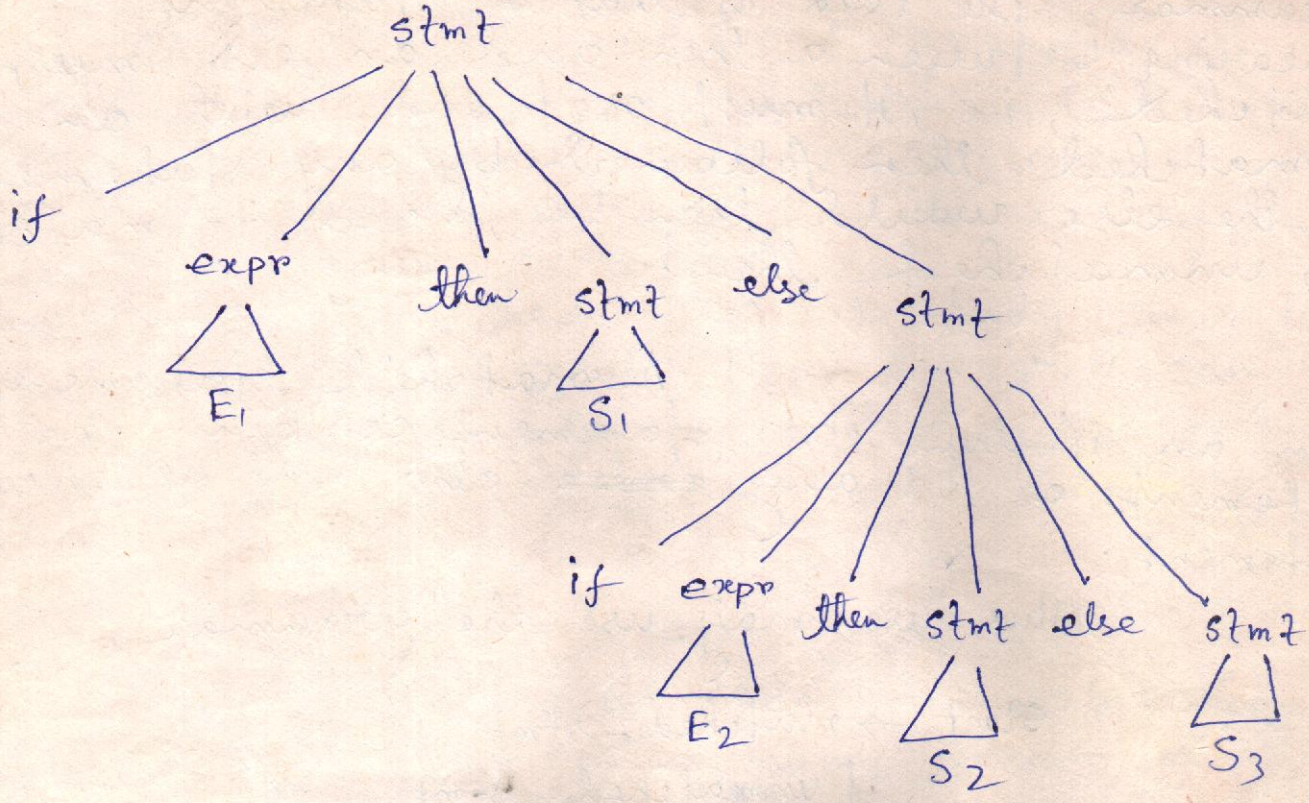
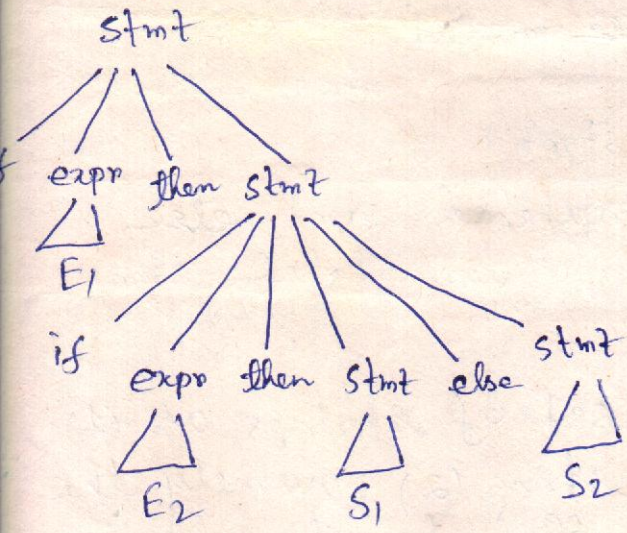
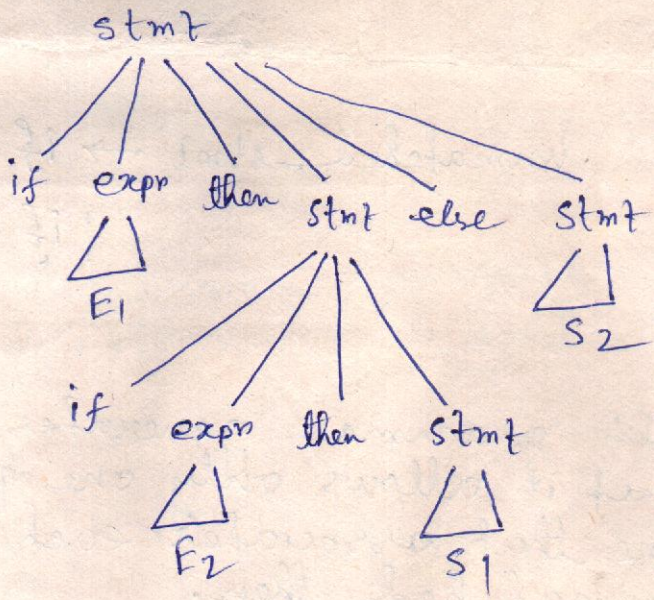


Fig 2 :- Parse tree for conditional statement





fig(3) :- Two parse trees for an ambiguous sentence

grammar. The idea is that a statement appearing between a then and an else must be "matched"; i.e., it must not end with an unmatched then followed by any statement, for the else would then be forced to match this unmatched then.

A matched statement is either an if-then-else statement containing no unmatched statements or it is any ~~order~~ other kind of unconditional statement.

Thus, we may use the grammar,

```
stmt → matched_stmt
     | unmatched_stmt
```

.....

```
matched_stmt → if expr then matched_stmt
                    else matched_stmt

unmatched_stmt → if expr then stmt
               | if expr then matched_stmt else
                    unmatched_stmt
```

This grammar generates the same set of strings as (1) but it allows only one parsing for string (2), namely the one that associates each else with the closest previous unmatched then.

Elimination of Left Recursion :- A grammar is left recursive if it has a nonterminal A such that there is a derivation $A \overset{+}{\Rightarrow} A\alpha$ for some string α. Top-down parsing methods can not handle left-recursive grammars, so a transformation that eliminates left-recursion is needed. Here we study how the left-recursive pair of productions $A \to A\alpha \mid \beta$ can be replaced by the non left-recursive productions

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

without changing the set of strings derivable from A. This rule by itself suffices in many grammars.

Example :- Consider the following grammar for arithmetic expressions.

$$E \to E+T \mid T$$
$$T \to T*F \mid F \qquad \text{.....(4)}$$
$$F \to (E) \mid id$$

Eliminating the immediate left recursion (productions of the form $A \to A\alpha$) to the productions for E and to for T, we obtain,

$$E \to TE'$$
$$E' \to +TE' \mid \epsilon$$
$$T \to FT' \qquad \text{.....(5}$$
$$T' \to *FT' \mid \epsilon$$
$$F \to (E) \mid id$$

No matter how many A-productions there are, we can eliminate immediate left recursion from them by the following technique. First, we group the A-productions as,

$$A \to A\alpha_1 \mid A\alpha_2 \mid \cdots \mid A\alpha_m \mid \beta_1 \mid \beta_2 \mid \cdots \mid \beta_n$$

where no β_i begins with an A. Then we replace the A-productions by,

$$A \to \beta_1 A' \mid \beta_2 A' \mid \cdots \mid \beta_n A'$$

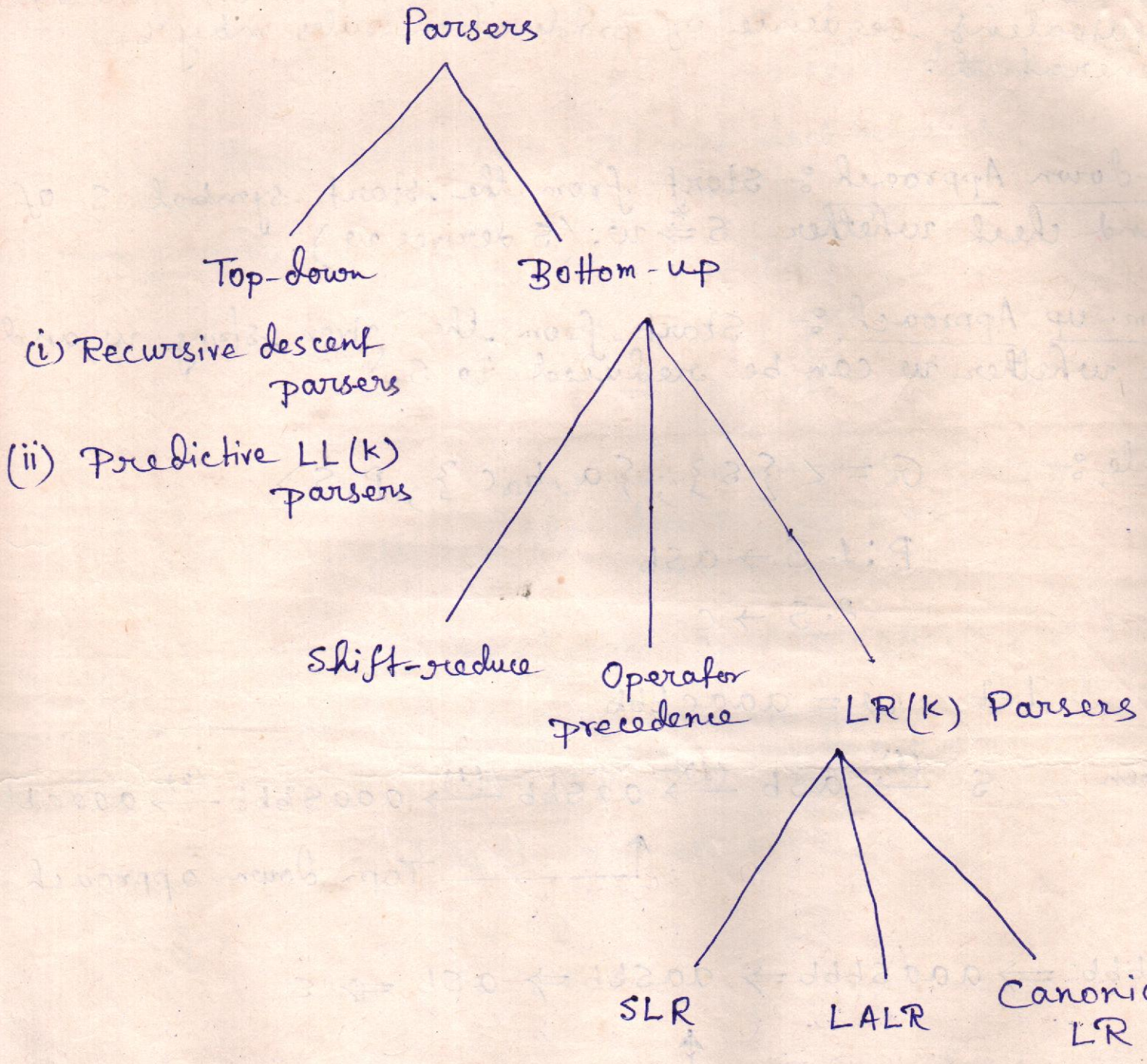
$$A' \to \alpha_1 A' \mid \alpha_2 A' \mid \cdots \mid \alpha_m A' \mid \epsilon$$

The nonterminal A generates the same strings as bef but is no longer left recursive. This procedure eliminates all immediate left recursion from the A and A' productions (provided no α_i is ϵ), but it doe not eliminate left recursion involving derivations o two or more steps. For example, consider the gram

$$S \to Aa \mid b$$

$$A \to Ac \mid Sd \mid \epsilon$$

The nonterminal S is left-recursive because $S \Rightarrow Aa \Rightarrow Sda$, but it is not immediately left recur



LL(k) ≡ left-to-right scan of the input, produc
a leftmost derivation with at most k sy
lookahead on the input.

LR(k) ≡ left-to-right scan of the input, produce a rightmost derivation in reverse, with atmost k symbols lookahead as the input string.

To write a recursive descent parser for grammar we need to do the following :-

(i) Check whether G is ambiguous or not ; if yes disambiguate it.

(ii) Reconstruct the grammar to remove any left-recursion

(iii) Left-factor the grammar if necessary.

(iv) Write recursion routines for each nonterminal.

Left Factoring :- Left factoring is a grammar transformation that is useful for producing a grammar suitable for predictive parsing. The basic idea is that when it is not clear which of two alternative productions to use to expand a nonterminal A, we may be able to rewrite the A-productions to defer the decision until we have seen enough of the input to make the right choice

For example, if we have the two productions,

$$stmt \rightarrow \text{if } expr \text{ then } stmt \text{ else } stmt$$
$$\mid \text{if } expr \text{ then } stmt$$

on seeing the input token if, we cannot immediately t
which production to choose to expand stmt. In general,
if $A \rightarrow \alpha\beta_1 \mid \alpha\beta_2$ are two A-productions, and the input
begins with a nonempty string derived from α, we do
not know whether to expand A to $\alpha\beta_1$ or to $\alpha\beta_2$. How
we may defer the decision by expanding A to $\alpha A'$. The
after seeing the input derived from α, we expand A
to β_1 or to β_2. That is, left-factored, the origina
productions become,

$$A \rightarrow \alpha A'$$
$$A' \rightarrow \beta_1 \mid \beta_2$$

Example :- The following grammar abstracts the
dangling-else problem :-

$$S \rightarrow iEtS \mid iEtSeS \mid a \qquad \ldots\ldots (1)$$
$$E \rightarrow b$$

Here i, t, and e stand for if, then and else,
E and S for "expression" and "statement".
Left-factored, this grammar becomes :-

$$S \rightarrow iEtSS' \mid a$$
$$S' \rightarrow eS \mid \epsilon \qquad \ldots\ldots (2)$$
$$E \rightarrow b$$

Thus we may expand S to iEtSS' on input i and wait until iEtS has been seen to decide whether to expand S' to eS or to ϵ. Of course, grammar (1) and (2) are both ambiguous, and on input e, it will not be clear which alternative for S' should be chosen.

Example :- Fig 1 contains a collection of transition diagrams for grammar (5). The only ambiguities concern whether or not to take an ϵ-edge. If we interpret the edges out of the initial state for E' as saying take the transition on + whenever that is the next input and take the transition on ϵ otherwise, and make the analogous assumption for T', then the ambiguity is removed and we can write a predictive parsing program for grammar (5)

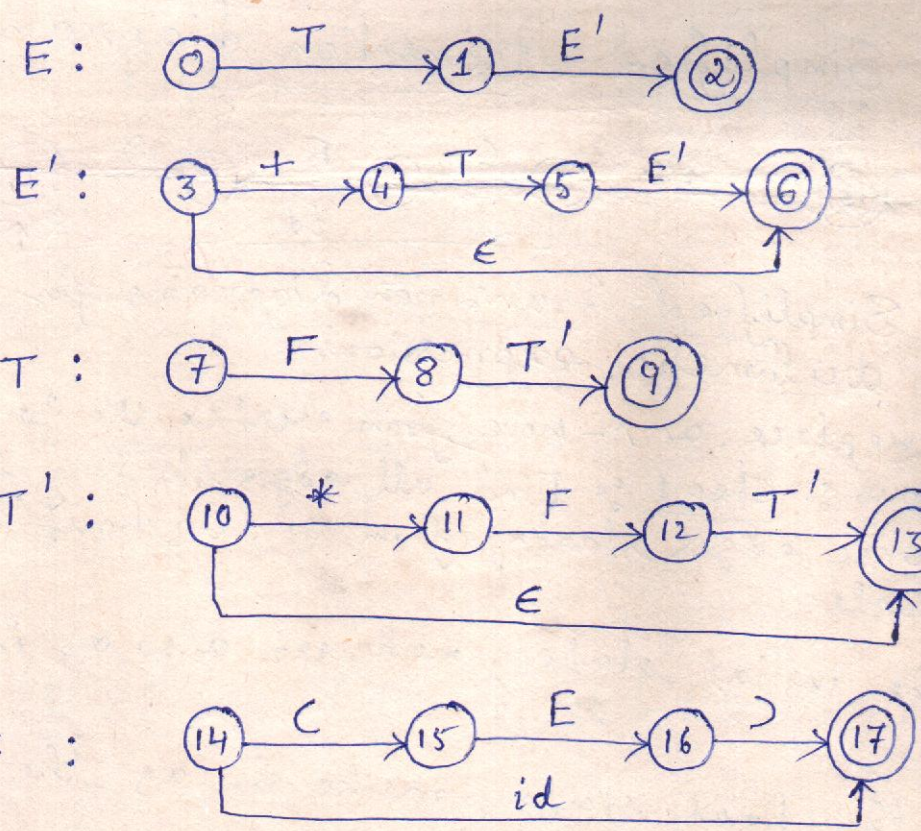


Fig (1) :- Transition diagrams for grammar (5).

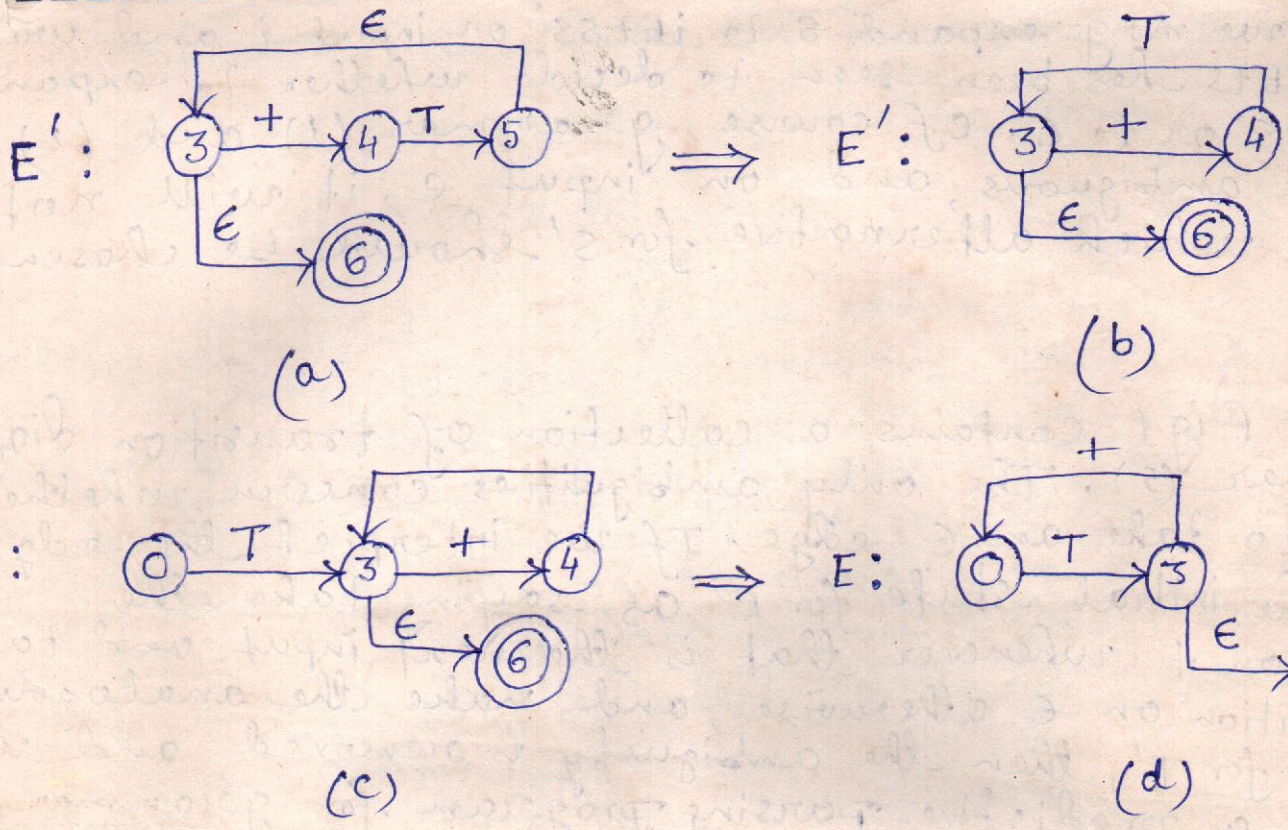


Fig (2) :- Simplified transition diagrams

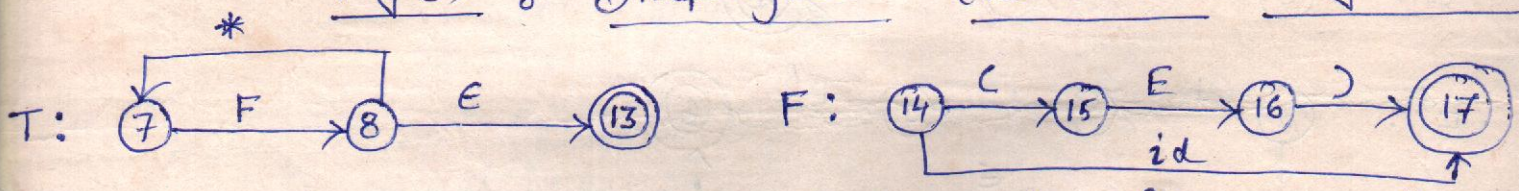


Fig (3) :- Simplified transition diagrams for arithmetic expressions.

Suppose we want to replace a $\wedge$-move from vertex v_1 to vert
Then we proceed as follows :- Step 1 :- Find all edges starting from
Step 2 :- Duplicate all these edges starting from v_1, without chan
the edge labels.

Step 3 :- If v_1 is the initial state, make v_2 also as initia
state.

Step 4 :- If v_2 is the final state, make v_1 as the fina
state.