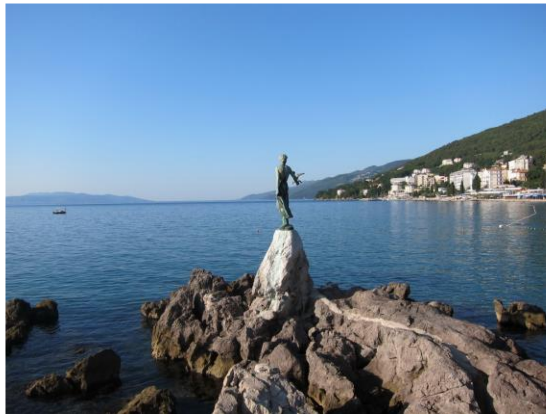
# Chapter 5

# Context-Free Grammars and Languages

(2015/11)



Adriatic Sea shore at Opatija, Croatia

## Outline

5.0 Introduction

5.1 Context-Free Grammars (CFG’s)

5.2 Parse Trees

5.3 Applications of CFG’s

5.4 Ambiguity in Grammars and Languages

## 5.0 Introduction

- **Concepts** ---
  - *Context-free grammars* (CFG's) generate *context-free languages* (CFL's).
  - CFG's play a central role in compiler technology since 1960's.

- **Applications of CFG's ---**
  - Recently, CFG's are used to describe document formats via document-type definitions (DTD's) which are used in XML's (extensible markup languages).
  - XML's are used mainly for information exchange on the Web.

## 5.1 Context-free Grammars (CFG's)

### 5.1.1 An Informal Example

- *Palindromes* ---
  - A palindrome is a word, phrase, number, or other sequence of symbols or elements, whose meaning may be interpreted the same way in either forward or reverse direction.
  - e.g., otto, madamimadam ("Madam, I'm Adam")…

- **An Example ---**

  Design a CFG for the language of binary palindromes $L_{pal} = \{w \mid w \in \{0, 1\}^*, w^R = w\}$.

  - Examples of strings in $L_{pal}$: 00, 0110, 011110, …
  - CFG ---

$$
\begin{array}{ll}
P \rightarrow \varepsilon & (1) \\
P \rightarrow 0 & (2) \\
P \rightarrow 1 & (3) \\
P \rightarrow 0P0 & (4) \\
P \rightarrow 1P1 & (5)
\end{array}
$$

    - Productions (4) & (5) are *recursive*.
    - $P$ is a *variable* (*nonterminal*).
    - 0 and 1 are *terminals.*

  - Examples of derivations of strings ---
    - $P \Rightarrow_{(4)} 0P0 \Rightarrow_{(5)} 01P10 \Rightarrow_{(5)} 011P110 \Rightarrow_{(1)} 011110$
    - $P \Rightarrow_{(4)} 0P0 \Rightarrow_{(3)} 010$ …

### 5.1.2 Definition of CFG's

- A CFG is a 4-tuple $G = (V, T, P, S)$ where
  - $V$ is the set of *variables (or nonterminals, syntactic categories)*;
  - $T$ is the set of *terminals*;
  - $S$ is the *start symbol*;
  - $P$ is the set of *productions or rules* of the form "head"→ "body"

    $A \rightarrow \alpha$

    where $A \in V$, $\alpha \in (V \cup T)^*$, i.e., the head is a single variable and the body is a string

of zero or more terminals and variables.

- **An Example (the last example continued)** ---
  - $V = \{P\}$
  - $T = \{0, 1\}$
  - $P = A$ = the set of the five productions (1)~(5) below:

    $P \rightarrow \varepsilon$ (1)

    $P \rightarrow 0$ (2)

    $P \rightarrow 1$ (3)

    $P \rightarrow 0P0$ (4)

    $P \rightarrow 1P1$ (5)
  - $S = P$.

  (*Note*: *P is used for two meanings here; do not get confused!*)

- **Example 5.3** ---

  A CFG $G_r$ for arithmetic expressions with productions as follows:

  - $E \rightarrow I$
  - $E \rightarrow E + E$
  - $E \rightarrow E*E$
  - $E \rightarrow (E)$

  where $I$ is an identifier describable by RE (**a** + **b**)(**a** + **b** + **0** + **1**)* and transformable into productions of the following forms:

  - $I \rightarrow a$
  - $I \rightarrow b$
  - $I \rightarrow Ia$
  - $I \rightarrow Ib$
  - $I \rightarrow I0$
  - $I \rightarrow I1$

  ♦ The above productions may be rewritten integrally as

  - $E \rightarrow I \mid E + E \mid E*E \mid (E)$
  - $I \rightarrow a \mid b \mid Ia \mid Ib \mid I0 \mid I1$

  ♦ An example of string derivations ---

  $E \Rightarrow E*E \Rightarrow I*E \Rightarrow a*E \Rightarrow a*(E) \Rightarrow a*(E + E) \Rightarrow a*(I0 + E) \Rightarrow ... \Rightarrow a*(a0 + b1)$

### 5.1.3 <u>Derivations Using a Grammar</u>

- **Concept:** we use productions to generate ("infer") strings in the language described by the grammar.

- Two ways for such *string inference* ---

  ♦ *Recursive inference* ---
  - bottom up ("from body to head");
  - starting from known strings (often from terminals in productions).

  ♦ *Derivation* ---
  - top down ("from head to body") in concept;

- as shown by the examples given before.

■ **Example 5.4** ---

Show a bottom-up recursive inference of the string $w = a*(a + b00)$ using the productions of the CFG $G_r$ described previously.

♦ A recursive inference of $w$ may be described by Fig. 5.1.
♦ Note: $w$ above is not a regular expression but an arithmetic expression.

| Derivation steps | String inferred | For language of | Production used | String(s) used |
|---|---|---|---|---|
| (*i*) | $a$ | $I$ | 5: $(I \rightarrow a)$ | – |
| (*ii*) | $b$ | $I$ | 6: $(I \rightarrow b)$ | – |
| (*iii*) | $b0$ | $I$ | 9: $(I \rightarrow I0)$ | (*ii*) |
| (*iv*) | $b00$ | $I$ | 9: $(I \rightarrow I0)$ | (*iii*) |
| (*v*) | $a$ | $E$ | 1: $(E \rightarrow I)$ | (*i*) |
| (*vi*) | $b00$ | $E$ | 1: $(E \rightarrow I)$ | (*iv*) |
| (*vii*) | $a + b00$ | $E$ | 2: $(E \rightarrow E + E)$ | (*v*), (*vi*) |
| (*viii*) | $(a + b00)$ | $E$ | 4: $(E \rightarrow (E))$ | (*vii*) |
| (*iv*) | $a*(a + b00)$ | $E$ | 3: $(E \rightarrow E*E)$ | (*v*), (*viii*) |

Fig. 5.1 Inference of a string $w = a*(a+b00)$.

■ **Notations used in derivations ---**

If $\alpha A \beta$ is a string of terminals and variables, and if $A \rightarrow \gamma$ is a production, then we write

$$\alpha A \beta \underset{G}{\Rightarrow} \alpha\gamma\beta$$

to denote a *derivation*.

♦ For zero and more derivations, we use the following notation.

$$A \overset{*}{\underset{G}{\Rightarrow}} \gamma$$

♦ The label $G$ under the double arrow may be omitted if which grammar is being used is understood.

■ **Example 5.5 ---**

A derivation of the string $w = a*(a+b00)$ is as follows.

$$E \Rightarrow E*E \Rightarrow I*E \Rightarrow a*E \Rightarrow a*(E) \Rightarrow a*(I + E) \Rightarrow a*(a + E)$$

$$\Rightarrow a*(a + I) \Rightarrow a*(a + I0) \Rightarrow a*(a + I00) \Rightarrow a*(a + b00)$$

### 5.1.4 <u>Leftmost and Rightmost Derivations</u>

■ **Definitions ---**

♦ *Leftmost derivation* --- replacing the leftmost variable in each derivation step (represented by the notation $\underset{lm}{\Rightarrow}$ or, for typing convenience, also by $\Rightarrow_{lm}$)

- *Rightmost derivation* --- rightmost instead (represented by $\underset{rm}{\Rightarrow}$ or by $\Rightarrow_{rm}$).

■ **Example 5.6** (continuation of Example 5.5) **---**

- The leftmost derivation of the string $w = a*(a+b00)$ of Example 5.5 is as follows.

$$E \underset{lm}{\Rightarrow} E*E \underset{lm}{\Rightarrow} I*E \underset{lm}{\Rightarrow} a*E \underset{lm}{\Rightarrow} a*(E) \underset{lm}{\Rightarrow} a*(E+E)$$
$$\underset{lm}{\Rightarrow} a*(I+E) \underset{lm}{\Rightarrow} \ldots \underset{lm}{\Rightarrow} a*(a+I00) \underset{lm}{\Rightarrow} a*(a+b00).$$

- The rightmost derivation of $w$ is as follows.

$$E \Rightarrow_{rm} E*E \Rightarrow_{rm} E*(E) \Rightarrow_{rm} E*(E+E) \Rightarrow_{rm} E*(E+I) \Rightarrow_{rm} E*(E+I0)$$
$$\Rightarrow_{rm} E*(E+I00) \Rightarrow_{rm} \ldots \Rightarrow_{rm} I*(a+b00) \Rightarrow_{rm} a*(a+b00).$$

- Any derivation has an equivalent leftmost derivation and an equivalent rightmost one (proved in the next section).

### 5.1.5 The Language of a Grammar

■ **Definition ---**

The language $L(G)$ of a CFG $G = (V, T, P, S)$ is

$$L(G) = \{w \mid w \in T^*, S \overset{*}{\underset{G}{\Rightarrow}} w\}.$$

■ The language of a CFG is called a context-free language (CFL).

■ Theorem 5.7 in the text book shows a typical way to prove that a given grammar really generates the desired language (the set of palindromes) (using induction) (read by yourself).

■ Derivations from the start symbol are called *sentential forms*.

- Given a CFG $G = (V, T, P, S)$, if $S \overset{*}{\Rightarrow} \alpha$ with $\alpha \in (V \cup T)^*$, then $\alpha$ is a sentential form.
- If $S \overset{*}{\underset{lm}{\Rightarrow}} \alpha$ where $\alpha \in (V \cup T)^*$, then $\alpha$ is a left-sentential form.
- If $S \overset{*}{\underset{rm}{\Rightarrow}} \alpha$ where $\alpha \in (V \cup T)^*$, then $\alpha$ is a right-sentential form.

## 5.2 Parse Trees

■ **Advantage of parse trees –**

- In a compiler, the *parse tree* structure facilitates *translation* of the source program into recursive executable codes.
- Parse trees are closely related to derivations and recursive inferences.
- An important application of the parse tree is the study of *grammatical ambiguity* which

makes the grammar unsuitable for a programming language.

- **A note about the term YACC in textbook ---**
  - The computer program YACC is a *parser generator* developed by Stephen C. Johnson at AT&T for the Unix operating system.
  - The name is an acronym for "Yet Another Compiler Compiler."
  - It generates a parser (the part of a compiler that tries to make syntactic sense of the source code) based on an analytic grammar written in a notation similar to BNF.
  - YACC generates the code for the parser in the C programming language. (Retrieved from Wikipedia, 2007/10/8)

## 5.2.1 Constructing Parse Trees

- **Definition ---**

  Given a grammar $G = (V, T, P, S)$, the parse tree is defined in the following way.

  - Each interior node is labeled by a variable in $V$.
  - Each leaf is labeled by either a variable, a terminal, or ε.
  - If ε is the label, it must be the only child of its parent.
  - If an interior node is labeled $A$, and its children are labeled $X_1, X_2, \ldots, X_k$, respectively, from the left, then $A \to X_1X_2\ldots X_k$ is a production in $P$.

- A Note --- the only time one of the $X$'s can be ε is when ε is the label of the only child, and $A \to \varepsilon$ is a production of $G$.

- **Example 5.10 ---**

  A parse tree of derivation $P \overset{*}{\Rightarrow} 0110$ of the palindrome is shown in Fig. 5.2.

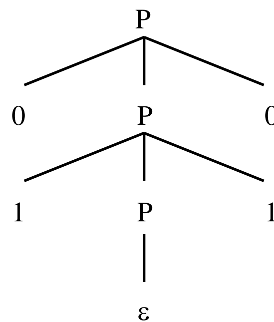



Fig. 5.2 A parse tree of Example 5.10.

## 5.2.2 The Yield of a Parse Tree

- The *yield* of a parse tree is the string obtained by concatenating all the leaves from the left, like 01ε10= 0110 for the tree of the last example (Example 5.10).

- Showing the yields of the parse trees of a grammar $G$ is another way to describe the

language of $G$ (provable).

### 5.2.3 Inference, Derivations, and Parse Trees

- **Theorems ---**

  Given a grammar $G = (V, T, P, S)$, the following facts are all equivalent:

  - the recursive inference procedure determines that terminal string $w$ is in the language of variable $A$;
  - $A \overset{*}{\Rightarrow} w$;
  - $A \overset{*}{\underset{lm}{\Rightarrow}} w$;
  - $A \overset{*}{\underset{rm}{\Rightarrow}} w$;
  - there is a parse tree with root $A$ and yield $w$.

- A note: equivalences of the facts in the previous page are proved in a way as shown in the diagram of Fig. 5.3 by theorems found in Sections 5.2.4~5.2.6 of the textbook (read by yourself).

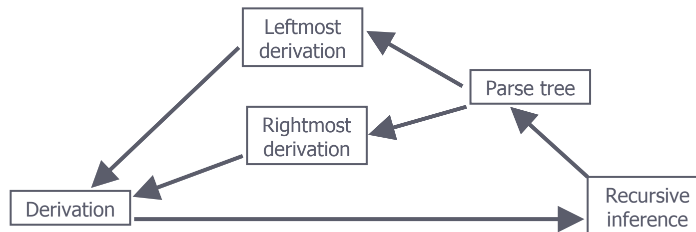



Fig. 5.3 Equivalences of ways to generate strings based on grammars.

## 5.3 Applications of CFG's

- History --- CFG's were originally conceived by N. Chomsky for describing natural languages, but not all natural languages can be so described.

- **Two applications of CFG's ---**
  - Describing programming languages ---
    - There is a mechanical way for turning a CFG into a parser.
  - Describing DTD's in XML's ---
    - DTD --- document type definition
    - XML --- extensible markup language

### 5.3.1 Parsers

- There are components in programming languages which are not RL's.
- Two examples of non-RL structures ---
  - Balanced structures ---

    like parentheses "( )," "begin-end" pair, "if-else" pair, …

- Unbalanced structures ---
  like *unbalanced* “if else” pairs…
  - e.g., a balanced if-else pair in C is

    *if (condition) Statement; else Statement;*

- **Example 5.19 ---**
  - A grammar for balanced parentheses is as follows.

    $G_{bal} = (\{B\}, \{(, )\}, P, B)$

    where

    $P: B \rightarrow BB \mid (B) \mid \varepsilon$

    - e.g., strings generated by $G_{bal}$ include $x$ = (()), ()(), etc.

  - A grammar for unbalanced if-else pairs is as follows.

    $S \rightarrow \varepsilon \mid SS \mid iS \mid iSeS$

    where $i$ = **if** …, $e$ = **else** …
    - The production $S \rightarrow iS$ is used to generate unbalance “**if**” (with no matching “**else**”)
    - e.g., the following is a generated segment of a C program

```
if (Condition) {
    …
    if (Condition) Statement;
       else Statement;
    …
    if (Condition) Statement;
       else Statement;
    …
}
```

### 5.3.2 The YACC Parser-Generator

- Input to YACC is a CFG with each production being associated additionally with an action (see {…} in Example 5.21 below).

- **Example 5.21 ---**

  A CFG in the YACC notation identical to that of Example 5.3 is shown in Fig. 5.4 (the right part).

```
Exp:  Id                     {…}
      |  Exp '+' Exp         {…}
      |  Exp '*' Exp         {…}
      |  '(' Exp ')'         {…}
      ;
Id:   'a'
      |  'b'                 {…}
      |  Id 'a'              {…}
      |  Id 'b'              {…}
      |  Id '0'              {…}
      |  Id '1'              {…}
```

Fig. 5.4 A CFG in the YACC notation identical to that of Example 5.3.

### 5.3.3 Markup Languages

- The strings in a markup language are documents with "marks," called *tags* which specify semantics of the strings.

- **An example ---**

  The HTML (HyperText Markup Language) for webpage design includes two functions:

  - ♦ creating links between documents;
  - ♦ describing formats ("looks") of documents.

- **Example 5.22 ---**

  The HTML of a webpage is shown in Fig. 5.5, where the left part shows the text seen on the webpage, and right part shows the HTML source defining the webpage content. Also the tags in the right part include:

  - ♦ <P>: defining a paragraph; (unmatched single tag)
  - ♦ <EM>…</PM>: emphasizing a text part; (matched tag pair)
  - ♦ <OL>…</OL>: defining an ordered list of items; (matched tag pair)
  - ♦ <LI>: defining a list item. (unmatched single tag)

---

The thing I *hate*:

1. Moldy bread.

2. People who drive too slow in the fast lane.

```
<P>The thing I <EM> hate </EM>
<OL>
<LI>Moldy bread.
<LI>People who drive too slow in the
     fast lane.
</OL>
```

---

Fig. 5.5 The HTML of a webpage.

- **Part of an HTML grammar ---**

  The following productions are part of those of an HTML grammar.

  1. *Doc* → | *Element Doc*
  2. *Element* → *Text* | <EM> *Doc* </EM> | <P> *Doc* | <OL> *List* </OL> | …
  3. *Text* → | *Char Text*
  4. *Char* → *a* | *A* | …
  5. *List* → | *Listitem List*
  6. *Listitem* → <LI> *Doc*

### 5.3.4 XML and Document-Type Definitions (DTD's)

- An XML (extensible markup language) describes the *semantics* of the text in a document using DTD's, while an HTML describes the *format* of a document.

- The DTD is itself described with a language which is essential *in the form of a CFG mixed with regular expressions*.
  - ♦ The form of a *DTD* is

    <!DOCTYPE name-of-DTD [list of element definitions]>

  - ♦ The form of an *element definition* is

    <!Element element-name (description of the element)>

  - ♦ Descriptions of elements are essentially regular expressions, which may be described in the following way:
    - An element may appear in another element.
    - The special term #PCDATA stands for any text involving no tags.
    - The allowed operators are
      - * A vertical bar | means union;
      - * A comma , denotes concatenation
      - * 3 variants of the closure operator --- *, +, ? as described before:

        (1) *: zero or more occurrences of;

        (2) +: one or more occurrences of;

        (3) ?: zero or one occurrence of.

- **Example 5.23** ---

  Fig. 5.6 (Fig. 5.14 in the textbook) shows a DTD for describing personal computers (PC's).

  - ♦ Each element is represented in the document by a tag with the name of the element and a matching tag at the end with an extra slash, just as in HTML.

```
<!DOCTYPE PcSpecs [
<!ELEMENT PCS (PC*)>
<!ELEMENT PC (MODEL, PRICE, PROCESSOR, RAM, DISK+)>
<!ELEMENT MODEL (#PCDATA)>
<!ELEMENT PRICE (#PCDATA)>
<!ELEMENT PROCESSOR (MANF, MODEL, SPEED)>
<!ELEMENT MANF (#PCDATA)>
<!ELEMENT MODEL (#PCDATA)>
<!ELEMENT SPEED (#PCDATA)>
<!ELEMENT RAM (#PCDATA)>
<!ELEMENT DISK (HARDDISK | CD | DVD)>
<!ELEMENT HARDDISK (MANF, MODEL, SIZE)>
 <!ELEMENT SIZE (#PCDATA)>
<!ELEMENT CD (SPEED)>
 <!ELEMENT DVD (SPEED)>
]>
```

Fig. 5.6 A DTD for describing personal computers (PC's).

  - ♦ For examples:
    - for <!ELEMENT MODEL (#PCDATA)>, an example is:

```
<MODEL>4560</MODEL>
```

    - for <!ELEMENT PCS (PC*)>, an example is:

```
<PCS>
    <PC>
        …
    </PC>
```

```
        <PC>
            ...
        </PC>
    </PCS>
```

- A description for two PC's using the DTD language is shown in Fig. 5.7 (Fig.5.15 in the textbook):

```
<PCS>
    <PC>
        <MODEL>4560</MODEL>
        <PRICE>$2295</PRICE>
        <PROCESSOR>
            <MANF>Intel</MANF>
            <MODEL>Pentium</MODEL>
            <SPEED>800MHz</SPEED>
        </PROCESSOR>
        <RAM>256</RAM>
        <DISK><HARDDISK>
            <MANF>Maxtor</MANF>
            <MODEL>Diamond</MODEL>
            <SIZE>30.5Gb</SIZE>
        </HARDDISK></DISK>
        <DISK><CD>
            <SPEED>32x</SPEED>
        </CD></DISK>
    </PC>
    <PC>
    ...
    </PC>
</PCS>
```

Fig. 5.7 Part of a document obeying DTD of Fig. 5.14.

- Examples of converting DTD rules, which include RE's, into CFG productions ---
  - <!ELEMENT PROCESSOR (MANF, MODEL, SPEED)>

    $\Rightarrow$ *Processor* $\rightarrow$ *Manf Model Speed*

    (Note: the commas mean concatenations)

  - <!ELEMENT DISK (HARDDISK | CD | DVD)>

    $\Rightarrow$ *Disk* $\rightarrow$ *Harddisk* | *Cd* | *Dvd*

    (Note: the vertical bars mean the same as in production bodies)

  - <!ELEMENT PC (MODEL, PRICE, PROCESSOR, RAM, DISK+)>

    $\Rightarrow$ *Pc* $\rightarrow$ *Model Price Processor Ram Disks*

    *Disks* $\rightarrow$ *Disk* | *Disk Disks*

    (Note: the 2nd rule corresponds to the RE DISK+ which means one or more disks)

- General induction technique for converting DTD rules, which includes RE's, into legal CFG productions ---
  - *Basis*:

    If the body of a production $P$ is a concatenation of elements, then $P$ is in the legal form for CFG's.
  - *Induction*:

    Assuming $E_1$ and $E_2$ are in legal forms, we have the following rules:

1. $A \rightarrow E_1, E_2 \quad \Rightarrow \quad (1)\ A \rightarrow BC \qquad (2)\ B \rightarrow E_1 \qquad (3)\ C \rightarrow E_2$

2. $A \to E_1 \mid E_2$ $\Rightarrow$ (1) $A \to E_1$ (2) $A \to E_2$
3. $A \to (E_1)^*$ $\Rightarrow$ (1) $A \to BA$ (2) $A \to \varepsilon$ (3) $B \to E_1$
4. $A \to (E_1)+$ $\Rightarrow$ (1) $A \to BA$ (2) $A \to B$ (3) $B \to E_1$
5. $A \to (E_1)?$ $\Rightarrow$ (1) $A \to \varepsilon$ (2) $A \to E_1$

- **Example 5.24 ---**

Try to use the above rules to convert the following into legal CFG productions:

<!ELEMENT PC (MODEL, PRICE, PROCESSOR, RAM, DISK+)> (a)

- By Rule 1: (a) $\Rightarrow Pc \to AB$
  $A \to Model\ Price\ Processor\ Ram$
  $B \to Disk+$ *(illegal)* (b)
- By Rule 4: (b) $\Rightarrow B \to CB \mid C$
  $C \to Disk$
- By observation, $A$ and $C$ may be eliminated, so the final result is

$Pc \to Model\ Price\ Processor\ Ram\ B$
$B \to Disk\ B \mid Disk$ *(compare with the last result!)*

- **Example 5.24a (supplemental) ---**

Convert the following into legal CFG productions:

<!ELEMENT PCS (PC*)> (c)

- Rule 3: $A \to (E_1)^* \Rightarrow$ (1) $A \to BA$ (2) $A \to \varepsilon$ (3) $B \to E_1$
- Therefore, (c) above may be converted in the following way:
  (c) $\Rightarrow Pcs \to B\ Pcs$
  $Pcs \to \varepsilon$
  $B \to Pc$

## 5.4 Ambiguity in Grammars & Languages

### 5.4.1 Ambiguous Grammars

- **Definition ---**

A CFG $G = (V, T, P, S)$ is *ambiguous* if there exists at least one string $w$ in $T^*$ for which there are two different parse trees, each with root labeled $S$ and yield $w$ *identically*.

If each string has at most one parse tree in $G$, then $G$ is *unambiguous*.

- Ambiguity causes problems in program compiling.

- **Example 5.25** ---

Given the productions of the arithmetic expression grammar $G_r$ of Example 5.3 as follows,

$$E \rightarrow I \mid E + E \mid E*E \mid (E)$$
$$I \rightarrow a \mid b \mid Ia \mid Ib \mid I0 \mid I1$$

the parse trees of the following two derivations

$$E \Rightarrow E + E \Rightarrow E + E*E$$
$$E \Rightarrow E*E \Rightarrow E + E*E$$

are shown in Fig. 5.8. The two parse trees obviously are *distinct*.

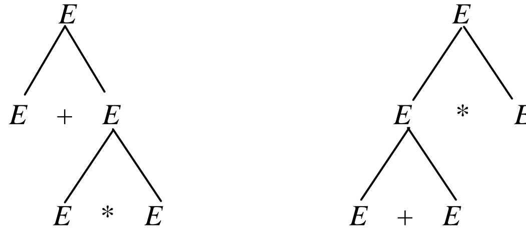



Fig. 5.8 Two different parse trees for the derivation $E \Rightarrow E + E*E$.

- Ambiguity in certain grammars may be removed by re-designing the grammar.
- But some CFL's are "*inherently ambiguous*," i.e., every grammar has more than one distinct parse tree for each of *some strings* in the language.
- Note: two different derivations might have the same parse tree (see Example 5.26 below). So, it is *not* multiplicity of derivations that causes ambiguity.

- **Example 5.26** ---

  Two derivations of the string $a + b$ are as follows:

  $E \Rightarrow E + E \Rightarrow I + E \Rightarrow a + E \Rightarrow a + I \Rightarrow a + b$
  $E \Rightarrow E + E \Rightarrow E + I \Rightarrow I + I \Rightarrow I + b \Rightarrow a + b$

  - The parse trees for the above two derivations may be checked easily to be the same.
  - So, it is *not* multiplicity of derivations that causes ambiguity**.**

### 5.4.2 Removing Ambiguity from Grammars

- **Concept ---**
  - There is *no* general algorithm that can tell whether a grammar is ambiguous or not. That is, testing of grammatical ambiguity is *undecidable*.
  - Ambiguity in *inherently ambiguous* grammars is also *irremovable*.
  - But elimination of ambiguity in some common programming language structures is possible.

- **Example 5.27 ---**

  Remove the ambiguity of the arithmetic expression grammar of Example 5.3 whose productions are repeated below:

$$E \rightarrow I \mid E + E \mid E*E \mid (E)$$
$$I \rightarrow a \mid b \mid Ia \mid Ib \mid I0 \mid I1$$

  - Trick for the removal: creating "terms," which *cannot be broken*, as the units of the

generated expressions (for details, see the textbook).

♦ An unambiguous version of the original expression grammar is as follows ($T$ represents "term"):

$$E \to T \mid E + T$$
$$T \to F \mid T * F$$
$$F \to I \mid (E)$$
$$I \to a \mid b \mid Ia \mid Ib \mid I0 \mid I1$$

♦ A check of the parse tree for $E + E*E$ --- *now only the one shown in Fig. 5.9 can be obtained now; no other alternative!*

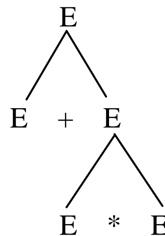



Fig. 5.9 Two different parse trees for the derivation $E \Rightarrow E + E*E$.

### 5.4.3 Leftmost Derivations as a Way to Express Ambiguity

- In an unambiguous grammar, the leftmost derivation is *unique* (so is the rightmost one), as shown by the following theorem.

- **Theorem 5.29 ---**

  For each grammar $G = (V, T, P, S)$ and string $w$ in $T^*$, $w$ has two distinct parse trees if and only if $w$ has two distinct leftmost (rightmost) derivations.

  *Proof.* See the textbook.

### 5.4.4 Inherent Ambiguity

- A CFL $L$ is said to be *inherently ambiguous* if all its grammars are ambiguous.

- An example of inherently ambiguous languages

  $$L = \{a^n b^n c^m d^m \mid n \geq 1, m \geq 1\} \cup \{a^n b^m c^m d^n \mid n \geq 1, m \geq 1\}.$$

  *Proof.* See the textbook for the proof of its inherent ambiguity.