Free Grammars - III

Prof. Licia Sbattella

aa 2007-08

Translated and adapted by L. Breveglieri

WEAK AND STRONG (or STRUCTURAL) EQUIVALENCE

WEAK EQUIVALENCE: two grammars G and G' are weakly equivalent if and only if they generate the same language, that is if $L(G) = L(G')$.

Such grammars might give completely different structures to the same phrase, one of which structures might be inadequate as for the desired semantic interpretation.

STRONG or STRUCTURAL EQUIVALENCE: two grammars G and G' are strongly equivalent if and only if they generate the same language, that is if $L(G) = L(G')$, and moreover if G and G' assign to every phrase syntactic trees that can be considered structurally equivalent (this means that for every phrase the condensed skeleton trees should be identical).

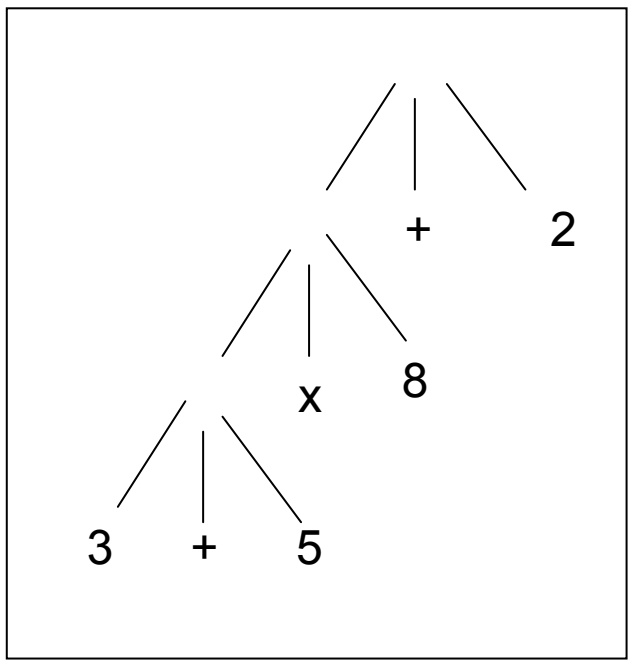
Strong equivalence implies weak equivalence, but the opposite implication does not hold.

Strong equivalence is a *decidable* property.

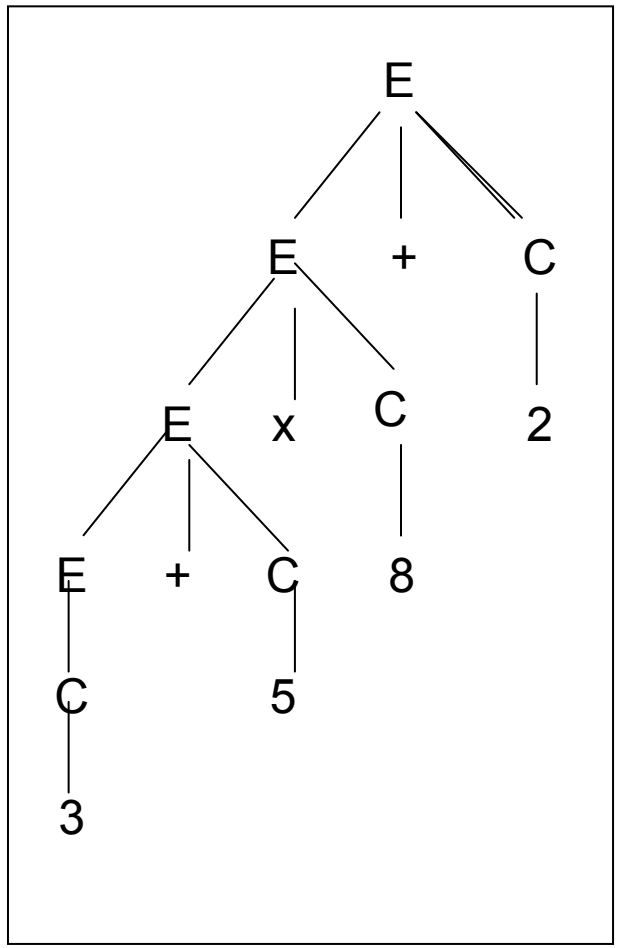
Weak equivalence is a *undecidable* property (but in some special cases).

EXAMPLE: structural equivalence of arithmetic expressions $-3+5\times 8+2$

$$G_1:\ E \to E+C\ \ E \to E\times C\ \ E \to C$$
$$C \to 0|1|2|3|4|5|6|7|8|9$$



skeleton tree



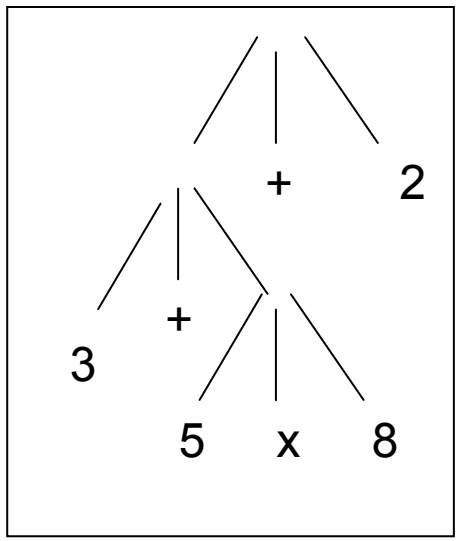
full tree

$$G_2: \quad E \to E+T \quad E \to T \quad T \to T \times C \quad T \to C$$
$$C \to 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9$$

$G_1$and G_2 are not equivalent in the structural sense.

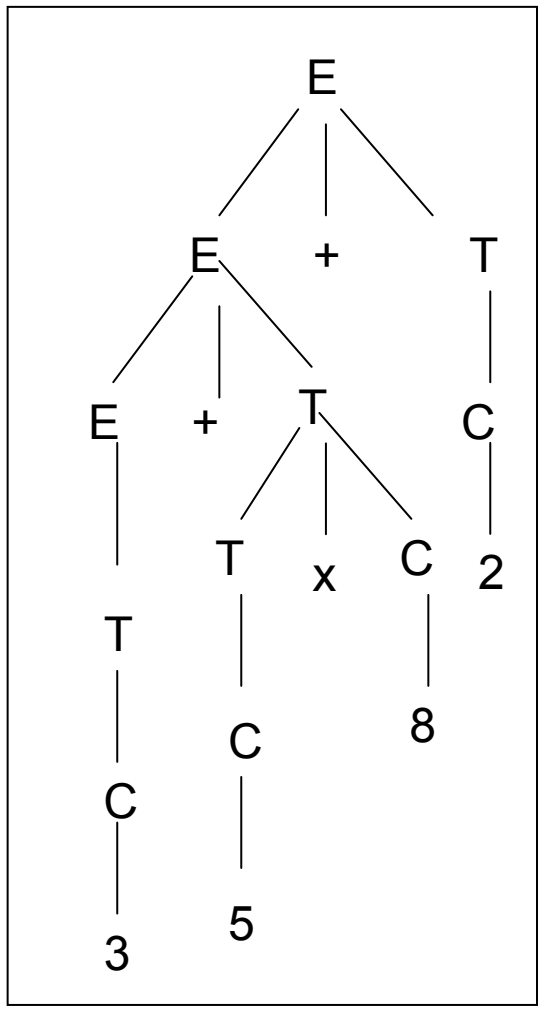
semantic interpretations:
G_1: (((3 + 5) × 8) + 2)
G_2: ((3 + (5 × 8)) + 2)

Only G_2 is structurally adequate with reference to the usual precedence rules between operators (but G_2 is more complex).



skeleton tree

full tree



When defining formally a language, STRUCTURAL ADEQUACY is an issue to be considered carefully, as usually the grammar generating the language is the syntactic basis to define the semantic interpretation or to design a syntax-driven transduction.

G_3 is structurally equivalent to the previous grammar:

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

G_3 contains more production rules than the previous grammar, as it is not designed by sharing or grouping substructures according to taxonomy. Cathegories and taxonomy may be helpful to reduce grammar complexity.

STRUCTURAL EQUIVALENCE IN THE LARGE SENSE

$$\{S \to Sa \mid a\} \quad \{X \to aX \mid a\}$$

$$L = a^{+}$$

$$\underbrace{\underbrace{a}\, a}_{} \qquad \underbrace{a\, \underbrace{a}}_{}$$

In the strict sense, these two grammars are not structurally equivalent, since the syntactic trees of the sample string *aa* below are not identical.

However, one can observe that every left linear tree of the first grammar corresponds to a linear right tree of the second grammar (that is, for the same string the two grammars generate two trees that are mirror images of each other).

The two grammars are strongly equivalent *in the large sense*, that is by abstracting from some minor structural detail.

GRAMMAR NORMAL FORM

Normal forms impose restrictions to the production rules, but without reducing the family of generated languages. Such forms are useful for instance in proving theorems, rather than in designing a grammar for a specific language.

There are some transformations to put a general grammar into an equivalent normal form. Such transformations are helpful to design syntactic analyzers.

GRAMMAR to start from:

$$G = \{\Sigma, V, P, S\}$$

NON-TERMINAL EXPANSION

Expanding a non-terminal does not change the generative power of the grammar.

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

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

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

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

REMOVING THE AXIOM FROM THE RIGHT MEMBER OF THE PRODUCTION RULES

It is always possible to restrict the right member of a production rules to being a string over the alphabet ($\Sigma \cup (V - \{ S \})$), that is to not containing the axiom S.

It suffices to introduce the new axiom S_0 and the new production rule $S_0 \rightarrow S$.

NULLABLE NON-TERMINAL SYMBOL and NON-NULLABLE NORMAL FORM

A non-terminal A is said to be *nullable* if and only if there exists a derivation as aside:

$$A \overset{+}{\Rightarrow} \varepsilon$$

CAUTION: this does not exclude that A can generate also a string different from ε.

$Null \subseteq V$ is the set of the nullable non-terminal symbols

Computation of the set of the nullable non-terminal symbols:

$$A \in Null \text{ if } A \rightarrow \varepsilon \in P$$

$$A \in Null \text{ if } (A \rightarrow A_1 A_2 \ldots A_n \in P \text{ with } A_i \in V \setminus \{A\}) \wedge \forall A_i (A_i \in Null)$$

EXAMPLE – how to identify the nullable non-terminal symbols

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

$$Null = \{A, B\}$$

NON-NULLABLE NORMAL FORM (and not containing empty rules): every non-terminal symbol different from the axiom is not nullable, and the axiom itself is nullable if and only if the language contains the empty string ε.

HOW TO OBTAIN THE NON-NULLABLE NORMAL FORM OF A GRAMMAR:

1) Compute the *Null* set.
2) For every production rulle P, add the alternative rules that can be obtained from the rule P by deleting each nullable non-terminal occurring in the right member of the rule, in all possible combinations.
3) Remove all the empty production rules of the type $A \rightarrow \varepsilon$, but when $A = S$.
4) Remove $S \rightarrow \varepsilon$ if and only if the language does not contain ε.
5) Put the grammar in reduced form and remove circular derivations, if any.

EXAMPLE (continued)

nullable	G original	G'' to be reduced	G'' without empty rules
F	$S \rightarrow SAB \mid \mid AC$	$S \rightarrow SAB \mid SA \mid SB \mid \mid S \mid AC \mid C$	$S \rightarrow SAB \mid SA \mid SB \mid \mid AC \mid C$
V	$A \rightarrow aA \mid \varepsilon$	$A \rightarrow aA \mid a$	$A \rightarrow aA \mid a$
V	$B \rightarrow bB \mid \varepsilon$	$B \rightarrow bB \mid b$	$B \rightarrow bB \mid b$
F	$C \rightarrow cC \mid c$	$C \rightarrow cC \mid c$	$C \rightarrow cC \mid c$

COPY RULES (cathegorization) AND HOW TO ELIMINATE THEM

Copy rule (or cathegorization rule): in the example, the syntactic class B is included in the syntatctic class A. By removing the copy rules, one obtains an equivalent grammar that has less deep syntactic trees.

Copy (A) $\subseteq$ V: the set of the non-terminal symbols A may be copied into, possibly in a transitive way.

$$A \to B \quad B \in V$$

$$frase\_iterativa \to frase\_while \mid frase\_for \mid frase\_repeat$$

$$Copy\ (A) = \left\{ B \in V \mid \text{there exists the derivation } A \overset{*}{\Rightarrow} B \right\}$$

Copy rules are not totally unuseful, as sometimes they allow to share some parts of the grammar. For this reason copy rules may be used in the grammars of technical languages, like for instance programming languages.

1) Compute the set *Copy* (assume the grammar does not contain empty rules). The computation is expressed by the following logicals clauses, to be iterated as long as a fixpoint has been reached.

$$A \in Copy\ (A)$$

$$C \in Copy\ (A) \text{ if } (B \in Copy\ (A)) \wedge (B \to C \in P)$$

2) Construct the production rules of the grammar G', equivalent to G but without copy rules.

$$P' := P \setminus \{A \to B \mid A, B \in V\}$$

$$P' := \{A \to \alpha \mid \alpha \in ((\Sigma \cup V)^* \setminus V)\}$$

where $B \to \alpha \in P \wedge B \in Copy\ (A)$

$A \overset{*}{\Rightarrow} B \Rightarrow \alpha$ becomes $A \Rightarrow \alpha$

EXAMPLE – grammar of the arithmetic expressions without copy rules

standard non-ambiguous grammar but with copy rules

$$E \to E + T \mid T \quad T \to T \times C \mid C$$

$$C \to 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9$$

$$Copy(E) = \{E, T, C\} \quad Copy(T) = \{T, C\}$$

$$Copy(C) = \{C\}$$

equivalent grammar without copy rules

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

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

$$C \to 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9$$

CHOMSKY NORMAL FORM: the production rules are only of two types

1. binary rule (of the homogeneous type)

2. terminal rule (with one-letter right side)

$$A \to BC \quad \text{where} \quad B, C \in V$$

$$A \to \alpha \quad \text{where} \quad \alpha \in \Sigma$$

If the language contains the empty string, add the rule: $S \to \varepsilon$

Structure of the syntax tree: internal nodes all of arity 2, and nodes that are the parent of a leaf all of arity 1.

transformation

$$A_0 \to A_1 A_2 \ldots A_n$$

$$A_0 \to A_1 < A_2 \ldots A_n >$$

$$< A_2 \ldots A_n > \to A_2 \ldots A_n$$

if A_1 is a terminal $< A_1 > \to A_1$

EXAMPLE – transformation into Chomsky normal form

$$S \to dA \mid cB \quad A \to dAA \mid cS \mid c \quad B \to cBB \mid dS \mid d$$

$$S \to < d > A \mid < c > B$$

$$A \to < d >< AA > \mid < c > S \mid c$$

$$B \to < c >< BB > \mid < d > S \mid d$$

$$< d > \to d \qquad < c > \to c$$

$$< AA > \to AA \qquad < BB > \to BB$$

HOW TO TRANSFORM LEFT RECURSION INTO RIGHT RECURSION

Construction of the NON-LEFT RECURSIVE FORM (indispensable for designing left recursive descent syntactic analyzers).

1) TRANSFORMATION OF IMMEDIATE LEFT RECURSION

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

where no β_i is empty

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

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

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

$$A \Rightarrow A\beta_2 \Rightarrow A\beta_3\beta_2 \Rightarrow \gamma_1\beta_3\beta_2$$

$$A \Rightarrow \gamma_1 A' \Rightarrow \gamma_1\beta_3 A' \Rightarrow \gamma_1\beta_3\beta_2$$

EXAMPLE – how to shift to the right a left immediate recursion

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

$E \quad T$ are immediately recursive on the left

$E \to TE' \mid T \qquad E' \to +TE' \mid +T$

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

simple transformation but does not work always

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

2) TRANSFORMATION OF NON-IMMEDIATE RECURSION

HYPOTHESIS: G is non-homogeneous non-nullable form, with rules of length one (similar to the Chomsky normal form but without having two non-terminals).

The ALGORITHM works iteratively, in two interleaved steps:

1) expansion to expose left immediate recursions
2) replacement of left recursion with right recursion

It is adivisable to denumerate the non-terminal symbols from 1 to *m:*

$$V = \{A_1, A_2, ..., A_m\}$$

A_1 is the axiom

IDEA of the algorithm: modify the rules so that, if a rule is $A_i \rightarrow A_j$, it holds $j > i$.

ALGORITHM TO ELIMINATE LEFT RECURSION (possibly non-immediate)

for i := 1 **to** m **do**
  for j := 1 **to** i –1 **do**
    replace every rule of the form $A_i \rightarrow A_j \alpha$
    with the following rules:
    $A_i \rightarrow \gamma_1 \alpha \mid \gamma_2 \alpha \mid \ldots \mid \gamma_k \alpha$
    (this may introduce new left immediate recursion)
    where $A_j \rightarrow \gamma_1 \mid \gamma_2 \mid \ldots \mid \gamma_k$ are the alternatives for A_j
  end do
  eliminate, by means of the previous algorithm, the possible left immediate recursion that occur in the alternatives of A_i, and introduce the new non-terminal A'_i
end do

This same algorithm can be adapted to transform right recursion into left recursion.

Such transformation is sometimes required to design bottom-up syntactic analyzers.

EXAMPLE – apply the algorithm to grammar G_3

$$A_1 \to A_2 a \mid b \quad A_2 \to A_2 c \mid A_1 d \mid e$$
$$A_1 \Rightarrow A_2 a \Rightarrow A_1 da$$

i = 1 Eliminate the left immediate recursion of A_1 (here it does not exist). grammar is unchanged

i = 2 j = 1 Replace $A_2 \to A_1 d$ by means of the rules obtained by expanding A_1

$$A_1 \to A_2 a \mid b$$
$$A_2 \to A_2 c \mid A_2 ad \mid bd \mid e$$

Eliminate the left immediate recursion and eventually obtain G'_3

$$A_1 \to A_2 a \mid b$$
$$A_2 \to bdA' \mid eA'$$
$$A' \to cA' \mid adA' \mid e$$

REAL TIME NORMAL FORM (and GREIBACH NORMAL FORM)

In the REAL TIME NORMAL FORM every production rule begins with a terminal symbol.

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

The GREIBACH NORMAL FORM is a special case of the above.

Every rule begins with a terminal symbol, followed by zero or more non-terminal symbols.

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

"REAL TIME" - this terms is justified as a property of the syntax analysis algorithm, which at every step reads and consumes exactly one terminal symbol. Therefore, the number of steps needed to complete the analysis is equal to the length of the string to be analyzed.

ALGORITHM TO TRANSFORM THE GRAMMAR INTO REAL TIME NORMAL FORM AND GREIBACH NORMAL FORM

HYPOTHESIS: the grammar does not contain any nullable non-terminal symbol

1) elminate left recursion (immediate or not)
2) by means of elementary transformations, expand the non-terminal symbols possibly on the left end of the right member of the production rules
3) introduce new non-terminal symbols to replace the terminal symbols possibly occurring inside of the right member of the production rules (or at the right end)

If the last step of the above algorithm were skipped, the production rules might still contain terminal symbols in the middle or at the right end.

The grammar would then be in real time normal form, though in general not in Greibach normal form.

EXAMPLE – grammar to start from

$$A_1 \to A_2 a \quad A_2 \to A_1 c \mid b A_1 \mid d$$

1) eliminate left recursion

$$A_1 \to A_2 a \quad A_2 \to A_2 ac \mid b A_1 \mid d$$
$$A_1 \to A_2 a \quad A_2 \to b A_1 A'_2 \mid d A'_2 \mid d \mid b A_1 \quad A'_2 \to ac A'_2 \mid ac$$

2) replace iteratively each leftmost non-terminal symbol, as long as a terminal symbol takes its place

$$A_1 \to b A_1 A'_2 a \mid d A'_2 a \mid da \mid b A_1 a$$
$$A_2 \to b A_1 A'_2 \mid d A'_2 \mid d \mid b A_1$$
$$A'_2 \to ac A'_2 \mid ac$$

3) replace the terminal symbols occurring in the middle of the production (or at the right end) by means of non-terminal symbols (and of the related productions).

$$A_1 \to b A_1 A'_2 < a > \mid d A'_2 < a > \mid d < a > \mid b A_1 < a >$$
$$A_2 \to b A_1 A'_2 \mid d A'_2 \mid d \mid b A_1$$
$$A'_2 \to a < c > A'_2 \mid a < c >$$
$$< a > \to a \quad < c > \to c$$

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)