Context free grammars - II

Prof. A. Morzenti

AMBIGUITY

Examples from natural language

"I saw the man with the binoculars."
"La pesca è bella"
"half baked chicken"
"Questa è una rapina, dateci i soldi altrimenti ***spariamo***." "OK, allora ***sparite***"...

In artificial, technical languages ambiguity must be ruled out (in the natural ones…).

We only consider SYNTACTIC AMBIGUITY:

A sentence *x* of a grammar *G* is ambiguous if it admits several distinct syntax trees

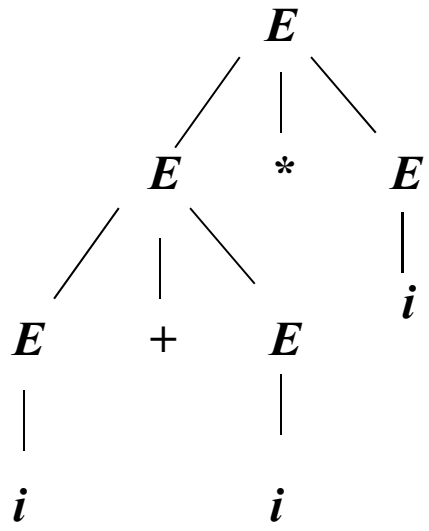
In such a case we say that the grammar *G* is ambiguous

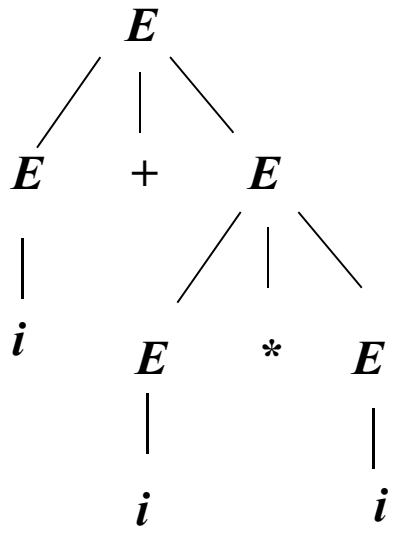
EXAMPLE: a "simple" grammar *G*' for arithmetic expressions (with bilateral recursion)

$$E \rightarrow E + E \mid E * E \mid (E) \mid i$$

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

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





the sentence $i+i+i$ is also ambiguous

G' is ambiguous and **does not enforce** the usual **precedence of product over sum**

NB: *G*' is smaller than the previous *G*: it has only 1 n.t. and 4 rules
in grammar design there is ofted a **trade-off** between size and ambiguity

The ***degree of ambiguity of a sentence*** x of a language $L(G)$

is the number of distinct trees of x compatible with G

for a ***grammar*** the degree of ambiguity

is the maximum among the degree of ambiguity of its sentences

Notice that the degree of ambiguity of sentences of a grammar can be unlimited

IMPORTANT PROBLEM: determine if a grammar is ambiguous

The ***problem*** is ***undecidable***:
there does not exist a general algorithm that, given any free grammar,
terminates (after a finite number of steps) with the correct answer

The ***absence of ambiguity*** in a specific grammar can be shown

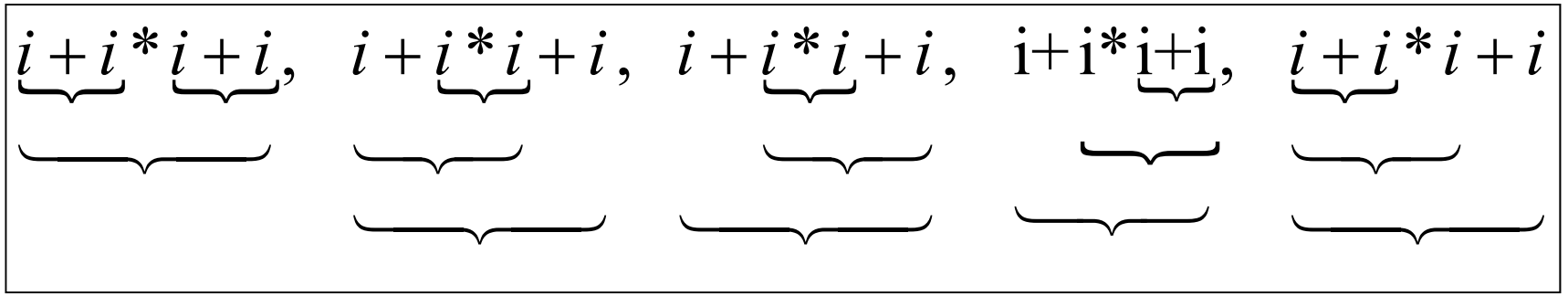
on a case by case basis, by hand, through ***inductive reasonings***,
hence by analyzing a finite number of cases

Instead, to show ambiguity one can exhibit a ***witness***: an ambiguous sentence

BEST APPROACH: AVOID AMBIGUITY IN THE GRAMMAR DESIGN PHASE

Example: arithmetic expressions with rule $E \rightarrow E + E \mid E * E \mid i$

The degree of ambiguity is 2 for $i + i + i$ and 5 for $i + i * i + i$



For longer sentences the degree of ambiguity increases with no limit

CATALOG OF AMBIGUOUS FORMS AND REMEDIES

1) AMBIGUITY FROM BILATERAL RECURSION: $A \rightarrow A \ \ldots \ A$

Example 1: grammar G_1 generates $i + i + i$ in two different ways (check!).

$$\boxed{G_1 : \quad E \rightarrow E + E \mid i}$$

But $L(G_1) = i\ (+i)^*$ is a regular language: other, simpler grammars are possible

Nonambiguous right-recursive grammar : $\boldsymbol{E \rightarrow i + E \mid i}$

Nonambiguous left-recursive grammar : $\boldsymbol{E \rightarrow E + i \mid i}$

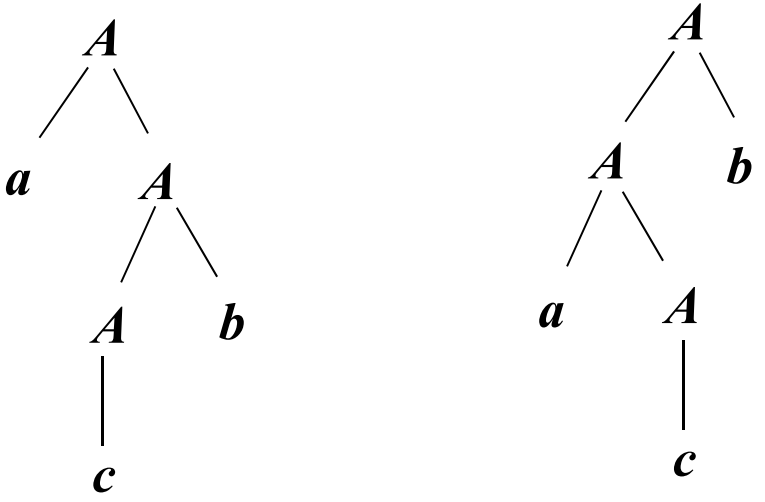
Example 2: Left and right recursion in different rules

$L(G_2) = a^*cb^*$

G_2 admits derivations where the ***a*** and ***b*** in a given sentence are obtained *in any order*

$$G_2: \quad A \rightarrow aA \mid Ab \mid c$$

Simplest example: two syntax trees for sentence *acb*



Example 2 (follows): Left and right recursion in different rules

$$L(G_2) = a^* cb^*$$

First method to eliminate ambiguity:
Generate the two lists by distinct nonterminals and rules

$$S \to AcB$$

$$A \to aA \mid \varepsilon$$

$$B \to bB \mid \varepsilon$$

$$L(G_2) = a^* cb^*$$

Second method: force an ***order in the derivations***:
ex., first generate the *a*'s then the *b*'s

$$S \to aS \mid X$$

$$X \to Xb \mid c$$

2) AMBIGUITY FROM LANGUAGE UNION

If L_1=$L(G_1)$ and L_2=$L(G_2)$ share some sentences (nonempty intersection)

The grammar G for the union language is ambiguous

A sentence $x \in L_1 \cap L_2$ admits two distinct derivations,

One with the rules of G_1 and the other with the rules of G_2

It is ambiguous for a grammar G that includes all the rules

Instead the sentences that belong to $L_1 \setminus L_2$ and to $L_2 \setminus L_1$ are nonambiguous

Remedy: provide disjointed set of rules for $L_1 \cap L_2$, $L_1 \setminus L_2$, and $L_2 \setminus L_1$

Examples in the textbook and in exam exercises

3) INHERENT AMBIGUITY (of a ***language***)

A language is INHERENTLY AMBIGUOUS if all its grammars are ambiguous

EXAMPLE:

$$L = \{a^i b^j c^k \mid i = j \vee j = k\} = \{a^i b^i c^* \mid i \geq 0\} \cup \{a^* b^i c^i \mid i \geq 0\} = L_1 \cup L_2$$

generates $\boldsymbol{a^i b^i c^*}$

G_1

$S_1 \rightarrow XC$

$X \rightarrow aXb \mid \varepsilon$

$C \rightarrow cC \mid \varepsilon$

G_2

$S_2 \rightarrow AY$

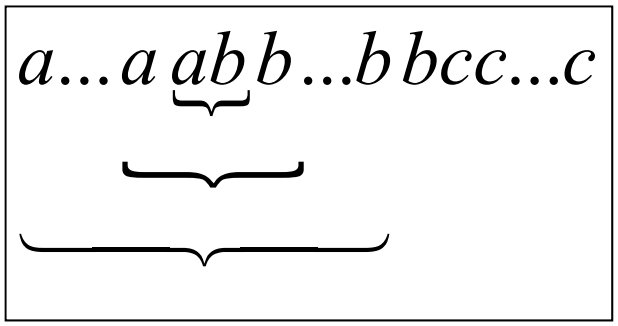
$Y \rightarrow bYc \mid \varepsilon$

$A \rightarrow aA \mid \varepsilon$

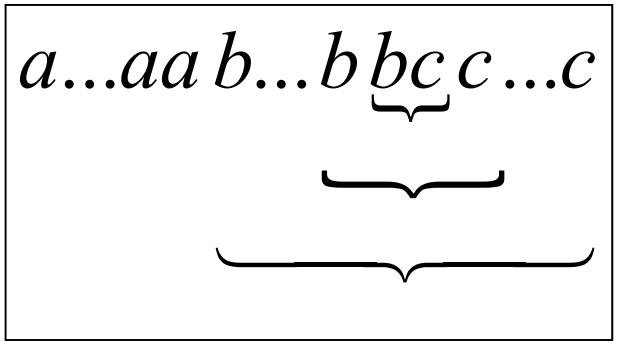
generates $\boldsymbol{a^* b^i c^i}$

The union grammar is ambiguous for the sentences ε, abc, $a^2b^2c^2$, ...
(which constitute a ***non***-context free language)

They are generated by G_1 which has rules ensuring that $|x|_a = |x|_b$ …



… and also by G_2, with rules ensuring that $|x|_b=|x|_c$



We ***intuitively*** argue that ***any*** grammar for L is ambiguous

Luckily inherent ambiguity is rare and can be avoided in technical languages

4) AMBIGUITY FROM CONCATENATION OF TWO LANGUAGES

G for the concatenation of L_1 and L_2

$$G = (\Sigma_1 \cup \Sigma_2, \{S\} \cup V_{N_1} \cup V_{N_2}, \{S \to S_1 S_2\} \cup P_1 \cup P_2, S)$$

Ambiguity from concatenation occurs if there exist strings x, y, z such that

$x \in L_1$ $\quad$ $yz \in \mathrm{L}_2$ $\quad$ $xy \in L_1$ $\quad$ $z \in L_2$

hence the string xyz has two left derivations (and two syntax trees)

$$S \Rightarrow S_1 S_2 \overset{+}{\Rightarrow} x S_2 \overset{+}{\Rightarrow} xyz$$

$$S \Rightarrow S_1 S_2 \overset{+}{\Rightarrow} xy S_2 \overset{+}{\Rightarrow} xyz$$

Example – Ambiguity in the concatenation of Dyck languages

$$\Sigma_1 = \{a, a', b, b'\} \quad \Sigma_2 = \{b, b', c, c'\}$$

$$aa'bb'cc' \in L = L_1 L_2$$

$$G(L): \quad S \to S_1 S_2$$

$$S_1 \to aS_1a'S_1 \mid bS_1b'S_1 \mid \varepsilon$$

$$S_2 \to bS_2b'S_2 \mid cS_2c'S_2 \mid \varepsilon$$

$$\overbrace{aa'bb'}^{S_1}\overbrace{cc'}^{S_2} \qquad \overbrace{aa'}^{S_1}\overbrace{bb'cc'}^{S_2}$$

To prevent ambiguity one must avoid the shift of the substring from the suffix in the first language to the prefix in the second one

Easy solution: insert a ***new*** terminal symb. (e.g. '#') acting as a separator

***new terminal symb.*:** the symbol may not belong to any of the two alphabets

$L_1 \# L_2$ is generated by the «concatenation grammar» with the axiomatic rule $S \to S_1 \# S_2$

The language is however modified

5) OTHER CASES OF AMBIGUITY

AMBIGUOUS REGULAR EXPRESSIONS

every sentence with two or more
c is ambiguous

$$S \to DcD \quad D \to bD \mid cD \mid \varepsilon$$

$$\{b,c\}^* \, c \, \{b,c\}^*$$

REMEDY: identify the leftmost c

$$\boldsymbol{S \to BcD \quad B \to bB \mid \varepsilon \quad D \to bD \mid cD \mid \varepsilon}$$

OTHER CASE: **LACK OF ORDER IN DERIVATIONS**

Example: linear grammar G,
$\forall x \in L(G) \;\; |x|_c \leq |x|_b \leq 2 \cdot |x|_c$
Every phrase x with $|x|_c < |x|_b < 2 \cdot |x|_c$ is ambiguous

$$S \to bSc \mid bbSc \mid \varepsilon$$

$$S \Rightarrow bbSc \Rightarrow bbbScc \Rightarrow bbbcc$$

$$S \Rightarrow bSc \Rightarrow bbbScc \Rightarrow bbbcc$$

REMEDY: Impose an order in the derivation
First the rule that generates balanced b's and c's
Then the one that generates eccess b's

$$S \to bSc \mid D \quad D \to bbDc \mid \varepsilon$$

6) **AMBIGUITY IN CONDITIONAL PHRASES**:

example of «historical interest»: the (in)famous ***dangling else*** problem (Pascal, C, …)

$$S \to \overbrace{if\ b\ then\ S\ else\ S \mid if\ b\ then\ S}^{} \mid a$$

$$\overbrace{if\ b\ then \overbrace{if\ b\ then\ a\ else\ a}^{}}^{}$$

$$\overbrace{if\ b\ then\ \overbrace{if\ b\ then\ a}^{}\ else\ a}^{}$$

$$S \to S_E \mid S_T$$
$$S_E \to if\ b\ then\ S_E\ else\ S_E \mid a$$
$$S_T \to if\ b\ then\ S_E\ else\ S_T \mid if\ b\ then\ S$$

REMEDY 1 Rule out this interpretation (i.e., use the ***closest match interpretation***);
two n.t., S_E and S_T:
S_E always has the ***else*** (cannot have only the ***then***),
S_T can have the ***else*** or not have it
$\Rightarrow$ In the new grammar only S_E can precede the ***else***
… not a very nice solution, but one can live with it

$$S \to if\ b\ then\ S\ else\ S\ endif \mid if\ b\ then\ S\ endif \mid a$$

REMEDY 2 Modify the language,
introduce a closing mark ***endif***

GRAMMAR NORMAL FORMS AND TRANSFORMATION

Normal forms constrain the rules without reducing the family of generated languages

They are useful for both proving properties and for language design

Let us see some transformations useful both to

- obtain an equivalent normal form
- design the syntax analyzers

expansion/substitution of a nonterminal (to ELIMINATE it from the rules where it appears)

In the example, we eliminate nonterm. B

Grammar $A \to \alpha B \gamma \quad B \to \beta_1 \mid \beta_2 \mid \ldots \mid \beta_n$

Becomes $A \to \alpha \beta_1 \gamma \mid \alpha \beta_2 \gamma \mid \ldots \mid \alpha \beta_n \gamma$

Derivation $A \Rightarrow \alpha B \gamma \Rightarrow \alpha \beta_i \gamma$

Becomes $A \Rightarrow \alpha \beta_i \gamma$

ELIMINATION OF THE AXIOM FROM RIGHT PARTS :

It is always possible to obtain right part of rules as strings $\in (\Sigma \cup (V \setminus \{S\}))^*$

Simply introduce a new axiom S_0 and the rule $S_0 \rightarrow S$

NULLABLE NONTERMINALS AND ELIMINATION OF EMPTY RULES

a nonterminal is ***nullable*** iff there exists a derivation: $A \overset{+}{\Rightarrow} \varepsilon$

Example – Nullable nonterminals

$$S \to SAB|AC \quad A \to aA|\varepsilon \quad B \to bB|\varepsilon \quad C \to cC|c$$

A and ***B*** are nullable
If the grammar included rule $S \to AB$ then also S would be nullable

NB: If $\varepsilon \in L$ then the axiom is necessarily nullable

NORMAL FORM WITHOUT NULLABLE NONTERMS

that is: no nonterminal other than the axiom S is nullable (S is nullable iff $\varepsilon \in L$)

The textbook (Chapt. 2) presents an algorithm to obtain, for any grammar,
a version without nullable nonterminals

COPY (or CATEGORIZATION) RULES AND THEIR ELIMINATION

A typical example: *iterative_phrase* → *while_phrase* | *for_phrase* | *repeat_phrase*

NB: copy rules factorize common parts ⇒ they reduce the grammar size

That's why they are often present in grammars of technical languages

However copy elimination shortens derivations and reduces the height of syntax trees

A typical ***tradeoff***

The textbook (Chapt. 2) presents algorithm to obtain, for any grammar,
a version without copy rules

Example – Eliminating copy rules from a grammar of arithmetic expressions

$$E \to E + T \mid T \quad T \to T \times C \mid C$$
$$C \to 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9$$

Equivalent grammar without copy rules: remove rules $E \to T$ and $T \to C$, then ...

$E \Rightarrow^+ T$ $\quad$ $E \Rightarrow^+ C$ $\quad$ $T \Rightarrow^+ C$

$$E \to E + T \mid T \times C \mid 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9$$
$$T \to T \times C \mid 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9$$
$$C \to 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9$$

Conversion of Left recursions to Right recursions

Grammars with no left recursion (l-recursion) are necessary for designing ***top down parsers***

Case 1 (simple): Conversion of ***IMMEDIATE*** L-RICURSIONS

$$A \to A\beta_1 \mid A\beta_2 \mid \ldots \mid A\beta_h$$

where no β_i is empty

$$A \to \gamma_1 \mid \gamma_2 \mid \ldots \mid \gamma_k$$

Introduce a new n.t. A'

$$A \to \gamma_1 A' \mid \gamma_2 A' \mid \ldots \mid \gamma_k A' \mid \gamma_1 \mid \gamma_2 \mid \ldots \mid \gamma_k$$

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

in the "new" derivation the prefix γ is generated ***first***, the succeeding parts of type β ***afterwards***

left recursion: string generated from the ***right***; ***right*** recursion: string generated from the ***left***;

derivation in the grammar with l-recursion $\longrightarrow$ $A \Rightarrow A\beta_2 \Rightarrow A\beta_3\beta_2 \Rightarrow \gamma_1\beta_3\beta_2$

deriv. in the gramm. without l-recursion $\longrightarrow$ $A \Rightarrow \gamma_1 A' \Rightarrow \gamma_1\beta_3 A' \Rightarrow \gamma_1\beta_3\beta_2$

Example – Conversion from l-recursion to r-recursion for arithmetic expressions

$$E \to E+T \mid T \qquad T \to T*F \mid F \qquad F \to (E) \mid i$$

E and T are left immediately recursive

$$E \to TE' \mid T \qquad E' \to +TE' \mid +T$$
$$T \to FT' \mid F \qquad T' \to *FT' \mid *F \qquad F \to (E) \mid i$$

Case 2 : non immediate left recursion

it is more complex, not treated here, see textbook, §2.5.13.8

CHOMSKY NORMAL FORM: two types of rules

1. *homogeneous binary rules:* $A \to BC$ *with* $B,C \in V$
2. *Terminal rules with singleton right part* $A \to a$, $a \in \sum$

NB: Syntax trees have internal nodes of degree 2 and leaf parent nodes of degree 1

There exists a procedure to obtain from *G* its Chomsky normal form

Real-time and Greibach normal forms

Real-time normal form: the right part of any rule has a terminal symbol as a prefix

$$A \to a\alpha \quad \text{with} \quad a \in \Sigma, \alpha \in \{\Sigma \bigcup V\}^*$$

Greibach normal form is a special case:

Every right part consists of a terminal followed by zero or more nonterminals

$$A \to a\alpha \quad \text{with} \quad a \in \Sigma, \alpha \in V^*$$

"REAL - TIME" - the term refers to a property of syntax analysis:

Every step reads and consumes one terminal symbol

Number of steps of the analysis = length of the string

FREE GRAMMARS EXTENDED WITH REGULAR EXPRESSIONS (Extended BNF or EBNF or Regular Right Part Grammars-RPPG)

MORE READABLE thanks to the star and cross (iteration) and choice (union) operators

EBNF's allow for the definition of SYNTAX DIAGRAMS which can be viewed as a blueprint of the of the syntax analyzer flowchart

NB: The ***CF*** family is closed under all regular operations, therefore the generative power of EBNF is the same as that of BNF

EBNF GRAMMAR $G = \{ V, \Sigma, P, S \}$
Exactly $|V|$ rules, each in the form $A \rightarrow \eta$, with η r.e. over alphabet $V \cup \Sigma$

Example: Algol-like Language
B: block; **D**: declaration; **I**=Imperative part;
F=phrase; **a**=assignment; **c**=char; **i**=int;
r=real; **v**=variable; **b**=begin; **e**=end

$$B \rightarrow b[D]Ie$$
$$D \rightarrow ((c \mid i \mid r)v(,v)^{*};)^{+}$$
$$I \rightarrow F(;F)^{*}$$
$$F \rightarrow a \mid B$$

Example of a Declaration section

char text1, text2; *real* temp, result;
int alpha, beta2, gamma;

Alternative BNF grammar for ***D*** : $D \rightarrow DE \mid E \quad E \rightarrow AF; \quad A \rightarrow c \mid i \mid r \quad F \rightarrow v, F \mid v$

The BNF grammar for ***D*** is longer and obviously less readable
it conceals the existence of two hierarchically nested lists
Furthermore, the choice of nonterminal symbol names (A, E, F) can be arbitrary

DERIVATIONS AND TREES IN EXTENDED FREE GRAMMARS

Derivation in EBNF G defined by considering an equivalent BNF G' with infinite rules

G includes $A \to (aB)^+$

G' includes $A \to aB \mid aBaB \mid aBaBaB \mid ....$

DERIVATION RELATION FOR EBNF G:

Given strings η_1 and $\eta_2 \in (\Sigma \bigcup V)^*$

η_2 is said to be *derived* immediately in G from η_1

$\eta_1 \Rightarrow \eta_2$, if the two strings can be factorized as:

$\eta_1 = \alpha A \gamma$, $\eta_2 = \alpha \vartheta \gamma$ and there exists a rule

$A \to e$ such that the r.e. e admits the derivation $e \overset{*}{\Rightarrow} \vartheta$

Notice that η_1 and η_2 do not contain

r.e. operators nor parenthesis.

Only string e is a r.e. but it does not appear in the derivation if it is not terminal

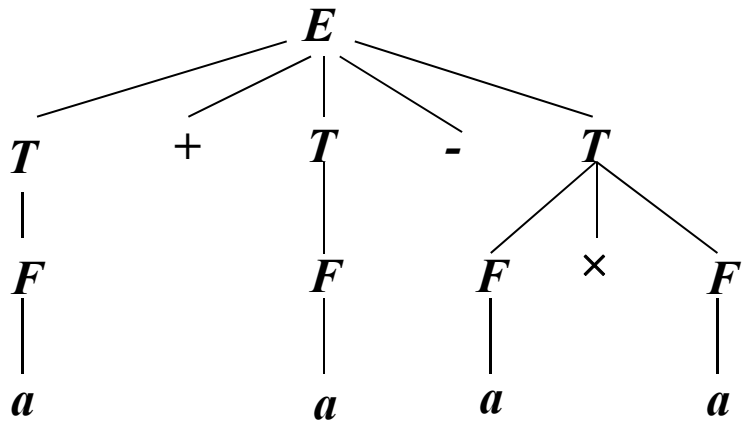
Example: Extended derivation for arithmetic expressions
Extended grammar for arithmetic expressions with four infix operators, parentheses and variable *a*.

$E \to [\,+\,|\,-\,]\; T\,(\,(\,+\,|\,-\,)\; T\,)^*$ $\qquad T \to F\,(\,(\,\times\,|\,/\,)\; F\,)^*$ $\qquad F \to\; a \;\mid\; \text{‘(’}\; E\; \text{‘)’}$

left derivation:

$$E \Rightarrow T+T-T \Rightarrow F+T-T \Rightarrow a+T-T \Rightarrow a+F-T \Rightarrow$$

$$\Rightarrow a+a-T \Rightarrow a+a-F\times F \Rightarrow a+a-a\times F \Rightarrow a+a-a\times a$$



unbounded node degree

the tree is in general **wider** …

… and **reduced in depth**

AMBIGUITY IN EXTENDED FREE GRAMMARS

An ambiguous BNF remains so when viewed as an EBNF grammar

An EBNF grammar is ambiguous if it has a sentence with several syntax trees

An ***additional cause*** of ambiguity for EBNF grammars:
when the r.e. in the right part of a rule is ambiguous

Example: grammar $S \rightarrow a^*\ b\ S \mid\ a\ b^*\ S\ \mid\ c$

is ambiguous because so is the r.e. $a^*\ b\ S \mid\ a\ b^*\ S\ \mid\ c$

in fact, from numbered version $a_1{}^*\ b_2\ S_3 \mid\ a_4\ b_5{}^*\ S_6\ \mid\ c_7$

one can derive $a_1\ b_2\ S_3$ and $a_4\ b_5\ S_6$

GRAMMARS OF REGULAR LANGUAGES

Regular languages are a special case of free languages

They are generated by grammars with suitable (rather strong) constraints on the rule form

FROM REGULAR EXPRESSIONS TO CONTEXT-FREE GRAMMARS

Define the form of the rules depending on that of the regular expression

Notice that (left- or right-) recursive rules match iterative operators (star '*' and cross '+')

REGULAR EXPRESSION	**FORM OF RULE $R \rightarrow E$**
1. $r = r_1.r_2 \ldots . r_k$	1. $R \rightarrow E_1 E_2 \ldots E_k$
2. $r = r_1 \mid r_2 \mid \ldots . \mid r_k$	2. $R \rightarrow E_1 \mid E_2 \mid \ldots \mid E_k$
3. $r = (r_1)^*$	3. $R \rightarrow RE_1 \mid \varepsilon$ or $R \rightarrow E_1 R \mid \varepsilon$
4. $r = (r_1)^+$	4. $R \rightarrow RE_1 \mid E_1$ or $R \rightarrow E_1 R \mid E_1$
5. $r = b \in \Sigma$	5. $R \rightarrow b$
6. $r = \varepsilon$	6. $R \rightarrow \varepsilon$

Example

$$E = (abc)^* \mid (ff)^+$$

$E = E_1 \mid E_2$	hence	$E \to E_1 \mid E_2$
$E_1 = (E_3)^*$	hence	$E_1 \to E_1\, E_3 \mid \varepsilon$
$E_3 = abc$	hence	$E_3 \to abc$
$E_2 = (E_4)^+$	hence	$E_2 \to E_2\, E_4 \mid E_4$
$E_4 = ff$	hence	$E_4 \to ff$

We can therefore conclude that every regular language is free

But there are free languages that are not regular (e.g. palindromes)

REG $\subset$ CF (REG $\subseteq$ CF and REG $\neq$ CF)

In a coming lesson we'll introduce ***unilinear grammars***, subclass of free grammars equivalent to regular expressions (every unilinear grammar generates a regular language)