Free Grammars - II

Prof. Licia Sbattella

aa 2007-08

Translated and adapted by L. Breveglieri

SYNTAX TREE AND CANONICAL DERIVATION

SYNTAX TREE: directed acyclic graph, such that for every pair of nodes there exists one and only one path (non necessarily directed) connecting them.

THE SYNTAX TREE:

- represents graphically the derivation process
- gives a parent-child relation or a root-node-leaf relation
- the sequence of leaves, scanned from left to right, is the so-called frontier
- the degree of a node (so-called node arity) is the length of the production rule

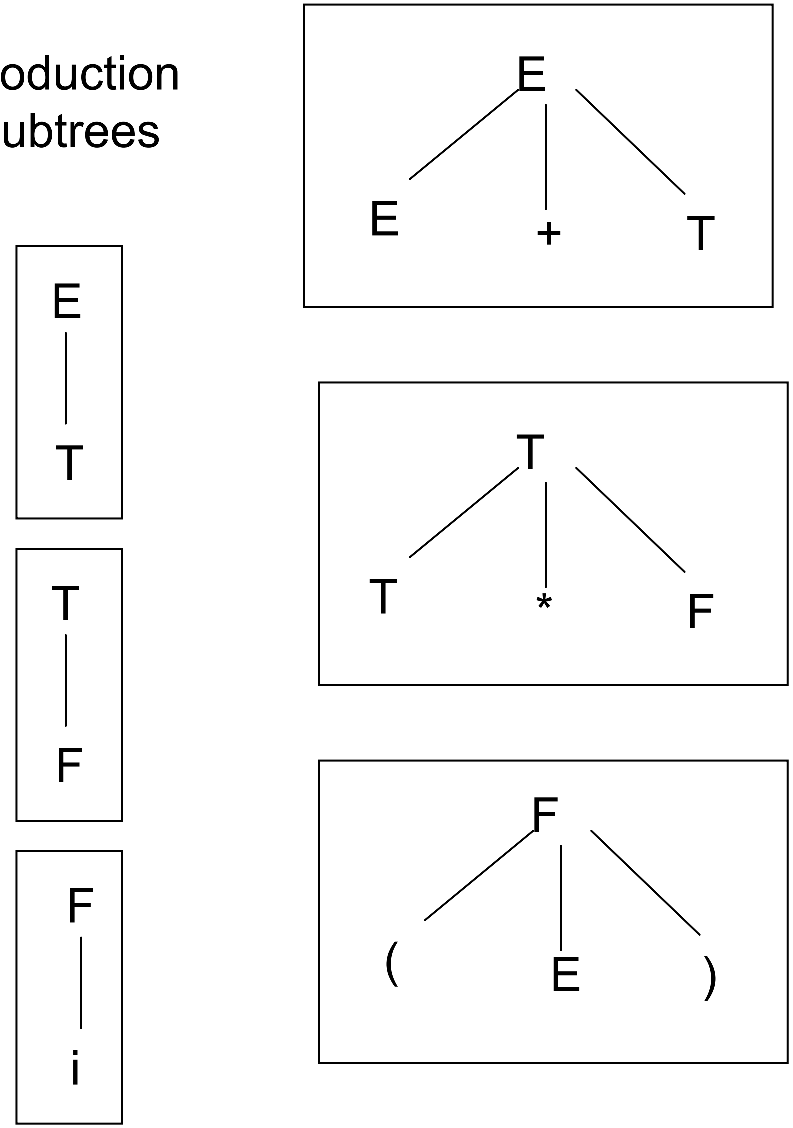
SUBTREE with root N: the tree that has root N and includes all the descendants of N (the immediate siblings of N, the siblings of the siblings, and so on) .

SYNTAX TREE: the root is the axiom and the frontier is the generated phrase.

grammar

1. $E \rightarrow E + T$
2. $E \rightarrow T$
3. $T \rightarrow T * F$
4. $T \rightarrow F$
5. $F \rightarrow (E)$
6. $F \rightarrow i$

production subtrees

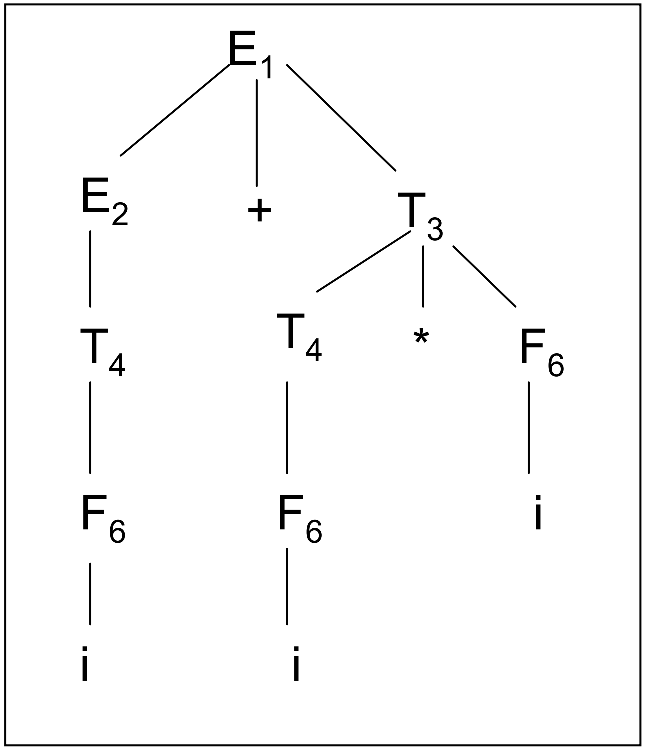


parenthesized representation of the syntax tree

$$\boxed{[[[[i]_F\,]_T\,]_E + [[[i]_F\,]_T * [i]_F\,]_T\,]_E}$$

LEFTMOST DERIVATION

$$E \underset{1}{\Rightarrow} E+T \underset{2}{\Rightarrow} T+T \underset{4}{\Rightarrow} F+T \underset{6}{\Rightarrow} i+T \underset{3}{\Rightarrow} i+T*F \underset{4}{\Rightarrow}$$
$$\underset{4}{\Rightarrow} i+F*F \underset{6}{\Rightarrow} i+i*F \underset{6}{\Rightarrow} i+i*i$$



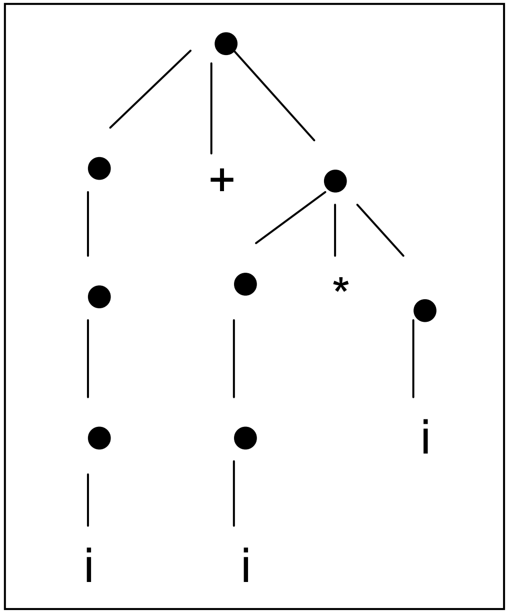
syntax tree

RIGHTMOST DERIVATION

$$E \underset{1}{\Rightarrow} E+T \underset{3}{\Rightarrow} E+T*F \underset{6}{\Rightarrow} E+T*i \underset{4}{\Rightarrow} E+F*i \underset{6}{\Rightarrow} E+i*i \underset{2}{\Rightarrow}$$
$$\underset{2}{\Rightarrow} T+i*i \underset{4}{\Rightarrow} F+i*i \underset{6}{\Rightarrow} i+i*i$$

SKELETON TREE (only the frontier and the connections).

skeleton syntax tree

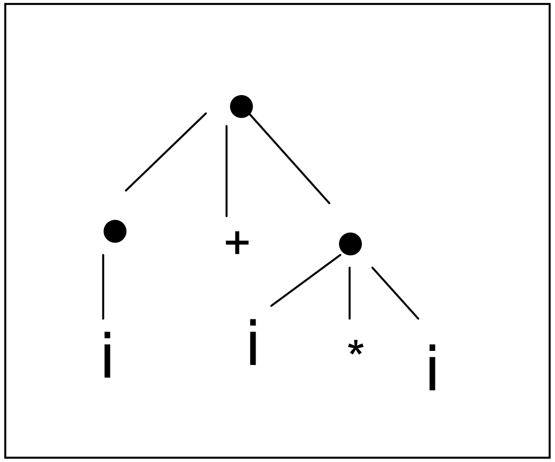


parenthesized representation of the syntax tree

$$\boxed{[[[[i]]]+[[[i]]*[i]]]}$$

CONDENSED SKELETON TREE (merge into one all the internal nodes aligned on a linear path, that is a path without branching points).

condensed tree



$$\boxed{[[i]+[[i]*[i]]]}$$

condensed representation of the syntax tree

LEFTMOST AND RIGHTMOST DERIVATION / FREE GRAMMAR

$$\boxed{E \overset{+}{\underset{s}{\Rightarrow}} i+i*i} \qquad \boxed{E \overset{+}{\underset{d}{\Rightarrow}} i+i*i}$$

$$\boxed{E \overset{+}{\underset{s,d}{\Rightarrow}} i+i*i} \quad \boxed{\begin{array}{l} E \underset{s,d}{\Rightarrow} E+T \underset{d}{\Rightarrow} E+T*F \underset{s}{\Rightarrow} T+T*F \Rightarrow T+F*F \underset{d}{\Rightarrow} T+F*i \underset{s}{\Rightarrow} \\ \qquad \underset{s}{\Rightarrow} F+F*i \underset{d}{\Rightarrow} F+i*i \underset{d}{\Rightarrow} i+i*i \end{array}}$$

EVERY PHRASE OF A FREE GRAMMAR CAN BE GENERATED BY MEANS OF A LEFTMOST DERIVATION (and by a RIGHTMOST DERIVATION as well).
This property is of great importance for the syntactic analysis algorithms.
It allows to organize in the most appropriate way the derivation of the phrase.

PARENTHESES LANGUAGE: artificial languages frequently contain nested structures (that is, parentheses), where an element pair starts and ends some substructure (like for instance a substring), and where the pair may in turn contain a nested pair, and so on recursively down to any depth.

Pascal:	begin ... end
C:	{ … }
XML:	< title > … < /title >
LaTeX:	\begin{equation} … \end{equation}

Do not consider the specific way the marker symbols are encoded.

The paradigm of parentheses languages is known as the *Dyck Language*.

Alphabet: $\Sigma = \{ ')', '(', ']', '[' \}$ Phrases: $()[[()[]]()]$

Parentheses phrases can be equivalently defined by means of *cancellation rules*: repeatedly remove from the string any factor that consists of a pair of adjacent open and close parentheses, as far as possible. The original string is valid if and only if the final result is the *empty* string.

$$[\ ] \Rightarrow \varepsilon \qquad (\ ) \Rightarrow \varepsilon$$

DYCK LANGUAGE: open parentheses *a, b, ...,* closed parentheses *a’, b’, ...*

$$\Sigma = \{a, a', b, b'\}$$

$$S \rightarrow aSa'S \mid bSb'S \mid \varepsilon$$

$$\underbrace{a\;\underbrace{a\;\underbrace{a\,a'}\;a'}\;\underbrace{a\;\underbrace{a\;\underbrace{a\,a'}\;a'}\;a'}\;a'}$$

LINEAR BUT NON-REGULAR LANGUAGE:

$$L_1 = \{a^n c^n \mid n \geq 1\} = \{ac, aacc, ...\}$$

$$S \rightarrow aSc \mid ac$$

L_1 is a subset of the Dyck language:
it does not admit more than one parentheses nest.

REGULAR COMPOSITION OF FREE LANGUAGES

The basic regular operations (union, concatenation and star), applied to free languages, still yield free languages.

The family of free languaegs is closed with respect to the language operations *union*, *concatenation* and *star*.

$$\boxed{\begin{array}{l} G_1 = \left(\Sigma_1, V_{N_1}, P_1, S_1\right) \ \text{ e } \ G_2 = \left(\Sigma_2, V_{N_2}, P_2, S_2\right) \\ V_{N_1} \bigcap V_{N_2} = \varnothing \qquad S \notin (V_{N_1} \bigcup V_{N_2}) \end{array}}$$

UNION: $\boxed{G = (\Sigma_1 \bigcup \Sigma_2, \{S\} \bigcup V_{N_1} \bigcup V_{N_2}, \{S \to S_1 \mid S_2\} \bigcup P_1 \bigcup P_2, S)}$

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

STAR: G of $(L_1)^*$ is obtained by adding to G_1 the rules $S \to S\ S_1 \mid \varepsilon$

(CROSS: G of $(L1)^+$ is obtained by adding to G_1 the rules $S \to S\ S_1 \mid S_1$)

EXAMPLE: union of languages

$$\boxed{L=\left\{a^i b^i c^* \mid i\ge 0\right\}\bigcup\left\{a^* b^i c^i \mid i\ge 0\right\}=L_1\bigcup L_2}$$

$$\boxed{\begin{array}{l} G_1 \\ S_1 \to XC \\ X \to aXb \mid \varepsilon \\ C \to cC \mid \varepsilon \end{array}}$$

$$\boxed{\begin{array}{l} G_2 \\ S_2 \to AY \\ Y \to bYc \mid \varepsilon \\ A \to aA \mid \varepsilon \end{array}}$$

$$\boxed{\left\{S \to S_1 \mid S^{"}\right\}\bigcup P_1 \bigcup P^{"}}$$

CAUTION: if the hypothesis that the two non-terminal alphabets are disjoint does not hold, then the construction above yields a grammar that generates a superset of the real union language. For example, replacing G_2 by G’’ would allow to generate the invalid phrase shown aside.

$$\boxed{\begin{array}{l} G^{"} \\ S^{"} \to AX \\ X \to bXc \mid \varepsilon \\ A \to aA \mid \varepsilon \end{array}}$$

$$\boxed{abcbc}$$

The family LIB of free languages is closed with respect to STRING MIRRORING.

Given the grammar G of the language, the grammar G_R that generates the mirror image of the language is obtained from G by mirroring the right member of every production rule of G.

REG and LIB are both closed with respect to union, concatenation and star

but later it will be proved that

they do not behave in the same way as for complement and intersection

AMBIGUITY: semantic versus syntactic

"Ho visto un uomo in giardino con il cannocchiale" (is the man who has the tool ?)

"La pesca è bella" (fruit or sport ?)

"half baked chicken" (half baked or half chicken ?)

Natural languages are largely ambiguous and this phenomenon is unavoidable, as thay aim at describing everything, but with only a finite dictionary of nouns. Instead, in the artifical languages ambiguity is an undesirable phenomenon (think of a program, how could we tolerate ambiguity ?).
Therefore ambiguity must be eliminated (if possible) or kept under control.

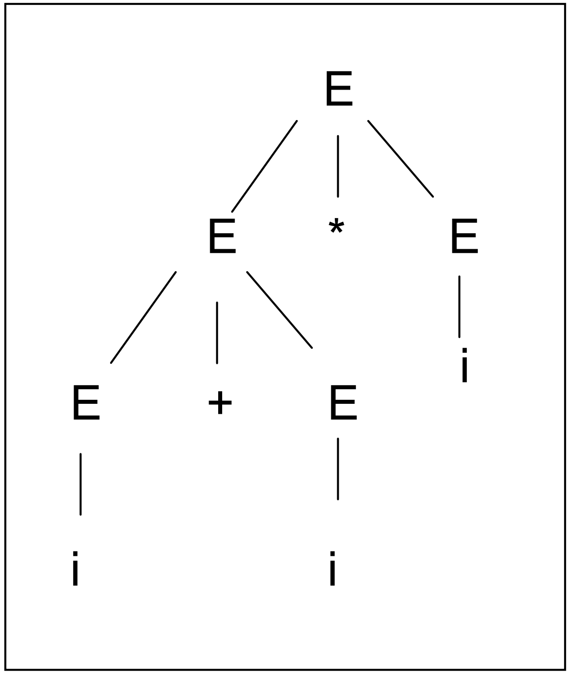
Consider SYNTACTIC AMBIGUITY. A phrase *x* of the language defined by the grammar G is said to ambiguous if it admits two or more different syntactic trees (or, equivalently, two different derivations). A grammar G is said to be ambiguous if it generates (at least) an ambiguous string.

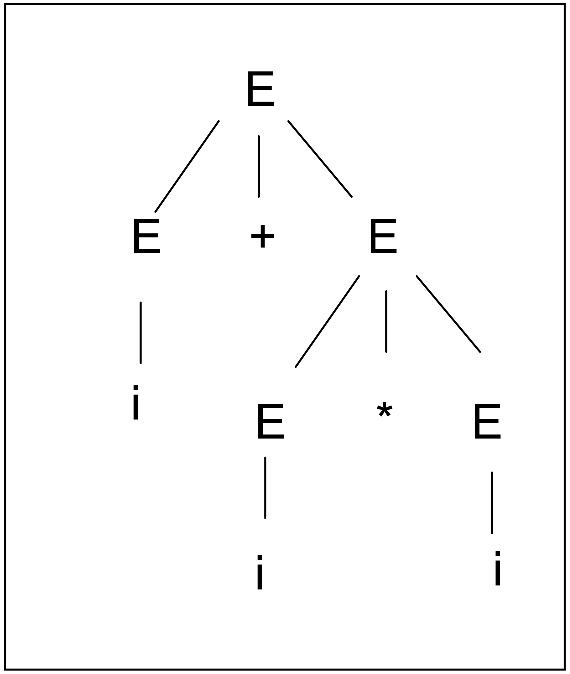
EXAMPLE – grammar G' of arithmetic expressions:

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

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

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





The phrase *i* + *i* * *i* is ambiguous. Therefore the grammar G’ is itself ambiguous.

In fact, G’ is not compliant with the standard convention that multiplication must precede addition.

Previously, a unambiguous grammar G that generates the arithmetic expressions has been shown. G’ is equivalent to G (that is, L(G) = L(G’)), but is smaller than G. However, G’ is ambiguous. This is often the case: simplifying the grammar frequently ends up with having an ambiguous behaviour, in general.

The AMBIGUITY DEGREE of a phrase *x* in the language L(G) is the number of different syntax trees that *x* admits. The ambiguity degree of a string may be infinite (only if there are nullable non-terminal symbols).
The AMBIGUITY DEGREE of a grammar G is the maximum ambiguity degree of the strings generated by G. While every string of L(G) may have a finite ambiguity degree, the ambiguity degree of G may still turn out to be infinite.

RELEVANT PROBLEM: decide whether a grammar G is ambiguous or not.
UNFORTUNATELY, THIS PROBLEM IS UNDECIDABLE: there does not exist any algorithm to decide whether a given grammar G is ambiguous or not.
This can be decided for some grammars, but in general not for every grammar.

By using the methods of theoretical information science, it is possible to prove that any potential decision algorithm should examine increasingly long derivations, which is unfeasible and hence leads to undecidability. However, the inexistence of a general algorithm does not preclude to decide for a specific grammar, by resorting to *ad hoc* methods depending on the structure of the grammar. From a practical point of view, examining all the derivations of G up to a given length is often sufficient to get convinced that the grammar is unambiguous, though it is not a rigorous proof.

AS AMBIGUITY MAY BE DIFFICULT TO IDENTIFY A POSTERIORI
IT IS ADVISABLE TO AVOID IT BY CONSTRUCTION (A FORTIORI)

EXAMPLE: resume the previous example

The phrase *i* + *i* + *i* has ambiguity degree equal to 2.

The phrase *i* + *i* * i + *i* has ambiguity degree equal to 5 (see below).

$$\underbrace{\underbrace{i+i}*\underbrace{i+i}},\quad \underbrace{\underbrace{i+\underbrace{i*i}}+i},\quad \underbrace{i+\underbrace{\underbrace{i*i}+i}},\quad \underbrace{\mathrm{i}+\underbrace{\mathrm{i}*\underbrace{\mathrm{i}+\mathrm{i}}}},\quad \underbrace{\underbrace{\underbrace{i+i}*i}+i}$$

Typically, as the derived phrase gets longer, its ambiguity degree grows as well.

DIRECTORY OF AMBIGUOUS MODELS AND HOW TO REMOVE AMBIGUITY

AMBIGUITY OF TWO-SIDED RECURSION: a non-terminal symbol that is recursive both on the left and on the right (two-sided recursion) always allows two or more derivations, and thus gives rise to an ambiguous behaviour (see below).

$$\boxed{A \overset{+}{\Rightarrow} A\ldots \quad A \overset{+}{\Rightarrow} \ldots A}$$

EXAMPLE 1: grammar G_1 generates the phrase *i + i + i* in two different ways (by means of two different leftmost derivations).

ambiguous grammar

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

Notice that L is regular:

$$L(G_1) = i(+i)^*$$

Non-ambiguous version:

$$E \to i + E \mid i$$

$$E \to E + i \mid i$$

EXAMPLE 2: ambiguity that originates form left and right recursion, separately

ambiguous grammar

$$\boxed{G_2 : \quad A \to aA \mid Ab \mid c}$$

ambiguity can be removed by generating separately the two lists

$$L(G_2) = a^* c b^*$$
$$S \to AcB$$
$$A \to aA \mid \varepsilon$$
$$B \to bB \mid \varepsilon$$

or one may choose to generate orderly the two lists (first *a* then *b*, or viceversa)

$$L(G_2) = a^* c b^*$$
$$S \to aS \mid X$$
$$X \to Xb \mid c$$

EXAMPLE 3: the grammar that generates Polish postfix expressions containing addition and multiplication, has left recursion (but not right) and is not ambiguous.

$$\boxed{S \to +SS \mid \times SS \mid i}$$

AMBIGUITY OF UNION (maybe the simplest to understand)

If two languages $L_1(G_1)$ and $L_2(G_2)$ share some phrases (that is, their intersection is not empty), the grammar G of the union language, constructed as shown before, is surely ambiguous.

CAUTION: one need assume that the two non-terminal sets are disjoint, otherwise, as seen before, the union grammar would generate
a superlanguage containing strictly both languages.

A phrase *x* belonging to the union of L_1 and L_2, and admitting two different derivations, the first only using rules of G_1 and the second only using rules of G_2, is ambiguous for the grammar G, as G contains both sets of rules. Only the phrases belonging to $L_1 \setminus L_2$ or to $L_2 \setminus L_1$, if any, would be generated in a non-ambiguous way.

EXAMPLE 1: ambiguity of union may arise when in a programming language a special case, out of a general one, is treated by means of separate rules.

$$E \to E + 1 \qquad E \to inc\ E$$

separate rules

$$E \to E + T \mid T \quad T \to V \mid C \quad V \to \ldots$$
$$C \to 0 \mid 1B \mid \ldots \mid 9B \quad B \to 0B \mid 1B \mid \ldots \mid 9B \mid \varepsilon$$

basic grammar

EXAMPLE 2: ambiguity of union may arise when in a programming language the same operator has two different meanings. For example, in Pascal the operator “+” could indicate both integer addition and set-theoretic union.

ambiguous grammar

$$E \to E + T \mid T \quad T \to V \quad V \to \ldots$$
$$E_{set} \to E_{set} + T_{set} \mid T_{set} \quad T_{set} \to V$$

If one keeps on overloading the “+” operator but also wants to remove ambiguity, one must give up with pretending to have separate production rules to generate arithmetic or set-theoretic expressions. Otherwise, one may split the operator “+” and use the symbol “+” only to denote integer addition, while a new operator, say for example “U”, is introduced to denote set-theoretic union. The latter solution, however, changes the language.

EXAMPLE 3:

$$\boxed{S \rightarrow bS \mid cS \mid \varepsilon}$$

non-ambiguous version

ambiguous grammar

$$\boxed{\begin{array}{l}
G: \quad S \rightarrow bS \mid cS \mid D \quad D \rightarrow bD \mid cD \mid \varepsilon \\
L(G) = L_S(G) \bigcup L_D(G) \\
L_S(G) = \{b, c\}^* = L_D(G) \\
S \overset{+}{\Rightarrow} bbcD \Rightarrow bbc \quad S \Rightarrow D \overset{+}{\Rightarrow} bbcD \Rightarrow bbc
\end{array}}$$

EXAMPLE 4:

$$\boxed{\begin{array}{l}
S \rightarrow B \mid D \mid \varepsilon \\
B \rightarrow bBc \mid bc \\
D \rightarrow dDe \mid de
\end{array}}$$

non-ambiguous version

ambiguous grammar

$$S \rightarrow B \mid D \quad B \rightarrow bBc \mid \varepsilon$$

$$D \rightarrow dDe \mid \varepsilon$$

B generates $b^n c^n, n \geq 0$

D generates $d^n e^n, n \geq 0$

ε is the only ambiguous phrase

LANGUAGES THAT ARE INHERENTLY AMBIGUOUS

A language is said to be INHERENTLY AMBIGUOUS if any grammar that generates it is necessarily ambiguous (that is, if all the equivalent grammars of the language happen to be ambiguous).

EXAMPLE: here is a classical inherently ambiguous language

$$\boxed{L = \left\{ a^i b^j c^k \mid (i, j, k \geq 0) \wedge ((i = j) \vee (j = k)) \right\}}$$

$$\boxed{L = \left\{ a^i b^i c^* \mid i \geq 0 \right\} \bigcup \left\{ a^* b^i c^i \mid i \geq 0 \right\} = L_1 \bigcup L_2}$$

ambiguous
union

Whatever grammar G is designed to generate L, the phrases *ε*, *abc*, $a^2b^2c^2$, ... ALWAYS HAPPEN TO BE DERIVABLE BY MEANS OF TWO OR MORE DERIVATIONS. This behaviour is caused by the very nature of the language L and does not depend on the specific grammar G that generates L.

In fact, such phrases may be generated by G_1 to check that $|x|_a = |x|_b$

$$\boxed{\underbrace{a\ldots\underbrace{a\,\underbrace{ab}\,b}\ldots b}\,bcc\ldots c}$$

generated by G_1

or may be generated by G_2 to check that $|x|_b=|x|_c$

$$\boxed{a\ldots aa\,\underbrace{b\ldots\underbrace{b\,\underbrace{bc}\,c}\ldots c}}$$

generated by G_2

But clearly since both sub-grammars can generate such phrases, ambiguity arises. In whatever way the two sub-grammars are modified, both checks are unavoidable, as they belong to the structure of the language, and therefore ambiguity is unavoidable as well.

AMBIGUITY OF CONCATENATION

Concatenating two languages may give rise to ambiguity if there exists a suffix of a phrase of the former language that is a prefix of a phrase of the latter language.

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

Assume that G$_1$ and G$_2$ are not ambiguous, then G is ambiguous if there exist two phrases x' $\in$ L$_1$ and x'' $\in$ L$_2$, and a non-empty string *v* such that:

$$x' = u'v \wedge u' \in L_1 \quad x'' = vz'' \wedge z'' \in L_2$$

$$u'vz'' \in L_1.L_2 \quad \text{and is ambiguous}$$

$$S \Rightarrow S_1 S_2 \overset{+}{\Rightarrow} u' S_2 \overset{+}{\Rightarrow} u'vz''$$

$$S \Rightarrow S_1 S_2 \overset{+}{\Rightarrow} u'v S_2 \overset{+}{\Rightarrow} u'vz''$$

EXAMPLE 1 – concatenation of Dyck languages

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

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

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

$$S_1 \to a S_1 a' S_1 \mid b S_1 b' S_1 \mid \varepsilon$$

$$S_2 \to b S_2 b' S_2 \mid c S_2 c' S_2 \mid \varepsilon$$

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

In order to eliminate ambiguity, it is necessary to exclude the case when a suffix moves from the former language to the latter one. A solution consists of inserting a separator between the two languages. Such a separator must not belong to the alphabets of the two languages. The concatenation language $L_1 \# L_2$ is generated by the additional axiomatic rule $S \to S_1 \# S_2$.

EXAMPLE 2 – encoding – the relationship between ambiguity and the uniqueness of encoding in information theory

A message is a sequence of symbols out of the set Γ = { A, B,…, Z }, with possible repetitions. Such symbols are then encoded into strings of letters out of the set of terminal symbols Σ. Most frequently Σ is the binary alphabet, Σ = { 0, 1 }.

$$\Gamma = \left\{ \overbrace{A}^{01}, \overbrace{C}^{10}, \overbrace{E}^{11}, \overbrace{R}^{001}, \right\}$$

$$ARRECA : 01\ 001\ 001\ 11\ 10\ 01$$

$$Testo \to ATesto \mid CTesto \mid ETesto \mid RTesto \mid A \mid C \mid E \mid R$$

$$A \to 01 \quad C \to 10 \quad \mathrm{E} \to 11 \quad \mathrm{R} \to 001$$

unambiguous grammar

This grammar is not ambiguous: every message encoded onto Σ admits only one syntax tree, and therefore can be decoded in a unique way.

$$\overbrace{\overbrace{01}^{A}\,\overbrace{\overbrace{10}^{C}}^{Testo}}^{Testo}$$

A bad choice of the set of encoding strings causes the grammar to be ambiguous:

ambiguous grammar

$$\Gamma = \left\{ \overbrace{A}^{00}, \overbrace{C}^{01}, \overbrace{E}^{10}, \overbrace{R}^{010}, \right\}$$

$$Testo \to ATesto \mid CTesto \mid ETesto \mid RTesto \mid A \mid C \mid E \mid R$$

$$A \to 00 \quad C \to 01 \quad \mathrm{E} \to 10 \quad \mathrm{R} \to 010$$

$$ARRECA = 00\ 010\ 010\ 10\ 01\ 00$$

$$ACAEECA = 00\ 01\ 00\ 10\ 10\ 01\ 00$$

The problem originates from two aspects:

01 00 10 = 010 010, and

01 is a prefix of 010

The *theory of codes* studies these and also other aspects, to identify conditions ensuring that a code (= a set of encoding strings) is decodable in a unique way.

OTHER AMBIGUOUS SITUATIONS – regexps may be themselves ambiguous

ambiguous grammar

EXAMPLE 1

Every phase containing two or more *c* is ambiguous. To remove ambiguity, one may impose that the mandatory *c* is the leftmost one.

$$\boxed{\begin{array}{l} S \to DcD \quad D \to bD \mid cD \mid \varepsilon \\ \{b,c\}^{*} c \{b,c\}^{*} \end{array}}$$

unambiguous version

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

EXAMPLE 2 – Fixing the order of application of the rules – The rule to generate two *b* can be applied before or after the rule that generates one *b*.

$$\boxed{\begin{array}{l} S \to bSc \mid bbSc \mid \varepsilon \\ S \Rightarrow bbSc \Rightarrow bbbScc \Rightarrow bbbcc \\ S \Rightarrow bSc \Rightarrow bbbScc \Rightarrow bbbcc \end{array}}$$

ambiguous grammar

$$\boxed{S \to bSc \mid D \quad D \to bbDc \mid \varepsilon}$$

unambiguous version

AMBIGUITY IN CONDITIONAL PHRASES

$$\boxed{\begin{array}{l} S \to if\ \ b\ then\ S\ else\ S \mid if\ \ b\ then\ S \mid a \\[2ex] \overbrace{if\ \ b\ then\ \overbrace{if\ \ b\ then\ a\ else\ a}} \\[2ex] \overbrace{if\ \ b\ then\ \ \overbrace{if\ \ b\ then\ a}\ \ else\ a} \end{array}}$$

so-called problem
of the
“dangling else”

$$\boxed{\begin{array}{l} S \to S_E \mid S_T \quad S_E \to if\ \ b\ then\ S_E\ \ else\ S_E \mid a \\ S_T \to if\ \ b\ then\ S_E\ \ else\ S_T \mid if\ \ b\ then\ S \end{array}}$$

Only S_E can precede *else*

unambiguous version 1

$$\boxed{\begin{array}{c} S \to if\ \ b\ then\ S\ else\ S\ end\_if \mid \\ \mid if\ \ b\ then\ S\ end\_if \mid a \end{array}}$$

unambiguous version 2

Use the additional keyword *end_if* to mark the end of the conditional phrase

Bibliography

- S. Crespi Reghizzi, *Linguaggi Formali e Compilazione*, Pitagora Editrice Bologna, 2006
- Hopcroft, Ullman, *Formal Languages and their Relation to Automata*, Addison Wesley, 1969
- A. Salomaa – *Formal Languages*, Academic Press, 1973
- D. Mandrioli, C. Ghezzi – *Theoretical Foundations of Computer Science*, John Wiley & Sons, 1987
- L. Breveglieri, S. Crespi Reghizzi, *Linguaggi Formali e Compilatori: Temi d’Esame Risolti,* web site (eng + ita)