Context Free Grammars - III

Prof. A. Morzenti

STRONG (STRUCTURAL) AND WEAK EQUIVALENCE

WEAK EQUIVALENCE: two grammars are weakly equivalent if they generate the same language: $L(G) = L(\text{G'})$.

G and *G'* might assign different structures (syntax trees) to the same sentence

the structure assigned to a sentence is important: it is used by translators and interpreters

STRONG or STRUCTURAL EQUIVALENCE of two grammars *G* and *G'*
$L(G) = L(G')$ (weak eq.) **and**
G and *G'* have the same ***condensed skeleton trees***

strong eq. $\rightarrow$ weak eq. **but** strong eq. $\neq$ weak eq.

(hence $\neg$(weak eq. $\rightarrow$ strong eq.))

strong eq. is **decidable**

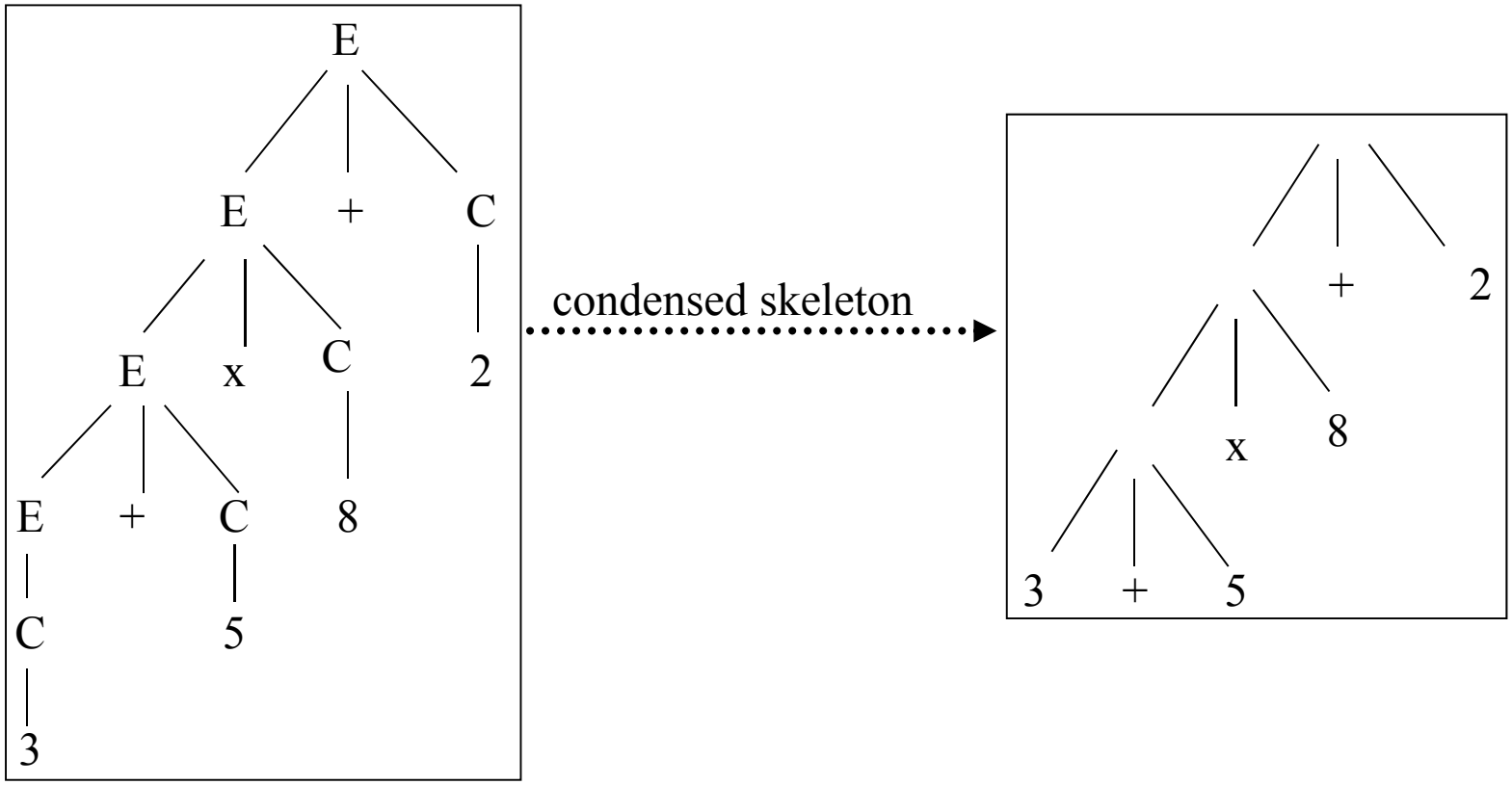
weak eq. is **not decidable**

(it can happen that one can establish that G_1 and G_2 **are not** strongly equivalent

but is unable to establish anything concerning their weak equivalence)

Example: Structural equivalence of arithmetic expressions: 3 + 5 x 8 + 2

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

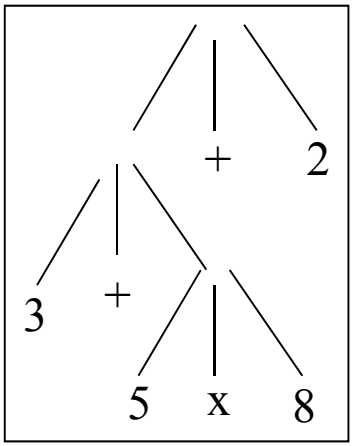


ANOTHER GRAMMAR FOR THE SAME LANGUAGE

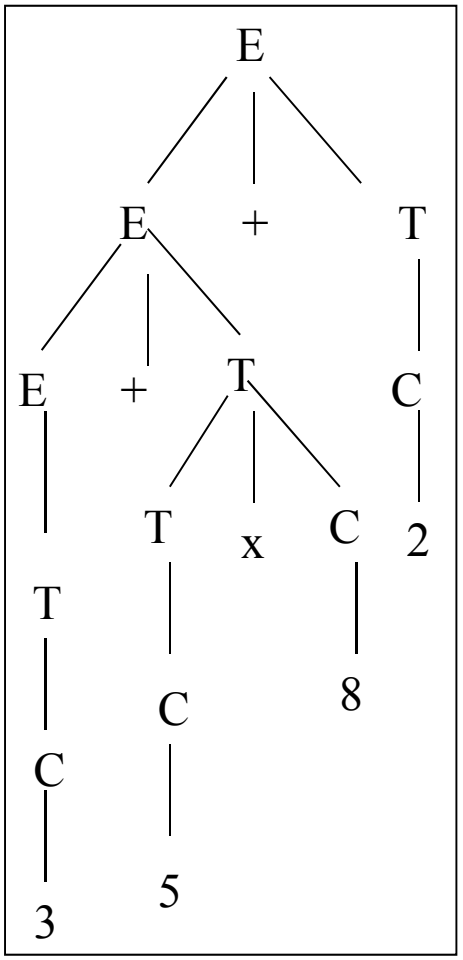
NB: copy (or categorization) rules

G_2 : $E \to E + T$ $E \to T$ $T \to T \times C$ $T \to C$

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



condensed skeleton



$G_1$and G_2 are not structurally equivalent

Semantic interpretations: G_1: (((3 + 5) x 8) + 2)
G_2: ((3 + (5 x 8)) + 2)

Only G_2 is **structurally adequate** w.r.t. the operator precedence rules (NB: G_2 is more complex): it forces the generation of a product from n.t. *T* only ***after*** *E* therefore it assigns a higher precedence to the product than to the sum
NB: **this is how operator priority can be enforced through grammars**

STRUCTURAL ADEQUACY must be carefully considered
it supports syntax-directed interpretation and translation

another grammar G_3 structurally equivalent to the previous one
(NB structural equiv. concerns ***condensed trees***):

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

G_3 has more rules: it does not exploit **categorization rules** as G_2 does

categorization can reduce the complexity of grammars

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

$Null \subseteq V$ is the set of nullable n.t.

computation of the set $Null$

$A \in Null$ if $A \rightarrow \varepsilon \in P$

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

NB: recursive rules cannot be used

Example – Computing nullable nonterminals

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

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

NB: If the grammar included rule S $\to$ AB then the result would include $S \in Null$

NORMAL FORM WITHOUT NULLABLE NONTERMS (non-nullable, with no empty rule)

defined as the condition that no nonterm. other than the axiom is nullable

The axiom is nullable only if the empty string ε is in the language

CONSTRUCTION OF THE NON-NULLABLE NORMAL FORM:

1) compute the *Null* set
2) for each rule $\in P$ add as alternatives those obtained by deleting, in the right part, **in all possible ways (NB: combinatorial effect)**, the nullable nonterm.
3) remove all empty rules $A \to \varepsilon$, except for $A = S$
4) clean the grammar and remove any circularity

Example (follows)

delete A and B in all possible ways, remove rules of type $X \to \varepsilon$

delete rule $S \to S$

Nullable	original G	G'' to be cleaned	G'' with no empty rules
F	$S \to SAB \mid \mid AC$	$S \to SAB \mid SA \mid SB \mid \mid S \mid AC \mid C$	$S \to SAB \mid SA \mid SB \mid \mid AC \mid C$
T	$A \to aA \mid \varepsilon$	$A \to aA \mid a$	$A \to aA \mid a$
T	$B \to bB \mid \varepsilon$	$B \to bB \mid b$	$B \to bB \mid b$
F	$C \to cC \mid c$	$C \to cC \mid c$	$C \to cC \mid c$

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***

Define ***Copy*(*A*) ⊆ *V***

set of n.t. into which the n.t. ***A*** can be copied, possibly transitively

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

1) Computation of ***Copy*** (assume a **grammar with non empty rules**) by applying logical clauses until a fixpoint is reached

(it is simply the reflexive, transitive closure of the «copy» relation defined by copy rules)

$$\boxed{\begin{array}{l} A \in Copy(A) \\ C \in Copy(A) \text{ if } (B \in Copy(A)) \wedge (B \to C \in P) \end{array}}$$

(NB: since there is no empty rule, cases such as $B \to CD$ and $D \overset{*}{\Rightarrow} \varepsilon$ can be ruled out)

2) Definition of the rules of a grammar *G*', equivalent to *G* but without copy rules

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

$P' := P' \cup \{A \to \alpha \mid \exists B (B \in Copy(A) \wedge (B \to \alpha) \in P)\}$ -- add new "compensating" rules

(NB: for each removed copy rule, ***many*** other (non-copy) rules may be added
$\Rightarrow$ the set of rules may increase considerably in size

The derivation $A \overset{*}{\Rightarrow} B \Rightarrow \alpha$ shrinks to $A \overset{*}{\Rightarrow} \alpha$

Example – Eliminating copy rules from a grammar of arithmetic expressions

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

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

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

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

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

$T \in Copy(E)$ $C \in Copy(E)$

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

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

$C \in Copy(T)$

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

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:* $\boldsymbol{A \to BC}$ *with* $\boldsymbol{B,C \in V}$
2. *Terminal rules with singleton right part* $\boldsymbol{A \to a}$, $\boldsymbol{a \in \Sigma}$

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

Procedure to obtain from *G* (assumed without nullable n.t.) its Chomsky normal form

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

Then apply iteratively the following process

for each rule of type ⟶ $A_0 \to A_1 A_2 \ldots A_n$

add a rule of type … ⟶ $A_0 \to < A_1 >< A_2 \ldots A_n >$ ⟵ **<$A_2 \ldots A_n$> a new ancillary n.t. introduced for this purpose**

… and also another rule … ⟶ $< A_2 \ldots A_n > \to A_2 \ldots A_n$

finally, soon or late … ⟶ if A_1 is terminal $< A_1 > \to A_1$

Example: Conversion to Chomsky normal form

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

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

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

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

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

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

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