Free Grammars - B

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, i.e., iff $L(G) = L(G')$

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, i.e., iff $L(G) = L(G')$, 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)

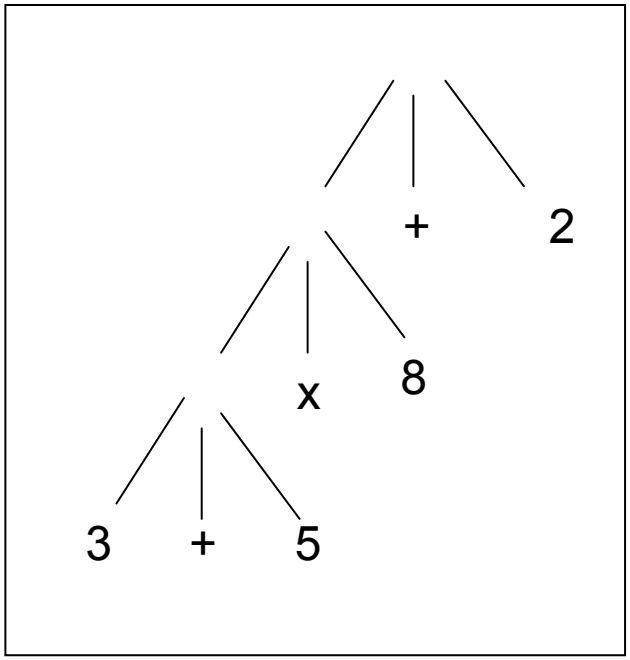
Strong equivalence implies weak equivalence, but the opposite implication does not hold

Strong equivalence is a *decidable* property

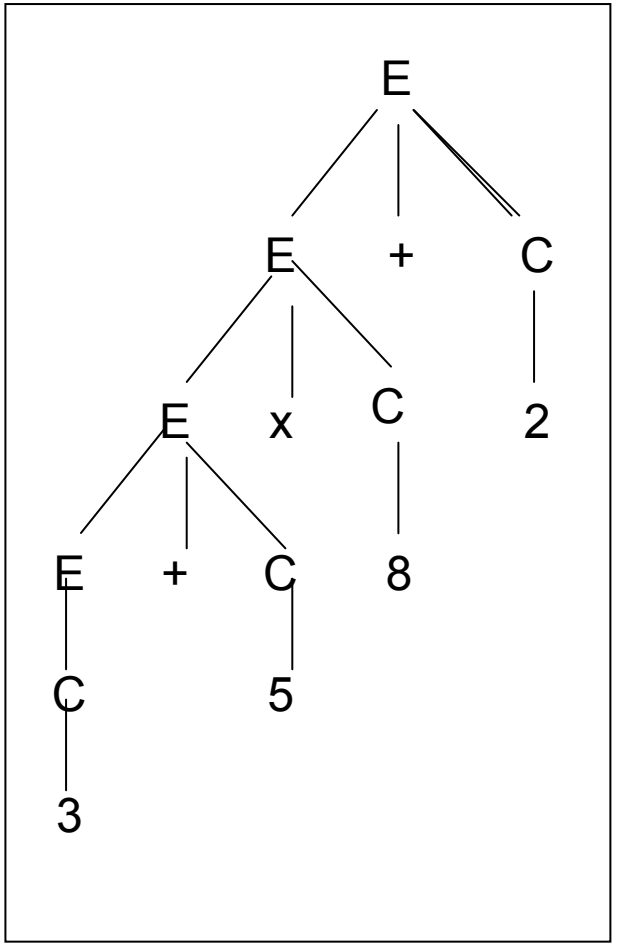
Weak equivalence is an *undecidable* property (but in some special cases)

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

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



skeleton tree



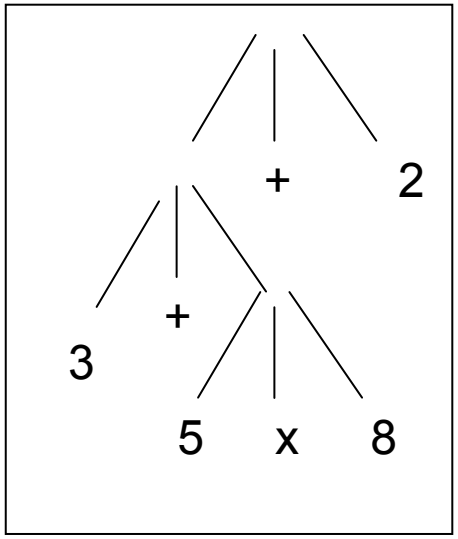
full tree

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

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

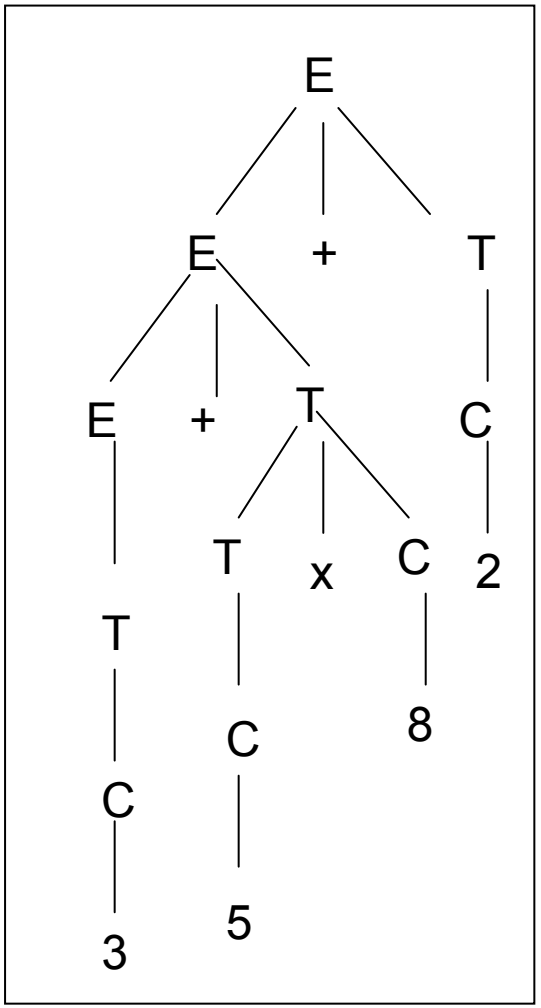
semantic interpretations
G_1: (((3 + 5) × 8) + 2)
G_2: ((3 + (5 × 8)) + 2)

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



GRAMMAR NORMAL FORM

Normal forms impose restrictions to the rules, but without reducing the family of generated languages. Such forms are useful for instance to prove theorems, rather than to design 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

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

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

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

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

REMOVING THE AXIOM FROM THE RIGHT MEMBER OF THE RULES

It is always possible to restrict the right member of a rule so as it is a string over the alphabet ($\Sigma \cup (V - \{ S \})$), i.e., it does not contain the axiom S

It suffices to introduce the new axiom S_0 and the new 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 generates 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 rule P, add the alternative rules that can be obtained from rule P by deleting each nullable non-terminal that occurs in the right member of the rule, in all possible combinations
3) remove all the empty rules of the type $A \rightarrow \varepsilon$, except when $A = S$
4) remove $S \rightarrow \varepsilon$ if and only if the language does not contain ε
5) put the grammar into reduced form and remove the circular derivations

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 with syntactic trees that are less deep

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

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

iterative_phrase $\to$ *while_phrase* | *for_phrase* | *repeat_phrase*

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

Copy rules are not totally useless, as sometimes they allow us to share some parts of the grammar. For this reason copy rules may be used in the grammars of technical languages, like for instance in the 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 until a fixpoint is reached

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

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

2) Construct the 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: every internal node has arity 2, and every node that is parent of a leaf has 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 a left immediate recursion to the right

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

$E \quad T$ 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 always work

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

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 $j > i$

GRAMMARS FOR REGULAR LANGUAGES

Regular languages are a special case of the free ones

Regular languages can be generated by grammars that exhibit strongly restricted rules

The phrases of a regular language necessarily contain repeated factors as the length of the string grows

It is possible to prove in a rigorous way that some (free) languages cannot be generated by any regular expression

REGULAR LANGUAGES

Recognizing phrases requires only a finite amount of memory

FREE LANGUAGES

Recognizing phrases requires an unbounded amount of memory

FROM THE REGULAR EXPRESSION TO THE FREE GRAMMAR

The iterative regular operators (star and cross) must be replaced with recursive rules

Decompose the regexp into subexpressions and denumerate them progressively. After the very definition of regexp, the following cases are obtained, which allow Us to design the corresponding rules shown aside

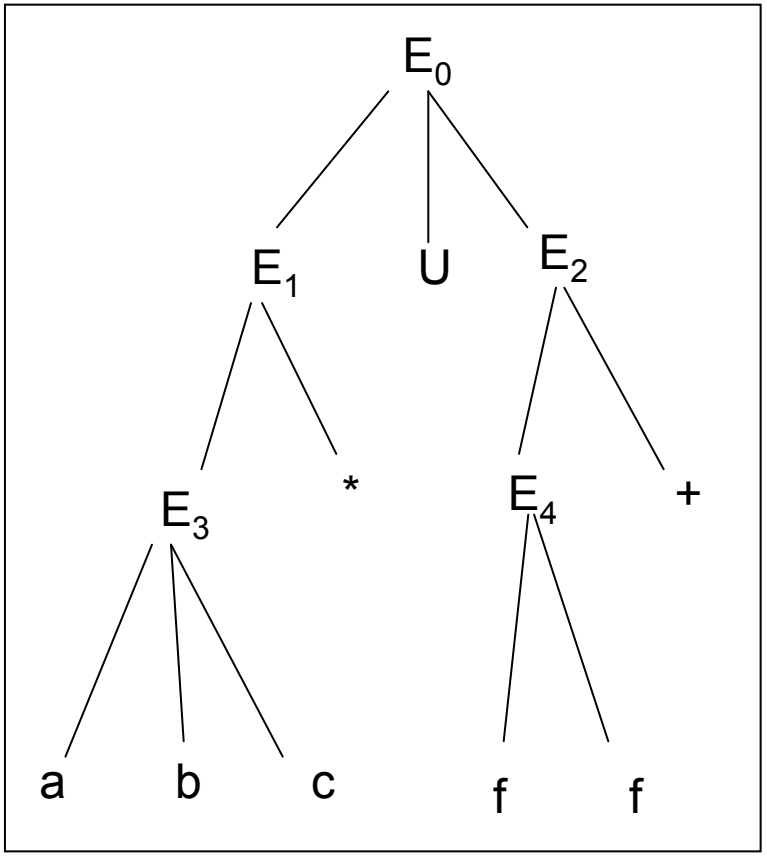
1. $r = r_1.r_2....r_k$	1. $E = E_1E_2...E_k$
2. $r = r_1 \cup r_2 \cup .... \cup r_k$	2. $E = E_1 \bigcup E_2 \bigcup ... \bigcup E_k$
3. $r = (r_1)^*$	3. $E = EE_1 \mid \varepsilon$ or $E = E_1E \mid \varepsilon$
4. $r = (r_1)^+$	4. $E = EE_1 \mid E_1$ or $E = E_1E \mid E_1$
5. $r = b \in \Sigma$	5. $E = b$
6. $r = \varepsilon$	6. $E = \varepsilon$

Uppercase letters indicate the non-terminal symbols. If E_i is a terminal symbol $a_i \in \Sigma$, the symbol a_i is written instead of E_i

EXAMPLE

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

2.	$E_1 \cup E_2$	$E_0 \to E_1 \mid E_2$
3.	E_3^*	$E_1 \to E_1 E_3 \mid \varepsilon$
4.	E_4^+	$E_2 \to E_2 E_4 \mid E_4$
1.	abc	$E_3 \to abc$
1.	ff	$E_4 \to ff$



If the regexp to start from is ambiguous, the derived equivalent free grammar is ambiguous as well

Every regular language is free, but there are many free languages that are not regular, like for instance the palindromes and the arithmetic expressions with parentheses

$$REG \subset LIB$$

UNI-LINEAR GRAMMAR (or GRAMMAR of TYPE 3)

RIGHT UNI-LINEAR RULE (or simply RIGHT LINEAR)

$$A \to uB \quad \text{where} \quad u \in \Sigma^*, B \in (V \bigcup \varepsilon)$$

LEFT UNI-LINEAR RULE (or simply LEFT LINEAR)

$$A \to Bv \quad \text{where} \quad v \in \Sigma^*, B \in (V \bigcup \varepsilon)$$

The corresponding syntax trees grow in a heavily unbalanced way

If a left or right uni-linear grammar contains a recursive derivation, then such a derivation is recursive on the left or right, respectively

EXAMPLE: the phrases that contain the substring *aa* and terminate with *b* are generated by the following regexp (ambiguous)

$$(a \mid b)^* aa(a \mid b)^* b$$

1. right linear grammar G_d

$S \to aS \mid bS \mid aaA \quad A \to aA \mid bA \mid b$

2. left linear grammar G_s

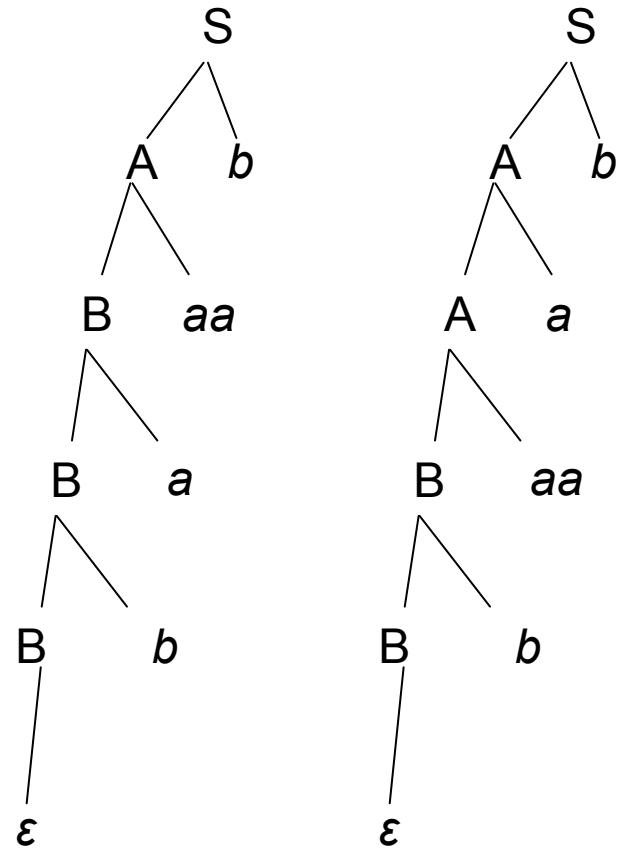
$S \to Ab \quad A \to Aa \mid Ab \mid Baa$

$B \to Ba \mid Bb \mid \varepsilon$

3. equivalent non-unilinear grammar

$E_1 \to E_2 aaE_2 b \quad E_2 \to E_2 a \mid E_2 b \mid \varepsilon$

G_s – syntax trees of the ambiguous phrase *baaab*



EXAMPLE : arithmetic expressions without parentheses

$$L = \{a, a+a, a*a, a+a*a, \ldots\}$$

$$G_d: \quad S \to a \mid a+S \mid a*S$$

$$G_s: \quad S \to a \mid S+a \mid S*a$$

G_s and G_d are not adequate to compilation, as their syntax structure does not reflect the usual precedence rules between arithmetic operators. It is convenient to transform a unilinear grammar into a strictly unilinear grammar, where every rule contains at most a single terminal character

$$A \to aB \quad \text{or} \quad A \to Ba \quad \text{where} \quad a \in (\Sigma \cup \varepsilon), B \in (V \cup \varepsilon)$$

Without any loss of generality, one can impose to have only null terminal rules

$$A \to aB \mid \varepsilon \quad \text{where} \quad a \in \Sigma$$

FROM UNILINEAR GRAMMAR TO REGULAR EXPRESSION: LINEAR EQUATIONS

UNILINEAR GRAMMAR / REGULAR LANGUAGE
First it is proved that the family of the languages generated by unilinear grammars strictly contains that of regular languages. Then it is proved that these two families actually coincide

One can think of the rules of a right (uni)linear grammar as of equations, the unknown terms of which are the languages generated by each non-terminal symbol.

Suppose that G is a strictly right linear grammar and that G contains only null terminal rules $A \rightarrow \varepsilon$

$$G = (V, \Sigma, P, S)$$

$$L_A = \left\{ x \in \Sigma^* \mid A \overset{+}{\Rightarrow} x \right\} \quad L(G) \equiv L_S$$

replace the (recursive) rule $A \rightarrow a\ A$ with the equation $L_A = L_a\ L_A$

replace the rule $A \rightarrow b\ B$ with the equation $L_A = L_b\ L_B$

replace the rule $A \rightarrow a$ with the equation $L_A = L_a$

unite terminal rules to non-terminal rules:
$A \rightarrow b\ B$ and $A \rightarrow c$ so that $L_A = L_c \mid L_b\ L_B$

and so on when there are more terms

then solve the equations by substitution

in the case of a recursive equation $L_A = L_x \mid L_y\ L_A$,
solve it by means of the Arden identity: $L_A = L_y^*\ L_x$

EXAMPLE: set-theoretic equations

Here follows a grammar that generates a list of elements *e* (possibly empty) separated by the character *s*

$$S \to sS \mid eA \quad A \to sS \mid \varepsilon$$

The grammar can be transformed into a set of equations, as aside

$$\begin{cases} L_s = sL_S \cup eL_A \\ L_A = sL_S \cup \varepsilon \end{cases}$$

$$\begin{cases} L_s = sL_S \cup e(sL_S \cup \varepsilon) \\ L_A = sL_S \cup \varepsilon \end{cases}$$

$$\begin{cases} L_s = (s \cup es)L_S \cup e \\ L_A = sL_S \cup \varepsilon \end{cases}$$

and by using Arden identity one obtains

$$\begin{cases} L_s = (s \cup es)^* e \\ L_A = sL_S \cup \varepsilon \end{cases} \qquad L_A = s(s \cup es)^* e \cup \varepsilon$$

EVERY UNILINEAR LANGUAGE IS REGULAR

THE ROLE OF SELF-INCLUSIVE DERIVATIONS

$$A \overset{+}{\Rightarrow} uAv \quad u \neq \varepsilon \wedge v \neq \varepsilon$$

A self-inclusive derivation may not occur in the unilinear grammar of regular languages. This does not happen in the free grammars that generate free non-regular languages (palindromes, Dyck, etc)

The absence of self-inclusive derivations allows to solve the set-theoretic equations associated with unilinear grammars, and gives rise to the simple structure of the languages that such grammars generate

A GRAMMR THAT DOES NOT ADMIT ANY SELF-INCLUSIVE DERIVATION ALWAYS GENERATES A REGULAR LANGUAGE

THE LIMITATIONS OF FREE LANGUAGES

EXAMPLE: THE LANGUAGE WITH THREE EXPONENTS IS NOT FREE
(the proof exploits the possibility of pumping a long phrase by repeating two independent factors, which is incompatible with the definition of such a language)

$$L = \left\{ a^n b^n c^n \mid n \geq 1 \right\}$$

EXAMPLE: THE COPY LANGUAGE

This sub-language is frequent in the programming languages, whenever two lists need to have identical matching elements, like for instance in the lists of the formal and actual parameters of a procedure of function declaration and call

$$L_{\,replic} = \left\{ uu \mid u \in \Sigma^+ \right\}$$

$$\Sigma = \left\{ a, b \right\} \quad x = abbbabbb$$

THE CLOSURE PROPERTIES OF THE REG AND LIB FAMILIES

Here the closure properties of the REG and LIB families of languages are examined, with respect to the most common language operators

Let L and R be two languages belonging to LIB and REG, respectively

$R^R \in REG$	$L^R \in LIB$
$R^* \in REG$	$L^* \in LIB$
$\neg R \in REG$	$\neg L \notin LIB$
$R_1 \oplus R_2 \in REG$	$L_1 \oplus L_2 \in LIB$
$R_1 \bigcap R_2 \in REG$	$L_1 \bigcap L_2 \notin LIB$
	$L \bigcap R \in LIB$

denotes both union and concatenation

denotes both union and concatenation

4) The intersection of two free languages, in general, is not a free language

$$\left\{a^n b^n c^n \mid n \geq 1\right\} = \left\{a^n b^n c^+ \mid n \geq 1\right\} \bigcap \left\{a^+ b^n c^n \mid n \geq 1\right\}$$

5) There exist free languages the complements of which (w.r.t. the universal language over the same alphabet) are not free languages

CAUTION: of course, there exist also free languages the complements of which are still free languages (and similarly for intersection)

INTERSECTION OF FREE LANGUAGES WITH REGULAR LANGUAGES

One way to make a free grammar more selective, is to restrict the generated free language through a regular filter

EXAMPLE: regular filter over the Dyck language - $\Sigma = \{ a, c \}$ (*a* open *c* close)

$$L_1 = L_D \bigcap \neg(\Sigma^* cc \Sigma^*) = (ac)^*$$

$$L_2 = L_D \bigcap \neg(\Sigma^* ca \Sigma^*) = \left\{ a^n c^n \mid n \geq 0 \right\}$$

The former intersection filters the Dyck language by removing the phrases that contain the factor *cc*, that is the phrases containing nested structures

The latter intersection filters the Dyck language by removing the phrases that contain two or more concatenated nested structures, leaving only the phrases with one nested structure

Both filtered languages are free, and the former one is regular as well

FREE GRAMMARS EXTENDED BY MEANS OF REGULAR EXPRESSIONS

REGULAR EXPRESSIONS ARE MORE READABLE AND PERSPICUOUS than free grammars, thanks to the iteration operators (star and cross) and to the choice operator (union)

To make free grammars more readable as well, it is often permitted to use regexps also in the right sides of the rules of the grammar. One thus obtains the so-called EXTENDED FREE GRAMMARS (Extended BNF or EBNF). The grammars of the technical languages (programming languages, description languages, etc) are usually presented in an extended form

The EBNF rules can be represented as syntax diagrams. A syntax diagram can be conceived as the flow graph of a syntax analysis algorithm

The family LIB of free languages is closed with respect to regular operators (union, star, concatenation); therefore the extended free grammars have the same generating power as the non-extended ones

EXAMPLE: list of declarations of variable names

character testo1, testo2; *real* temp, result;

integer alfa, beta2, gamma;

$\Sigma = \{c, i, r, v, \ ',' \ , \ ';'\}$ where *v* is a variable name

$\left(\left(c \mid i \mid r \right) v (, v)^{*} ; \right)^{+}$

$S \to SE \mid E \quad E \to AF; \quad A \to c \mid i \mid r \quad F \to v, F \mid v$

The grammar is longer than the regexp, and makes less evident the existence of a hierarcy of two lists. Moreover, the names A, E and F are arbitrary

A FREE EXTENDED GRAMMAR (EBNF) G = { V, Σ, P, S } contains exactly |*V* | rules, each of the form A → η, where η is a regexp over the alphabet V U Σ. For better clarity it is allowed to use derived regular operators, like for instance cross, optionality, repetition, etc

EXAMPLE: a simplified Algol-like language

A more compact but less readable version. The non-terminal B cannot be eliminated, as it is indispensable to generate nested structures

$$B \to b\left[D\right]Ie$$

$$D \to ((c \mid i \mid r)v(,v)^{*};)^{+}$$

$$I \to F(;F)^{*}$$

$$F \to a \mid B$$

$$B \to b\left[((c \mid i \mid r)v(,v)^{*};)^{+}\right]F(;F)^{*}e$$

$$B \to b((c \mid i \mid r)v(,v)^{*};)^{*}F(;F)^{*}e$$

$$B \to b((c \mid i \mid r)v(,v)^{*};)^{*}(a \mid B)(;(a \mid B))^{*}e$$

DERIVATION AND SYNTAX TREE IN THE EXTENDED GRAMMARS

The right side of an extended rule is a regexp that defines an infinite set of strings

Such strings could be imagined as the right sides of a grammar G', equivalent to G, except that it contains infinitely many rules

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

$$A \to aB \mid aBaB \mid ....$$

DERIVATION RELATION IN AN EXTENDED GRAMMAR

given the strings η_1 $\eta_2 \in (\Sigma \bigcup V)^*$

say that η_2 *derives* (immediately) from η_1

$\eta_1 \Rightarrow \eta_2$ if the two strings are factored as follows

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

$A \to e: \quad e \overset{*}{\Rightarrow} \vartheta$

note that η_1 η_2 ($\alpha A \gamma$ $\alpha \vartheta \gamma$) do not contain

either regular operators or parentheses, and *e* is a regexp

Similarly, one can define multiple step derivations and the generated language

EXAMPLE: extended derivation for arithmetic expressions

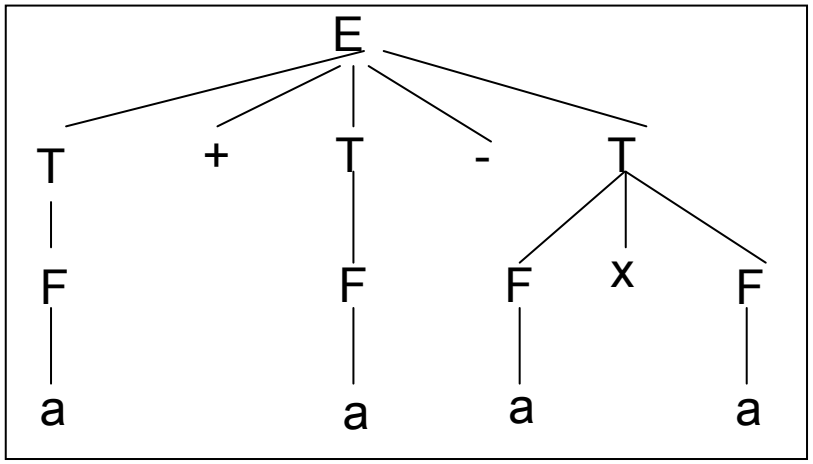
The extended grammar G shown below generates the arithmetic expressions with the four standard operators (in infix notation), round brackets and the symbol *a* to represent generic variables or constants

$$E \Rightarrow [+|-]T((+|-)T)^* \quad T \rightarrow F((\times|/)F)^* \quad F \rightarrow (a|'('E')')$$

leftmost derivation

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

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



In EBNF, the arity of a tree node may be unlimited, in general

However, as the tree gets larger the depth gets lower

AMBIGUITY IN EXTENDED GRAMMARS

A non-extended but ambiguous grammar is ambiguous also when conceived as extended (as BNF is a special case of EBNF)

However, the presence of regexps in the rules may give rise to specific ambiguity forms

$a^*b \mid ab^*$ numbered as $a_1^*b_2 \mid a_3b_4^*$

is ambiguous because ab is

derivable as a_1b_2 or as a_3b_4

$S \to a^*b \mid ab^*$ is ambiguous as well