Chomsky Grammars

In general, to describe a language, there are two possible approaches:

1) recognition: describe rules (or a mechanism) to determine whether or not a certain string belongs to a language

   e.g. an automaton is such a mechanism

2) generation: define rules to generate all strings of a language

A grammar is a formalism for defining a language in terms of rules that generate all strings of the language.

Since 1920, various formal methods based on the notion of rewriting or derivation have been proposed by Axel Thue, Emil Post, A.A. Markov.

In the mid 1950s the linguist Noam Chomsky introduced the notion of formal grammar with the aim of formalizing natural language. Formal grammars are in fact too simplicistic to capture natural language, but they were adopted as the main formal tool to define syntactic properties of artificial languages (e.g. programming languages)

<u>Definition</u>: Given an alphabet Σ, a (formal) grammar G is a quadruple $G = (V_N, V_T, P, S)$ where

- $V_T \subseteq \Sigma$ is a finite nonempty set of symbols called terminals
- V_N is a finite nonempty set of symbols s.t. $V_N \cap \Sigma = \emptyset$, called variables or nonterminals, or syntactic categories. Each variable represents a language
- $S \in V_N$ is called start symbol or axiom, and represents the language being defined by G
- P is a binary relation over

$$(V_N \cup V_T)^* \cdot V_N \cdot (V_N \cup V_T)^* \quad \times \quad (V_N \cup V_T)^*$$

Each element $(\alpha, \beta) \in P$ is called a production or rule, and is generally written as: $\alpha \rightarrow \beta$.

Note : α ... sequence of terminals and nonterminals with at least one nonterminal

β ... sequence of terminals and nonterminals

<u>Definition</u>: The language $\mathcal{L}(G)$ generated by a grammar G is the set of strings of terminals only that can be generated starting from the axiom by a finite sequence of rule applications. Each application of a rule $\alpha \rightarrow \beta$ consists in replacing an occurrence of α with β.

Example: Palindromes:

A palindrome is a word that reads the same both forwards and backwards. (AILATIDITALIA, AMOROMA)

$$L_{pal} = \{ w \in \{0,1\}^* \mid w^R = w \}$$

Grammar $G_{pal} = (V_N, V_T, P, S)$, where P consists of

1) $S \rightarrow \varepsilon$
2) $S \rightarrow 0$
3) $S \rightarrow 1$

} basis: ε, 0, 1 are palindromes

4) $S \rightarrow 0S0$
5) $S \rightarrow 1S1$

} induction: if S is a palindrome, so are $0S0$ and $1S1$

Example of derivation:

0110 : $S \overset{4}{\Rightarrow} 0S0 \overset{5}{\Rightarrow} 01S10 \overset{1}{\Rightarrow} 0110$

11011 : $S \overset{5}{\Rightarrow} 1S1 \overset{5}{\Rightarrow} 11S11 \overset{2}{\Rightarrow} 11011$

$\boxed{\text{Exercise E5.1}}$ Prove that the above grammar generates all and only palindromes over $\{0,1\}$.

Hint: use induction on the length of the derivation

Example: natural language generation

Sentence $\rightarrow$ NounPhrase VerbPhrase

NounPhrase $\rightarrow$ Adjective NounPhrase

NounPhrase $\rightarrow$ Noun

Noun $\rightarrow$ car

Noun $\rightarrow$ train

Adjective $\rightarrow$ big

Adjective $\rightarrow$ broken

⋮

Notation:

1) To denote the set of productions

$$\alpha \to \beta_1 \quad , \quad \alpha \to \beta_2 \; , \quad \cdots \; , \quad \alpha \to \beta_m$$

we use

$$\alpha \to \beta_1 \mid \beta_2 \mid \cdots \mid \beta_m$$

2) We use $V = V_N \cup V_T$

A production of the form $\alpha \to \varepsilon$, with $\alpha \in V^* \cdot V_N \cdot V^*$ is called <u>ε-production</u>.

Example: $L_{eq} = \{ w \in \{0,1\}^* \mid w \text{ has equal number of 0's and 1's} \}$

We have already seen that this language is not regular.

Idea to define G_{eq} s.t. $\mathcal{L}(G_{eq}) = L_{eq}$: use induction

base: ε is in L_{eq}

induction: • $0 w_A \in L_{eq}$ if w_A has one more 1 than 0

• $1 w_B \in L_{eq}$ if w_B — " — 0 — " — 1

Characterize also languages for w_A and w_B inductively

Grammar $G_{eq} = (\{S, A, B\}, \{0,1\}, P, S)$ with P

$$S \to \varepsilon \mid 0A \mid 1B$$
$$A \to 1S \mid 0AA$$
$$B \to 0S \mid 1BB$$

(A generates strings with one more 1 than 0.
B generates strings with one more 0 than 1)

Exercise E5.2 Prove that $\mathcal{L}(G_{eq}) = L_{eq}$ (by induction)

<u>Definition</u>: Given G, the <u>direct derivation</u> for G is the binary relation on $(V^* \circ V_N \circ V^*) \times V^*$ defined as follows:

(φ, ψ) is in the relation if there are

$\alpha \in V^* \circ V_N \circ V^*$, $\beta, \gamma, \delta \in V^*$

such that $\varphi = \gamma\alpha\delta$, $\psi = \gamma\beta\delta$ and $\alpha \to \beta \in P$.

We write $\varphi \Rightarrow \psi$ and say that ψ directly derives from φ by G.

<u>Definition</u>: We call <u>derivation</u> the reflexive, transitive closure of direct derivation. In other words, ψ derives from φ by G, written $\varphi \stackrel{*}{\Rightarrow} \psi$ if

a) $\varphi = \psi$, or

b) there are $\varphi_1, \dots, \varphi_n \in V^*$ such that

$\varphi_1 = \varphi$, $\varphi_n = \psi$, and $\varphi_i \Rightarrow \varphi_{i+1}$, $\forall i$, $1 \le i < n$

<u>Definition</u>: Given a grammar G, the <u>language generated by</u> G is

$\mathcal{L}(G) = \{ w \in V_T^* \mid S \stackrel{*}{\Rightarrow} w \}$

Notice: words in $\mathcal{L}(G)$ are constituted by terminals only.

<u>Terminology</u>:

- sentence: any word $w \in V_T^*$ s.t. $S \stackrel{*}{\Rightarrow} t$, i.e. $w \in \mathcal{L}(G)$
- sentential form: any $\alpha \in V^* = (V_T \cup N_N)^*$ s.t. $S \stackrel{*}{\Rightarrow} \alpha$

<u>Notation</u>: terminals: $a, b, c \dots$

nonterminals: $A, B, C, \dots$

strings of terminals: $u, v, w, x, y, z \dots$

symbols of $V = V_N \cup V_T$: $X, Y, Z \dots$

sentential forms: $\alpha, \beta, \gamma \dots$

Example: Productions for G_{eq}:

$$S \to \varepsilon \mid 0A \mid 1B$$
$$A \to 1S \mid 0AA$$
$$B \to 0S \mid 1BB$$

derivation:

1) $001SA \Rightarrow 001S1S$ (using $A \to 1S$)
2) $001S1S \Rightarrow 0011S$ (using $S \to \varepsilon$)
3) $001SA \stackrel{*}{\Rightarrow} 0011S$ (using (1) and (2))
4) $S \stackrel{*}{\Rightarrow} 001110$

Example: Grammar for $L_{3n} = \{a^n b^n c^n \mid n \geq 1\}$ 18/12/2008

$G_{3n} = (\{A, B, C, S\}, \{a, b, c\}, P, S)$

with P
1) $S \to aSBC$
2) $S \to aBC$ } generate $aa..aBCBC..BC$
3) $CB \to BC$ } moves the C's to the end
4) $aB \to ab$
5) $bB \to bb$
6) $bC \to bc$
7) $cC \to cc$
} generate the terminals from left to right
Note: we cannot simply have $B \to b$, $C \to c$ because this would generate many words not in L_{eq}

Example of derivation of aaabbbccc:

$S \stackrel{1}{\Rightarrow} a\underline{S}BC \stackrel{2}{\Rightarrow} aa\underline{S}BCBC \stackrel{2}{\Rightarrow} aaaBCB\underline{CB}C$
$\stackrel{3}{\Rightarrow} aaaB\underline{CB}BCC \stackrel{3}{\Rightarrow} aaaBB\underline{CB}CC$
$\stackrel{3}{\Rightarrow} \underline{aaaB}BBCCC \stackrel{4}{\Rightarrow} aaa\underline{bB}BCCC$
$\stackrel{5}{\Rightarrow} aaab\underline{bB}CCC \stackrel{5}{\Rightarrow} aaabb\underline{bC}CC$
$\stackrel{6}{\Rightarrow} aaabb\underline{bcC}C \stackrel{7}{\Rightarrow} aaabbbc\underline{cC}$
$\stackrel{7}{\Rightarrow} aaabbbccc$

<u>Note</u>: not each sequence of direct derivations leads to a sentence in $\mathcal{L}(G_{3n})$

e.g. with the previous grammar we could generate

$$\underline{S} \Rightarrow a\underline{S}BC \Rightarrow aa\underline{S}BCBC \Rightarrow aaaBC\underline{B}\underline{CB}C$$
$$\Rightarrow aa\underline{aB}CBBCC \Rightarrow aaa\underline{bC}BBCC$$
$$\Rightarrow aaabcBBCC$$

we cannot apply any other production

Also, the application of productions could go on forever (e.g. rule 1 in the previous example)

Classification of Chomsky grammars into 4 groups, depending on the form of the productions:

- grammars of type 0 : no limitations
- — " — 1 : context-sensitive
- — " — 2 : context-free
- — " — 3 : regular (or right linear)

<u>Definition</u>: grammar of type 0.

Productions have the most general form $\alpha \to \beta$, with $\alpha \in V^* \circ V_N \circ V^*$ $\qquad \beta \in V^*$

Grammars of type 0 allow for derivations that shorten the sentential form.

A language generated by a grammar of type 0 is called of type 0.

Definition: grammar of type 1, or context sensitive

Productions have the form $\alpha \to \beta$, with

$$\alpha \in V^* \circ V_N \circ V^* \quad , \quad \beta \in V^+ \quad , \quad |\alpha| \leq |\beta|$$

These productions cannot shorten the length of the sentential form to which they are applied.

A language generated by a grammar of type 1 is called of type 1, or context sensitive.

Example: G_{3n} is context sensitive. Obviously, it is also of type 0.

Definition: grammar of type 2, or context-free

Productions have the form $A \to \beta$, with $A \in V_N$, $\beta \in V^+$.

These productions are productions of type 1, with the additional requirement that on the left there is a single nonterminal.

A language generated by a grammar of type 2 is called of type 2, or context free.

Example $L_{2n} = \{a^n b^n \mid n \geq 1\}$ is of type 1, since the following grammar G'_{2n} generates L_{2n}

$S \to aB \mid SAB$

$BA \to AB$ $\quad aA \to aa$ $\quad aB \to ab$

$bB \to bb$

L_{2n} is also of type 2, since it is generated by

$S \to aSb \mid ab$

OPTIONAL

We said that grammars of type 1 are also called context-sensitive (in contrast to context-free grammar). This is justified by the original definition by Chomsky for context-sensitive grammars:

Definition: Chomsky CS-grammar

Productions have the form $\varphi_1 A \varphi_2 \longrightarrow \varphi_1 \beta \varphi_2$

with $\varphi_1, \varphi_2 \in V^*$, $A \in V_N$, $\beta \in V^+$

Intuitively, A is replaced by β only if it appears "in the context" of φ_1 and φ_2

Theorem: Grammars of type 1 and Chomsky CS grammars generate the same class of languages

Proof: We show that, for every language L:

There is a type-1 grammar G_1 s.t. $L = \mathcal{L}(G_1)$ iff there is a Chomsky CS grammar G_c s.t. $L = \mathcal{L}(G_c)$

"if": immediate, since each Chomsky CS grammar is of type 1 (in $\varphi_1 A \varphi_2 \longrightarrow \varphi_1 \beta \varphi_2$ we have $\beta \in V^+$ and hence $|\varphi_1 A \varphi_2| \leq |\varphi_1 \beta \varphi_2|$)

"only-if": let G_1 be a type-1 grammar for L.

We construct from G_1 a Chomsky CS grammar G_c as follows:

1) for each $a \in V_T$, add a new nonterminal N_a,
2) replace in each production of G_1, each $a \in V_T$ by N_a

Now all productions have the form

$$A_1 A_2 \cdots A_m \longrightarrow B_1 B_2 \cdots B_n \quad \text{with } m \leq n$$

and all $A_i, B_j \in V_N$

3) For each such production $A_1 \cdots A_m \to B_1 \cdots B_n$, introduce a new nonterminal N, and replace the production by the following ones:

$$A_1 A_2 \cdots A_m \to N A_2 \cdots A_m$$
$$N A_2 \cdots A_m \to N B_2 A_3 \cdots A_m$$
$$N B_2 A_3 \cdots A_m \to N B_2 B_3 A_4 \cdots A_m$$
$$\vdots$$
$$N B_2 \cdots B_{m-1} A_m \to N B_2 \cdots B_{m-1} B_m \cdots B_n$$
$$N B_2 \cdots B_n \to B_1 B_2 \cdots B_n$$

(note that, due to the presence of N, these productions will not "interfere" with other ones)

Observe that all such productions are of the form

$$\varphi_1 A \varphi_2 \to \varphi_1 \beta \varphi_2 \quad \text{with } \varphi_1, \varphi_2 \in V^*, \ A \in V_N, \ \beta \in V^+$$

4) For each $a \in V_T$, add the production

$N_a \to a$ (where N_a is the new non-terminal associated to a)

It is not difficult to see that $\mathcal{L}(G_1) = \mathcal{L}(G_c)$ (the proof is by induction on the length of the derivation of a string $w \in \mathcal{L}(G_1)$ (resp. $\mathcal{L}(G_c)$))

↑ END OPTIONAL

Definition: grammar of type 3, or regular, or right linear

Productions have the form $A \to \delta$ with $A \in V_N$

$\delta \in V_T \cup (V_T \circ V_N)$

(i.e., $A \to aB$ or $A \to a$, with $A, B \in V_N$, $a \in V_T$)

A language generated by a grammar of type 3 is called of type 3 or regular

Example: $\{a^n b \mid n \geq 0\}$ is of type 3, since it is generated by the grammar $S \to aS$

$S \to b$

Note: a grammar of type 3 is called linear, because on the right hand side of a production there is at most one non-terminal. It is called right-linear because the non-terminal is on the right of the terminal

↓ OPTIONAL

$\boxed{\text{Exercise: E5.3:}}$ Show that grammars of type 3 generate the class of regular languages that do not contain ε.

Hint: given $G = (V_N, V_T, P, S)$, construct an NFA

$A_G = (V_N \cup \{F\}, V_T, \delta, S, \{F\})$ with

$B \in \delta(A, a)$ iff $A \to aB$ and

$F \in \delta(A, a)$ iff $A \to a$

Show by induction on $|w|$ that $w \in \mathcal{L}(A_G)$ iff $w \in \mathcal{L}(G)$.

Conversely, given an NFA A, construct a grammar G_A by having again nonterminals correspond to states of A.

↑ END OPTIONAL

Note on ε-productions (for grammars of type 1, 2, 3)

As we have defined them, grammars of type 1 (resp. 2, 3) cannot generate the empty string ε.

We could extend the definition by allowing also the generation of ε:

- if the start symbol S does not appear on the right-hand side of productions, we allow also for a production

  $S \to \varepsilon$ (ε-production)

- if the start symbol S appears on the right-hand side of productions, we introduce a new non-terminal S_{new}, make it the new start symbol, add a production $S_{new} \to S$, and allow for $S_{new} \to \varepsilon$.

Hence, an ε-production used just to generate ε is harmless.

Note that, allowing for ε-productions for every non-terminal is not that harmless.

OPTIONAL

$\boxed{\text{Exercise: E5.4:}}$ Show that, for every language L of type 0 there is a grammar of type 1 extended with ε-productions on arbitrary non-terminals that generates L.

<u>Hint</u>: introduce a new nonterminal N_ε that is eliminated through an ε-production $N_\varepsilon \to \varepsilon$, and use N_ε to make the right-hand side of productions as long as the left-hand side.

END OPTIONAL

Context-free grammars (CFGs)

In a CFG, the productions have the form $A \rightarrow \beta$ with $A \in V_N$, $\beta \in V^*$. (note: we allow for ε-productions)

Example: CFG for arithmetic expressions over variable i

$G = (\{E, T, F\}, \{i, +, *, (,), \}, P, E)$, where P is

$E \rightarrow T \mid E + T$ E ... expression

$T \rightarrow F \mid T * F$ T ... term

$F \rightarrow i \mid (E)$ F ... factor

This grammar generates, e.g., $i + i * i$

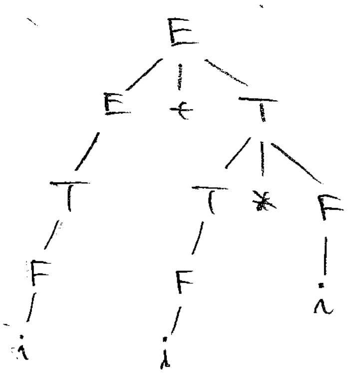
$$\underline{E} \rightarrow \underline{E} + T \rightarrow \underline{T} + T \rightarrow \underline{F} + T \rightarrow i + \underline{T} \rightarrow$$
$$\rightarrow i + \underline{T} * F \rightarrow i + i * \underline{F} \rightarrow i + i * i$$

We can also represent a derivation of a string by a CFG by means of a tree, called parse-tree:

Is a tree whose nodes are labeled by elements of $V \cup \{\varepsilon\}$ satisfying:

1) each interior node is labeled by a non-terminal
2) each leaf is labeled by a non-terminal, a terminal, or ε. If it is labeled by ε, then it is the only child of its parent
3) If an interior node is labeled A, and its children from left to right are labeled $X_1, X_2, \dots, X_k$, then there is a production $A \rightarrow X_1 X_2 \cdots X_k$ in P.

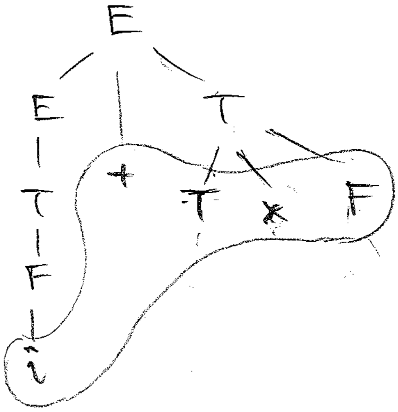
Example: parse tree for $i + i * i$



We call A-tree a subtree of the parse-tree rooted at non-terminal A.

<u>Yield</u> (or frontier) of a tree:
is the sequence of labels of the leaves from left to right.

Example:



<u>Theorem</u>: $\alpha \in V^+$ is the yield of an A-tree $\Longleftrightarrow A \stackrel{*}{\Longrightarrow} \alpha$

Proof: by induction on the height of the tree
(see textbook)

8/1/2009

<u>Note</u>: a parse tree does not specify a unique way to derive α from A. (the order in which non-terminals are expanded is not specified).

The parse tree specifies, however, which rule is applied for each non-terminal.

Specific <u>derivation</u> orders:

- leftmost derivation: obtained by traversing the tree depth-first, by first going to the left subtree, and then to the right one.
  E.g. $E \stackrel{lm}{\Longrightarrow} \underline{E}+T \stackrel{lm}{\Longrightarrow} \underline{T}+T \stackrel{lm}{\Longrightarrow} \underline{F}+T \stackrel{lm}{\Longrightarrow} i+\underline{T} \Longrightarrow \dots$
- rightmost derivation: defined similarly: $\underline{E} \stackrel{rm}{\Longrightarrow} E+\underline{T} \Longrightarrow E+T*\underline{F} \dots$

<u>Theorem</u>: the following are all equivalent statements for a CFG $G=(V,T,P,S)$ and a string $w \in T^*$

1) $w \in \mathcal{L}(G)$ (or $S \overset{*}{\Rightarrow} w$)
2) $S \underset{lm}{\overset{*}{\Rightarrow}} w$
3) $S \underset{rm}{\overset{*}{\Rightarrow}} w$
4) There exists an S-tree with yield w.

<u>Proof</u>: the equivalence of (1) and (4) follows from the previous theorem. The other equivalences are obvious.

Thus, we could always use lm-derivation as a canonical way to derive any $w \in \mathcal{L}(G)$, i.e. as a canonical way to interpret a parse tree for w.

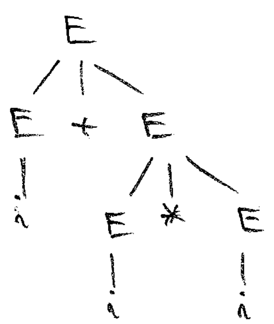
<u>Ambiguous grammars</u>:

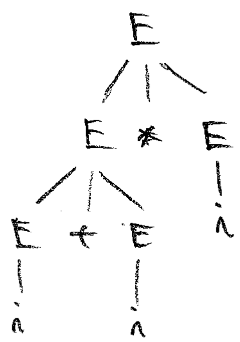
a $w \in \mathcal{L}(G)$ could have two distinct parse trees, and hence two distinct lm-derivations

Example: another grammar for arithmetic expressions

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

$w = i+i*i$





These parse trees correspond to two different lm-derivations, and also to two ways of interpreting w.

<u>Definition</u>: a CFG G is <u>ambiguous</u> if for some $w \in \mathcal{L}(G)$ there exist two distinct parse trees.

Ambiguity has to be avoided in compilers, since it corresponds to different ways of interpreting a string.

Sometimes grammars can be redesigned to remove ambiguity. (e.g., for arithmetic expressions)

This is not always possible:

<u>Definition</u>: a CF language is (inherently) <u>ambiguous</u> if all its grammars are ambiguous

Example: $L = \{a^n b^n c^m d^m \mid n, m \geq 1\} \cup$
$\{a^n b^m c^m d^n \mid n, m \geq 1\}$

L is CF (show for exercise)

Consider strings of the form $a^k b^k c^k d^k$.
We cannot tell whether they come from first or second types of strings in L, and any CFG must allow for both possibilities.

Properties of context-free languages

We will study

1) Normal forms for CFGs (useful for proving properties of CFLs)
2) Expressive power ⇒ pumping lemma for CFLs
3) Closure and decision properties

Normal forms for CFGs

We look at how to simplify CFGs, while preserving the generated language.

- gain efficiency in parsing
- simplify proving properties

1) Eliminate useless symbols:

We say that $X \in V$ is useful if

$$S \Rightarrow^* \alpha X \beta \Rightarrow^* w \qquad \text{with } w \in V_T^*, \ \alpha, \beta \in V^*$$

Thus, a symbol is useless (not useful) if it does not participate in any derivation of strings of the language.
⇒ can be eliminated

Definition: $X \in V$ is generating if $X \Rightarrow^* w$, for $w \in V_T^*$
$X \in V$ is reachable if $S \Rightarrow^* \alpha X \beta$, for $\alpha, \beta \in V^*$

Hence, X is useful, if it is both generating and reachable.

We identify useless symbols by

1) eliminating non-generating symbols and all their productions
2) — " — unreachable — " —

Note: it is important to do these two steps in the above order

Example: $\begin{cases} S \to AB \mid b \\ A \to a \end{cases}$ Let us consider what happens if we do first (2) and then (1)

- we eliminate unreachable symbols: all are reachable
- — " — non-generating — " — :

 we eliminate B and $S \to AB$

$\Rightarrow$ we obtain: $S \to b$
$A \to a$

But, if we do it in right order:

1) Eliminate non-generating symbols: B and $S \to AB$
2) — " — unreachable — " — : A and $A \to a$

$\Rightarrow$ We obtain: $S \to b$

1) Eliminating non-generating symbols:

Recursive algorithm to construct the set of generating symbols:

basis: mark all terminals as generating

recursive step: for each production $A \to X_1 \cdots X_k$
if all of $X_1, \dots, X_k$ are marked as generating
then mark A as generating

terminate: when no new generating symbol is found

Example: G_1:
$$\begin{cases} S \to AB \mid AC \mid CD \\ A \to BB \\ B \to AC \mid ab \\ C \to Ca \mid CC \\ D \to Bc \mid b \mid d \end{cases}$$

$\{a, b, d\}$

$\{ \text{—''—}, B, D\}$

$\{ \text{—''—}, A\}$

$\{ \text{—''—}, S\}$ $\Rightarrow$ C is non-generating

$\Rightarrow$ Remove C and all productions involving C

2) Eliminating unreachable symbols

Recursive algorithm to construct the set of reachable symbols:

basis: mark S as reachable

recursive step: for each production $A \to X_1 \cdots X_k$

if A is marked as reachable

then mark $X_1, \dots, X_k$ as reachable

terminate when no new reachable symbol is found

Example: G_2:
$$\begin{cases} S \to AB \\ A \to BB \\ B \to ab \\ D \to b \mid d \end{cases}$$

$\{S\}$

$\{S, A, B\}$

$\{S, A, B, a, b\}$ $\Rightarrow$ D, d are unreachable

$\Rightarrow$ Remove D, d and all productions involving them

2) Eliminate ε-productions

ε-production: $A \rightarrow \varepsilon$ slows down parsing

Definition: $X \in V_N$ is nullable if $X \Rightarrow^* \varepsilon$

We first need to find all nullable symbols:

Recursive algorithm to construct the set of nullable symbols:

basis: if P contains $A \rightarrow \varepsilon$, then mark A as nullable

inductive step: for each production $A \rightarrow X_1 \cdots X_n$
if all of $X_1, \ldots, X_n$ are marked as nullable
then mark A as nullable

terminate when no new nullable symbol is found

15/1/2008

Example: G_1: $\begin{cases} S \rightarrow ABC \mid BCB \\ A \rightarrow aB \mid a \\ B \rightarrow CC \mid b \\ C \rightarrow S \mid \varepsilon \end{cases}$

$\{C\}$
$\{C, B\}$
$\{C, B, S\}$

Knowing the nullable symbols allows us to compensate for the elimination of ε-transitions.

Example: in G_1, since B and C are nullable, we can derive

$S \Rightarrow^* BCB$, $S \Rightarrow^* CB$, $S \Rightarrow^* BC$, $S \Rightarrow^* BB$,
$S \Rightarrow^* C$, $S \Rightarrow^* B$, $S \Rightarrow^* \varepsilon$

Hence, if we eliminate $C \rightarrow \varepsilon$, we have to add direct productions for the above derivations.

Algorithm to eliminate ε-productions

1) Identify all nullable symbols
2) Replace each production $A \to X_1 \cdots X_k$
   by the set of all productions of the form $A \to \alpha_1 \cdots \alpha_k$
   where $\alpha_i = X_i$, if X_i is not nullable
   $\alpha_i = X_i$ or ε, if X_i is nullable
3) If the resulting grammar contains $S \to \varepsilon$, introduce a new start symbol S' and add the productions $S' \to S \mid \varepsilon$
4) Remove all ε-productions, except possibly the one for S'.

Example: by applying steps (1) and (2) to G_1, we get

$$
\begin{cases}
S \to ABC \mid AB \mid AC \mid A \mid \\
\qquad BCB \mid BC \mid BB \mid CB \mid B \mid C \mid \varepsilon \\
A \to aB \mid a \\
B \to CC \mid C \mid \varepsilon \mid b \\
C \to S \mid \varepsilon
\end{cases}
$$

Since we have $S \to \varepsilon$, we add
$S' \to S \mid \varepsilon$
and remove the remaining ε-productions.

3) Eliminate unit productions

Unit-production: $A \to B$ slows down parsing

Algorithm to eliminate unit-productions

1) Remove ε-productions
2) For all $A, B \in V_N$
   if $A \stackrel{*}{\Rightarrow} B$ and $B \to \alpha$ is not unit
   then add $A \to \alpha$
3) Eliminate all unit-productions

How do we determine whether $A \Rightarrow^* B$ holds?

Since we have no ε-productions, we have that

$A \Rightarrow^* B$ if and only if

$A \Rightarrow B_1 \Rightarrow B_2 \Rightarrow \cdots \Rightarrow B_{k-1} \Rightarrow B_k \Rightarrow B$

where all B_i's are pairwise distinct. Hence $k \leq |V_N|$.

(if we had a sequence where two B_i's are the same, we could eliminate all steps inbetween, and get a new sequence where all B_i's are pairwise distinct)

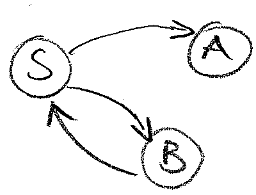
Each single derivation step $B_i \Rightarrow B_{i+1}$ must correspond to a unit production $B_i \rightarrow B_{i+1}$ of G.

Hence, we can detect whether $A \Rightarrow^* B$ by checking whether B is reachable from A in the graph of the unit productions:

- nodes: non-terminals
- edges: one edge $A \rightarrow B$ for each unit prod. $A \rightarrow B$

Example: G_1: $\begin{cases} S \rightarrow A \mid B \\ A \rightarrow Sa \mid a \\ B \rightarrow S \mid b \end{cases}$

Graph of unit productions



reachability: $S \Rightarrow^* A$ $\quad S \Rightarrow^* B$

$B \Rightarrow^* S$ $\quad B \Rightarrow^* A$

We get: $\begin{cases} S \rightarrow A \mid B \mid Sa \mid a \mid S \mid b \\ A \rightarrow Sa \mid a \\ B \rightarrow S \mid b \mid A \mid B \mid Sa \mid a \end{cases}$

Removing unit productions, we get $\begin{cases} S \rightarrow Sa \mid a \mid b \\ A \rightarrow Sa \mid a \\ B \rightarrow Sa \mid a \mid b \end{cases}$

Note: A and B have become unreachable

We have seen: removal of : useless symbols, ε-prod, unit-prod.

Does the order of the steps matter?

Observation:

- removing useless : does not add productions at all (and therefore not ε-prod. or unit-prod)
- removing ε-prod: could add unit-prod
- removing unit-prod: - needs removing ε-prod first
  - could create useless symbols
  - cannot create ε-prod.

$\Rightarrow$ The right order for removal is

1) ε-productions
2) unit-productions
3) useless symbols : first non-generating then unreachable

<u>Chomsky</u> <u>Normal Form</u>

<u>Definition</u>: a CFG G is in Chomsky Normal (CNF) if all its productions are of the form

$$A \to a \qquad \text{with} \qquad a \in V_T$$
$$A \to BC \qquad\qquad A, B, C \in V_N$$

Given a CFG G, we can always construct a CFG G_C that is in CNF and such that $\mathcal{L}(G_C) = \mathcal{L}(G) \setminus \{\varepsilon\}$.

Note: since a CFG in CNF cannot generate ε, if G generates ε, then we cannot have that $\mathcal{L}(G_C) = \mathcal{L}(G)$. However, apart from ε, the two languages are equal.

Starting from G, we construct G_C in several steps:

1) Eliminate ε-productions (without introducing the new start symbol S' with $S' \to S \mid \varepsilon$)
2) Eliminate unit-productions

$\Rightarrow$ All productions are of the form

$A \to a$

$A \to X_1 \cdots X_k$ with $k \geq 2$

with $A \in V_N$, $a \in V_T$, $X_1, \dots, X_k \in V$

3) Remove non-generating symbols
4) Remove unreachable symbols
5) Remove "mixed bodies"

for each $a \in V_T$, add a new nonterminal V_a and production $V_a \to a$

in each production $A \to X_1 \cdots X_k$, replace a with V_a

$\Rightarrow$ All productions are of the form

$A \to a$

$A \to A_1 \cdots A_k$ $(k \geq 2)$

with $a \in V_T$, $A, A_1, \dots, A_k \in V_N$

6) "Factor" long productions

for each $A \to A_1 \cdots A_k$ with $k \geq 3$

- add new nonterminals $B_1, \dots, B_{k-2}$
- replace $A \to A_1 \cdots A_k$

with

$$
\begin{aligned}
A &\to A_1 B_1 \\
B_1 &\to A_2 B_2 \\
&\vdots \\
B_{k-2} &\to A_{k-1} A_k
\end{aligned}
$$

The grammar we get is in CNF by construction.

It is easy to show that the language is preserved, apart possibly for the empty string ε, which cannot be generated by a grammar in CNF.

Example: G_1
$$\begin{cases} S \to ABB \mid ab \\ A \to Ba \mid ba \\ B \to aAbB \end{cases}$$

Steps 1-4 nothing to do

Step 5:
$$\begin{cases} V_a \to a \\ V_b \to b \\ S \to ABB \mid V_a V_b \\ A \to B V_a \mid V_b V_a \\ B \to V_a A V_b B \end{cases}$$

Step 6:
$$V_a \to a$$
$$V_b \to b$$
$$\begin{cases} S \to A B_1 \mid V_a V_b \\ B_1 \to BB \end{cases}$$
$$A \to B V_a \mid V_b V_a$$
$$\begin{cases} B \to V_a C_1 \\ C_1 \to A C_2 \\ C_2 \to V_b B \end{cases}$$

Note: If the original grammar generated ε, and we want that the grammar in CNF also generates ε, we can execute step 1 by introducing the new start symbol S' and the productions: $S' \to S \mid \varepsilon$

Step 2 will then replace the unit production $S' \to S$ by other productions, but none of the transformations 2-5 will introduce a production where S' is on the right side.

Therefore, in the end, we will have a grammar in CNF, apart for the production $S' \to \varepsilon$.