CS 304 Compiler Design

Text books

1. Compilers – Principles, Techniques & Tools , Aho, Ravi Sethi, D. Ullman

- **Introduction to Syntax Analysis:- Role of the Syntax Analyser – Syntax error handling.**
- **Review of Context Free Grammars - Derivation and Parse Trees, Eliminating Ambiguity.**
- **Basic parsing approaches - Eliminating left recursion, left factoring.**
- **Top-Down Parsing - Recursive Descent parsing, Predictive Parsing, LL(1) Grammars.**

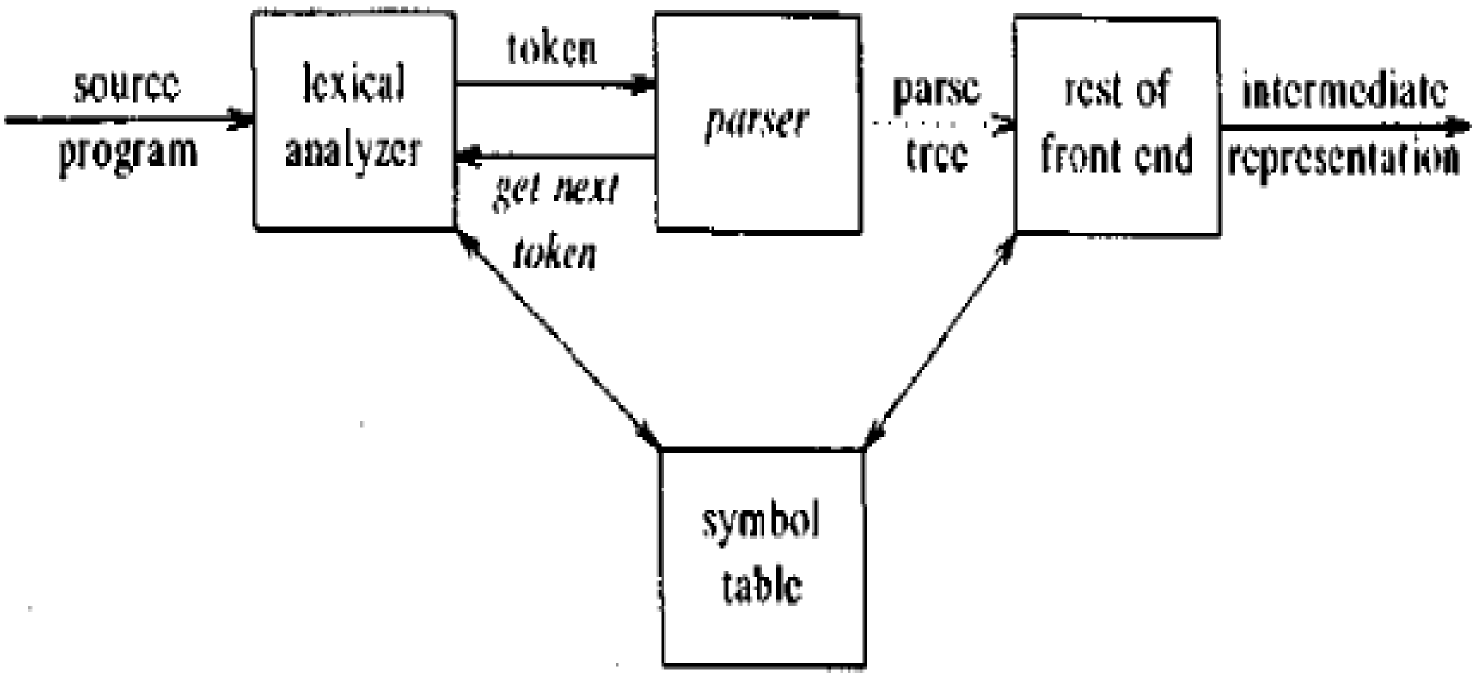
Syntax Analysis

- Every programming language has rules that prescribe the syntactic structure of well-formed programs
- In Pascal, a program is made out of blocks, a block out of statements, a statement out of expressions, an expression out of tokens and so on
- The syntax of programming language constructs can be described by context free grammars or BNF notation

The Role of the Parser

- Obtains a string of tokens from the lexical analyzer
- Verifies that the string can be generated by the grammar for the source language (by constructing parse trees)
- Reports any syntax errors
- Recover from commonly occurring errors so that it can continue processing the remainder of the input

Fig 4.1 Position of parser in compiler model



Context Free Grammars

- Programming language constructs have a recursive structure that can be defined by context free grammars
- Eg : Conditional statement can be defined by a rule

  If S_1 and S_2 are statements and E is an expression, then

  "if E then S_1 else S_2" is a statement 4.1
- This cannot be defined by notations of regular expressions

- Using the syntactic variable *stmt* to denote the class of statements and *expr* the class of expressions, 4.1 can be expressed using the grammar production

$$stmt \rightarrow \textbf{if}\ expr\ \textbf{then}\ stmt\ \textbf{else}\ stmt \qquad 4.2$$

- CFG consists of terminals, non terminals, a start symbol and productions
- <u>Terminals</u>
  - Basic symbols from which strings are formed
  - A token
  - Keywords if, then and else are terminals

- Nonterminals
  - Syntactic variables that denote sets of strings
  - Eg : stmt and expr
- Start symbol
  - One special nonterminal
- Productions
  - Specify the manner in which the terminals and nonterminals can be combined to form strings
  - Consists of a nonterminal followed by an arrow, followed by a string of nonterminals and terminals

Productions for simple arithmetic expressions

expr	→	*expr op expr*
expr	→	*(expr)*
expr	→	- *expr*
expr	→	**id**
op	→	+
op	→	-
op	→	*
op	→	/
op	→	↑

Notational conventions

1. Terminals
   - Lower case letters early in the alphabet such as a,b,c
   - Operator symbols such as +, - etc
   - Punctuation symbols such as parentheses, comma etc
   - The digits 0,1,2….9
   - Boldface strings such as **id** or **if**

2. Nonterminals

- Upper case letters early in the alphabet such as A,B,C
- The letter S, usually the start symbol
- Lower case italic names such as expr or stmt

3. Upper case letters late in the alphabet such as X,Y,Z represent grammar symbols that are either terminals or nonterminals
4. Lower case letters late in the alphabet, chiefly u,v,...z represent strings of terminals
5. Lower case Greek letters α,β,γ represent strings of grammar symbols ($A\rightarrow\alpha$)
6. If $A\rightarrow\alpha_1$, $A\rightarrow\alpha_2$.....$A\rightarrow\alpha_k$ are productions with A on the left (A-productions). Can be written as $A\rightarrow\alpha_1|\alpha_2|\ldots|\alpha_k$. $\alpha_1,\alpha_2,\ldots\alpha_k$ are alternatives of A
7. Unless otherwise stated, the left side of the first production is the start symbol

Grammar of 4.2

$E \rightarrow E\ A\ E \mid (E) \mid -E \mid id$

$A \rightarrow + \mid - \mid * \mid / \mid \uparrow$

DERIVATIONS

- Gives a precise description of the top-down construction of a parse tree
- Production is treated as a rewriting rule in which the nonterminal on the left is replaced by the string on the right side of the production

$$E \rightarrow E + E \mid E * E \mid (E) \mid -E \mid id \qquad 4.3$$

- A single E can be replaced with, say –E
- This action is described by writing $E \Rightarrow -E$
- Read as E derives –E
- $E \Rightarrow -E \Rightarrow -(E) \Rightarrow -(id)$

- Such a sequence of replacements is a derivation of –(id) from E
- We say that $\alpha A\beta \Rightarrow \alpha\gamma\beta$ if $A \rightarrow \gamma$ is a production and α and β are arbitrary strings of grammar symbols

- $\alpha_1 \Rightarrow \alpha_2 \Rightarrow \ldots\ldots \Rightarrow \alpha_n$, we say α_1 derives α_n
- $\Rightarrow$ derives in one step
- $\overset{*}{\Rightarrow}$ derives in zero or more steps

- $\alpha \overset{*}{\Rightarrow} \alpha$ for any string α
- If $\alpha \overset{*}{\Rightarrow} \beta$ and $\beta \Rightarrow \gamma$, then $\alpha \overset{*}{\Rightarrow} \gamma$

- One or more steps $\overset{+}{\Rightarrow}$

- Given a grammar G with start symbol S, we can use the $\overset{+}{\Rightarrow}$ relation to define L(G)
- Strings in L(G) contains only terminal symbols of G
- A string of terminals w is in L(G) if and only if $S \overset{+}{\Rightarrow} w$
- w is called a ***sentence*** of G
- A language that can be generated by a grammar is said to be a CFL

- If two grammars generate the same language, the grammars are said to be equivalent
- If $S \overset{*}{\Rightarrow} \alpha$, where α may contain non-terminals, we say that α is a sentential form of G
- A sentence is a sentential form with no non-terminals
- Example
- The string –(id+id) is a sentence of grammar 4.3 as there is a derivation 4.4

$$E \Rightarrow -E \Rightarrow -(E) \Rightarrow -(E+E) \Rightarrow -(id+E) \Rightarrow -(id+id) \qquad (4.4)$$

- At each step in a derivation, we need to choose which non-terminal to replace
  - Left most
  - Right most
- There are parsers in which only the left most non-terminal in any sentential form is replaced
- Such derivations are called leftmost
- If $\alpha \Rightarrow \beta$ by a step in which the left most non-terminal in α is replaced, we write $\alpha \underset{lm}{\Rightarrow} \beta$
- Rewrite the previous derivation

$$E \underset{lm}{\Rightarrow} -E \underset{lm}{\Rightarrow} -(E) \underset{lm}{\Rightarrow} -(E+E) \underset{lm}{\Rightarrow} -(\mathbf{id}+E) \underset{lm}{\Rightarrow} -(\mathbf{id}+\mathbf{id})$$

- Every left most step can be written

$$wA\gamma \underset{lm}{\Rightarrow} w\delta\gamma$$

  - w consists of terminals only
  - A->δ is the production applied and
  - γ is a string of grammar symbols
- $\alpha \underset{lm}{\Rightarrow} \beta$,denotes that α derives β by a left most derivation
- If $S \underset{lm}{\Rightarrow} \alpha$, then α is a left-sentential form of the grammar
- Similarly we can define right most derivations
- Right most derivations are also called as Canonical derivations

Parse trees and derivations

- Parse tree – A graphical representation for a derivation that filters out the choice regarding replacement order
- Each interior node of a parse tree is labeled by some nonterminal A
- The children of the node are labeled from left to right, by symbols in the right side of the production by which A was replaced in the derivation
- The leaves are labeled by nonterminals or terminals read from left to right, constitute the sentential form, called the yield or frontier of the tree

- Parse tree for 4.4 –(id+id)

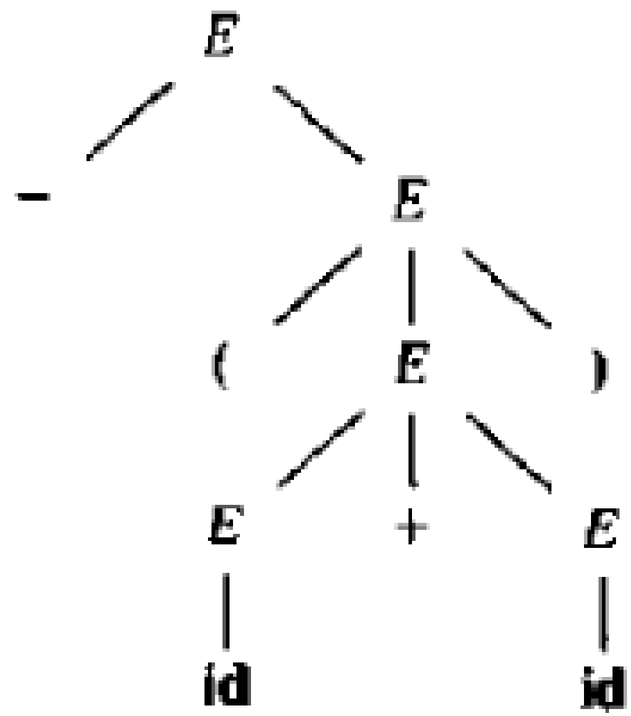


Fig. 4.2. Parse tree for $-($**id** $+$ **id**$)$.

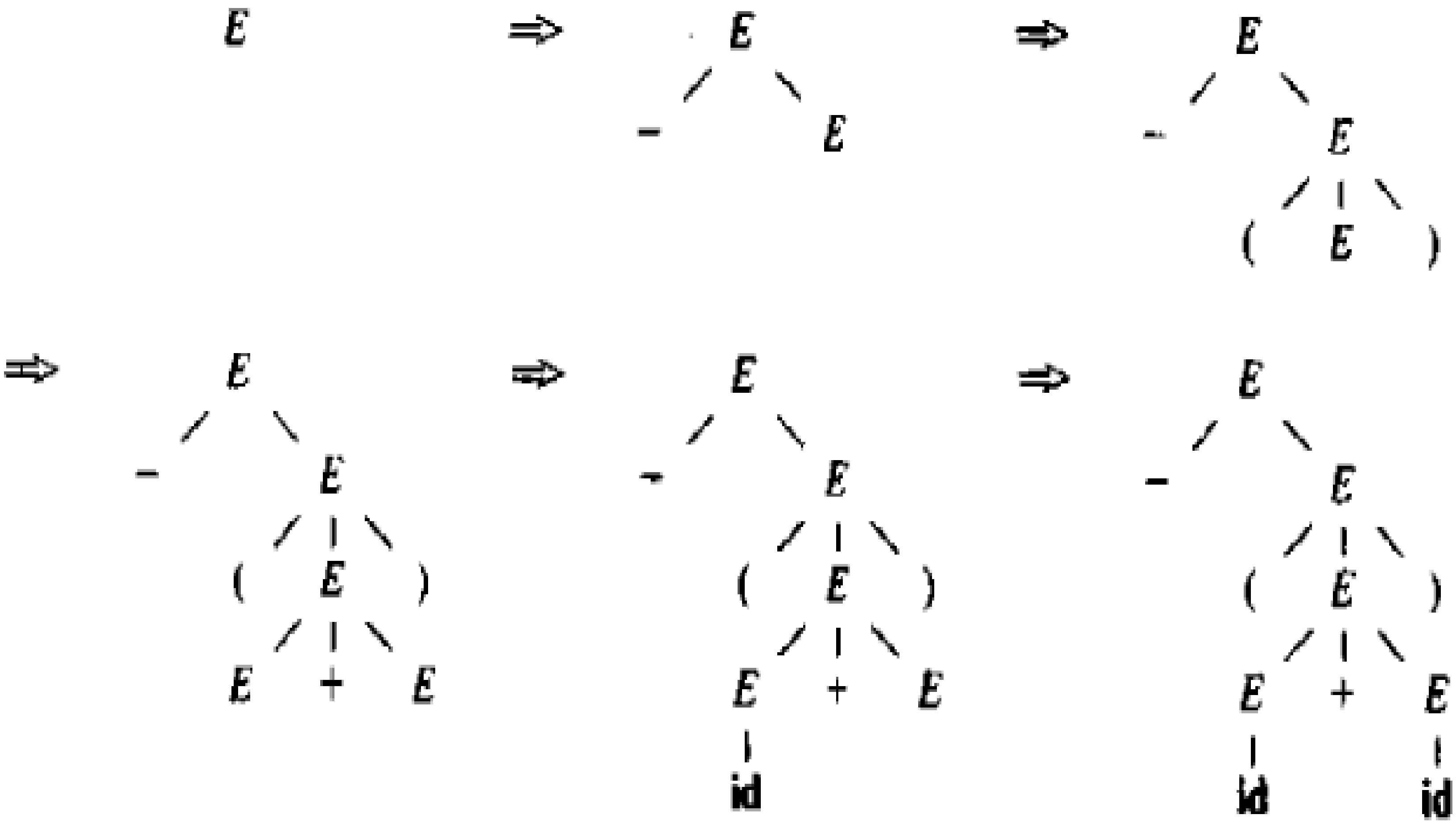
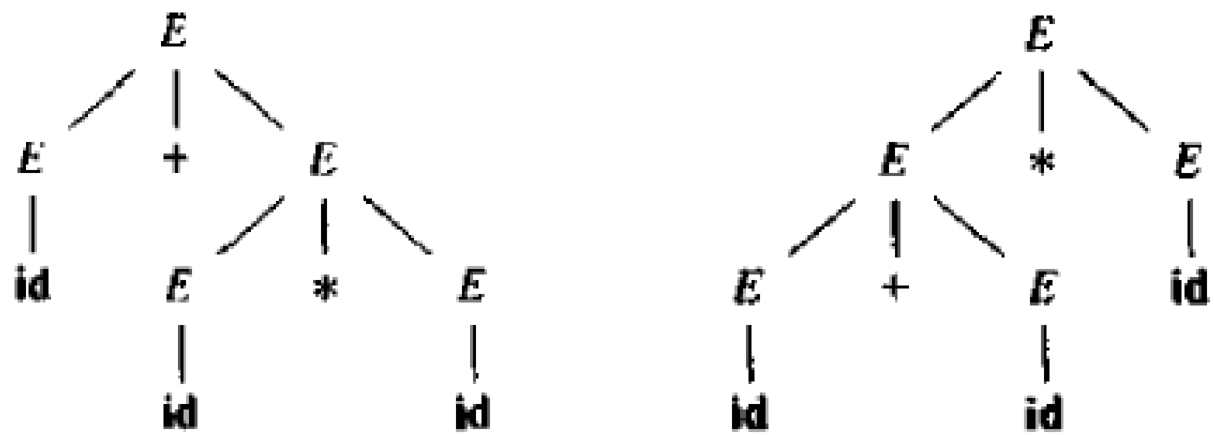


Fig. 4.3. Building the parse tree from derivation (4.4).

- Derivation for the sentence id+id*id
- Parse tree for the same

$$
\begin{aligned}
E &\Rightarrow E+E \\
&\Rightarrow \mathbf{id}+E \\
&\Rightarrow \mathbf{id}+E*E \\
&\Rightarrow \mathbf{id}+\mathbf{id}*E \\
&\Rightarrow \mathbf{id}+\mathbf{id}*\mathbf{id}
\end{aligned}
\qquad\qquad
\begin{aligned}
E &\Rightarrow E*E \\
&\Rightarrow E+E*E \\
&\Rightarrow \mathbf{id}+E*E \\
&\Rightarrow \mathbf{id}+\mathbf{id}*E \\
&\Rightarrow \mathbf{id}+\mathbf{id}*\mathbf{id}
\end{aligned}
$$



(a) (b)

Fig. 4.4. Two parse trees for **id+id*id**.

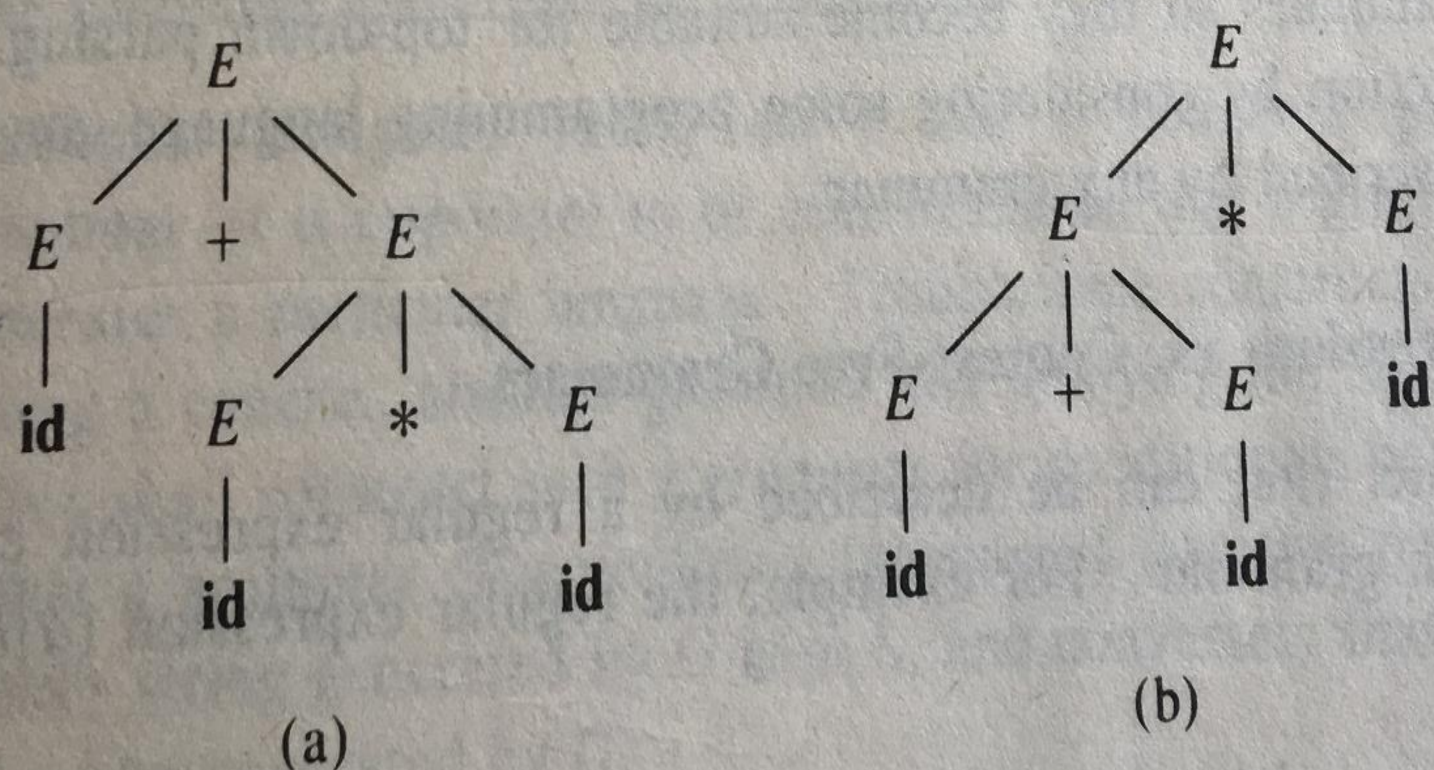


Fig. 4.4. Two parse trees for **id+id*id**.

Ambiguity

- A grammar that produces more than one parse tree for some sentence is said to be ambiguous
- A grammar that produces more than one left most or more than one right most derivations for the same sentence

Eliminating ambiguity

- "dangling-else" grammar

  stmt -> if expr then stmt

  | if expr then stmt else stmt

  | other
- other stands for any other statement
- This is an ambiguous grammar

- Consider the parse tree for the grammar
- if E_1 then S_1 else if E_2 then S_2 else S_3

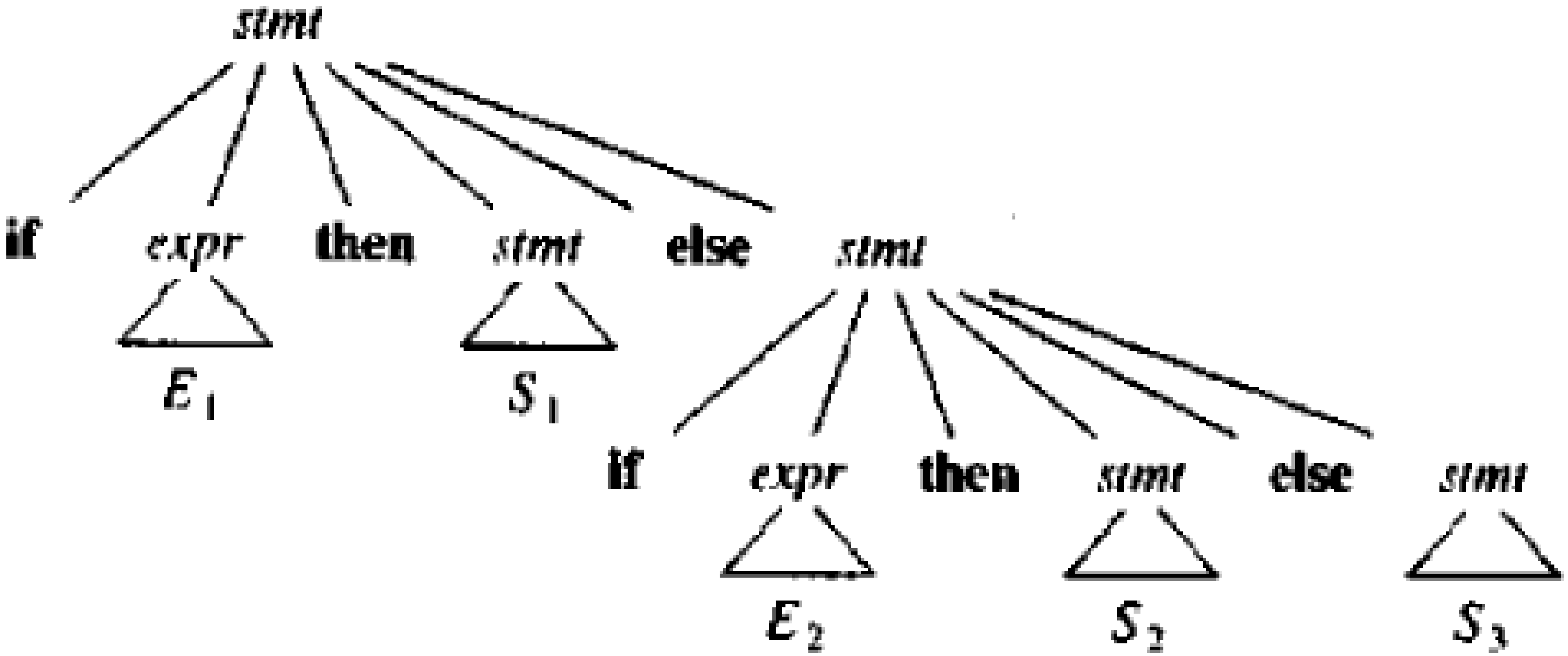
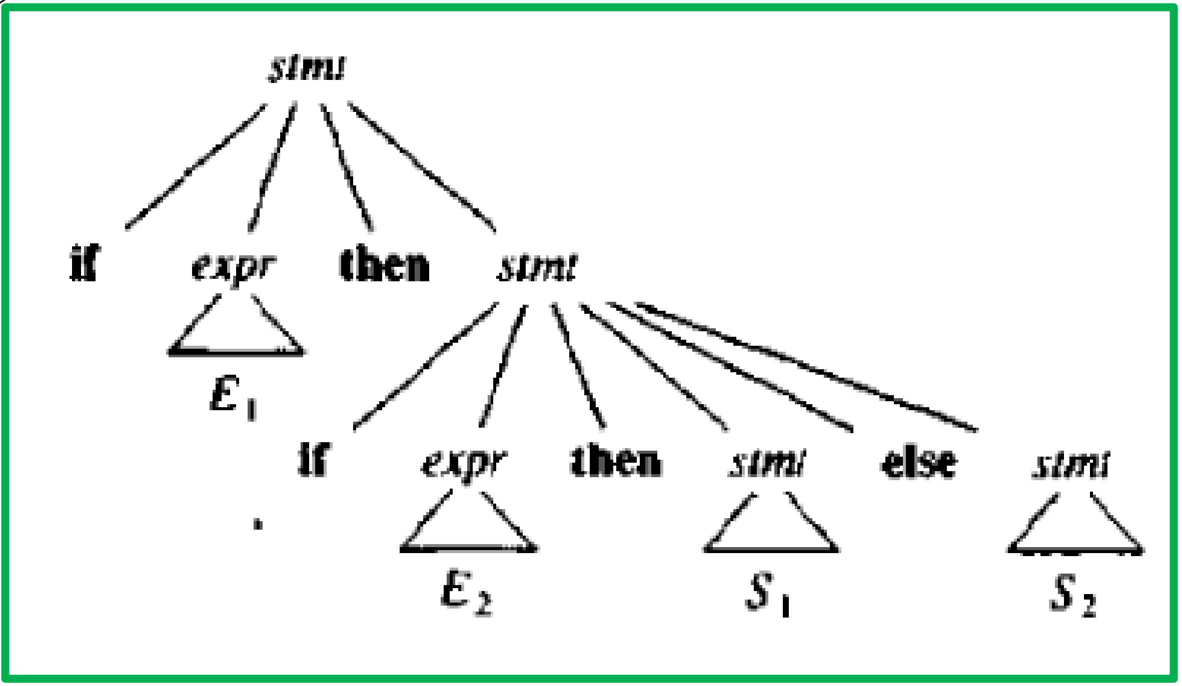
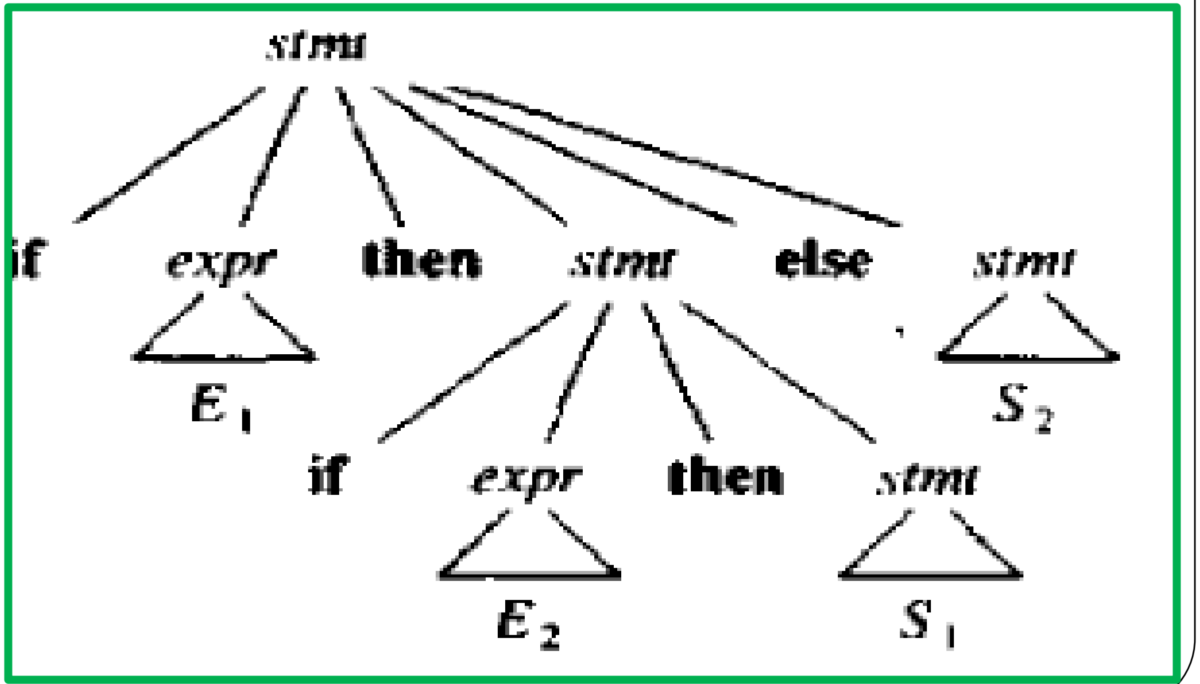


Fig. 4.5. Parse tree for conditional statement.

- Parse tree for the grammar

  if E1 then if E2 then S1 else S2


stmt
if
expr
then
stmt
E_1
if
expr
then
stmt
else
stmt
E_2
S_1
S_2


stmt
if
expr
then
stmt
else
stmt
E_1
S_2
if
expr
then
stmt
E_2
S_1

- The general rule is “Match each else with the closest previous then”

- Rewrite the grammar as follows
  - A statement appearing between a then and an else must be matched
  - ie, it must not end with an unmatched then followed by any statement, as this else would 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 other kind of conditional statement
  - Thus,

Modified grammar

```
    stmt -> matched_stmt
          | unmatched_stmt
          | other
matched_stmt -> if expr then matched_stmt else matched_stmt
          | other

unmatched_stmt -> if expr then stmt
          |if expr then matched_stmt else unmatched_stmt
```

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 cannot handle left-recursive grammars, so a transformation that eliminates left recursion is needed

- Immediate left recursion and not immediate

Eliminating immediate left recursion

- Productions of the form

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

- Replace such productions with

  $$A \rightarrow \beta A'$$

  $$A' \rightarrow \alpha A' \mid \varepsilon$$

- Eg
- Grammar for arithmetic expression

  $$E \rightarrow E + T \mid T$$

  $$T \rightarrow T * F \mid F$$

  $$F \rightarrow (E) \mid id$$

E -> E + T | T

Here A= E, α = +T and β = T

E -> TE'

E' -> +TE' | ε

T -> T * F | F

Here A=T, α = *F and β = F

E -> FT’

E’ -> *FT’| ε

- Thus we have

E -> TE'

E' -> +TE' | ε

E -> FT'

E' -> *FT'| ε

F -> (E) | id

- Can be eliminated no matter how many A-productions are there
- As a first step, group A productions as

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

Where no β_i begins with an A

- Then replace such productions by

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

$$A' \to \alpha_1 A' \mid \alpha_2 A' \mid \ldots \mid \alpha_m A' \mid \varepsilon$$

S-> (L) | a
L->L,S | S

bexpr -> bexpr or bterm | bterm
bterm -> bterm and bfactor | bfactor
bfactor -> not bfactor | (bexpr) | true | false

R -> R `| ` R | RR| R* | (R) | a | b

S-> SS | (S) | ε

E ->E+T | T
T -> TF | F
F -> F * |a|b

E -> E sub R | E sup E | { E } |c

R -> E sup E | E

- But, this method does not eliminate left recursion involving derivations of two or more steps
- Eg : Consider the grammar

$$S \rightarrow Aa \mid b$$

$$A \rightarrow Ac \mid Sd \mid \varepsilon \qquad 4.12$$

Eliminating left recursion

- Algorithm 4.1 Eliminating left recursion
- Input : Grammar G with no cycles or ε productions
- Output : An equivalent grammar with no left recursion
- Method : Apply the following algorithm to G.

1. Arrange the nonterminals in some order $A_1, A_2, \ldots, A_n$.

2. **for** i := 1 **to** n **do begin**

    for j := 1 **to** $i-1$ **do begin**

        replace each production of the form $A_i \rightarrow A_j\gamma$

            by the productions $A_i \rightarrow \delta_1\gamma \mid \delta_2\gamma \mid \cdots \mid \delta_k\gamma$,

            where $A_j \rightarrow \delta_1 \mid \delta_2 \mid \cdots \mid \delta_k$ are all the current A_j-productions;

    end

    eliminate the immediate left recursion among the A_i-productions

end

Fig. 4.7. Algorithm to eliminate left recursion from a grammar.

- Apply the algorithm to 4.12

- Yields

$$S \rightarrow Aa \mid b$$
$$A \rightarrow bdA' \mid A'$$
$$A' \rightarrow cA' \mid adA' \mid \varepsilon$$

Left factoring

- A grammar transformation suitable for producing a grammar for predictive parsing
- Basic idea : When it is not clear which of the two alternative productions to use to expand a nonterminal A, we may be able to defer the decision until we have seen enough of the input to make the right choice

- Eg :

  stmt -> if expr then stmt else stmt

  | if expr then 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 $\alpha\beta_1 \mid \alpha\beta_2$
- We may then defer the decision by expanding A to $\alpha A'$
- After seeing the input derived from α,we expand A' to β_1 or β_2

- ie, left-factored, the original productions become

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

- Algorithm : Left factoring a grammar
- Input : Grammar G
- Output : An equivalent left-factored grammar
- Method : For each non-terminal A, find the longest prefix α common to two or more of its alternatives. If $\alpha \neq \varepsilon$, ie, there is a nontrivial common prefix, replace all the A productions $A \to \alpha\beta_1 \mid \alpha\beta_2 \mid \ldots \mid \alpha\beta_n \mid \gamma$ where γ represents all alternatives that do not begin with α by

$$A \rightarrow \alpha A' \mid \gamma$$

$$A' \rightarrow \beta_1 \mid \beta_2 \mid \dots \mid \beta_n$$

- Here A' is a nonterminal
- Repeatedly apply this transformation until no two alternatives for a nonterminal have a common prefix

- The following grammar abstracts the dangling-else problem

$$S \rightarrow iEtS \mid iEtSeS \mid a$$

$$E \rightarrow b$$

- Here i,t and e stand for if,then and else, E, S for "expressions" and "statement"

- Left-factored, this grammar becomes,

  S -> iEtSS’ | a

  S’ -> eS | ε

  E -> b

1. Consider the grammar

$$S \rightarrow (\ L\ )\ \mid\ a$$

$$L \rightarrow L\ ,\ S \mid S$$

a. What are the terminal, nonterminals and start symbol?

b. Find the parse trees for the following sentences :

i. (a,a)

ii. (a, (a,a))

iii. (a,(a,a),(a,a))

c. Construct the left most derivation for each of the sentences in (b)

d. Construct the right most derivations for each of the sentences in (b)

e. What language does this grammar generate?

2. Consider the grammar

$$S \rightarrow aSbS \mid bSaS \mid \varepsilon$$

a. Show that this grammar is ambiguous by constructing two different leftmost derivations for the sentence abab
b. Construct the corresponding right most derivations for abab
c. Construct the corresponding parse trees for abab
d. What language does this grammar generate?

3. Consider the grammar

$$bexpr \rightarrow bexpr \textbf{ or } bterm \mid bterm$$

$$bterm \rightarrow bterm \textbf{ and } bfactor \mid bfactor$$

$$bfactor \rightarrow \textbf{not } bfactor \mid (bexpr) \mid \textbf{true} \mid \textbf{false}$$

a. Construct a parse tree for the sentence **not(true or false)**
b. Show that this grammar generates all boolean expressions
c. Is this grammar ambiguous? Why?