# Languages and Machines (Module 7 TCS+IAM)

## L&M 5: Chomsky Normal Form and CYK parsing

## Ch 4:1-6

Sebastiaan Joosten

Formal Methods and Tools, University of Twente

Lecture 5



## Contents





## Summary

Context-free Grammars (CFGs)

- Distinction *variables* ($V$ with $S \in V$) and *symbols* ($\Sigma$)
- Variables are similar to states of an NFA
- Set of *rules* of the form $A \to w$;
  $A \in V, w \in (V \cup \Sigma)*$

Derivations and derivation trees

- One-to-one-relationship *leftmost* derivations and derivation trees

Language of a CFG

- Set of derivable sentences: $S \Rightarrow^* w$, $w \in \Sigma^*$

Special case: regular grammars

- Translation from and to NFAs
- Regular languages subset of context-free languages

Ambiguity of a grammar

- Sentence(s) with more than one derivation tree



## Contents

## Syntax of programming languages

- Completely modeled by non-ambiguous context-free grammars
- Example: CFG for syntax of expressions

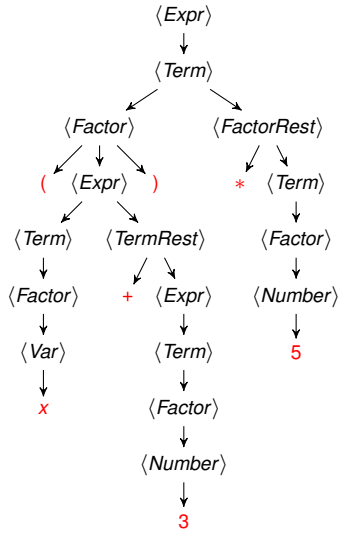
$$\begin{aligned}
\langle Expr \rangle &\rightarrow \langle Term \rangle \ \langle TermRest \rangle_{opt} \\
\langle TermRest \rangle &\rightarrow + \langle Expr \rangle \mid - \langle Expr \rangle \\
\langle Term \rangle &\rightarrow \langle Factor \rangle \ \langle FactorRest \rangle_{opt} \\
\langle FactorRest \rangle &\rightarrow * \langle Term \rangle \mid / \langle Term \rangle \\
\langle Factor \rangle &\rightarrow \langle Var \rangle \mid \langle Number \rangle \mid ( \langle Expr \rangle )
\end{aligned}$$

- *Backus-Naur-Vorm* (BNF): notation-agreements
  - Non-terminals in angular brackets, for example $\langle Number \rangle$
  - Subscript $_{opt}$ (optional) for "$\lambda$-choice:"
    - $\langle A \rangle \rightarrow \langle B \rangle_{opt}$ short for $\langle A \rangle \rightarrow \langle B \rangle | \lambda$
    - $\langle A \rangle \rightarrow \langle B \rangle_{opt} \langle C \rangle$ short for $\langle A \rangle \rightarrow \langle B \rangle \langle C \rangle | \langle C \rangle$
  - Many variations and extensions exist
- See book (appendix IV) for complete BNF grammar for Java programs



## Syntax of programming languages: example

Parse (i.e., find a derivation for): $(x + 3) * 5$



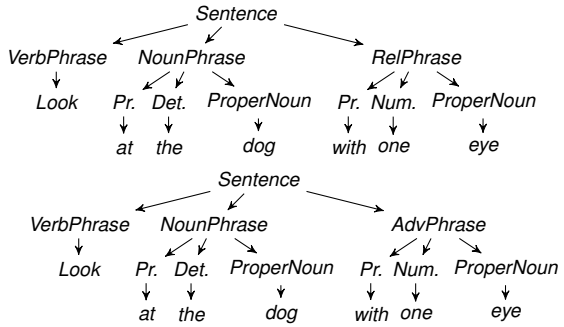



## Example of ambiguity

- Sentence: *Look at the dog with one eye*
- Grammar:

$$\begin{aligned}
Sentence &\rightarrow VerbPhrase\ NounPhrase\ RelPhrase \mid \\
&\quad\ VerbPhrase\ NounPhrase\ AdvPhrase \\
NounPhrase &\rightarrow Pr.\ Det.\ PropoerNoun \\
&\ldots
\end{aligned}$$

- Derivation trees:




- Different derivations result in different *interpretations*



## Definition of (un)ambiguity

**Definition**

A CFG is *ambiguous* if a word exists in $L(G)$ with two distinct derivation trees.

**Remarks**

- Ambiguity results from the grammar, not from the language
  The following language has an ambiguous ánd an unambiguous grammar:
  $G_1: S \rightarrow aS \mid Sa \mid a \qquad G_2: S \rightarrow aS \mid a$
  $L(G_1) = L(G_2) = a^+$ holds; but $G_1$ is ambiguous, $G_2$ is not
- Fact: There are also languages with *only* ambiguous grammars
  Example: $L = \{a^m b^m c^n \mid m, n \geq 0\} \cup \{a^m b^n c^n \mid m, n \geq 0\}$
- Unambiguous grammars exist for regular languages
- Programming languages have unambiguous grammars

## The principle of parsing

**Definition**

- A *parsing* of a sentence is a derivation tree for that sentence
- A *parsing algorithm* for a grammar is an algorithm that produces a parsing for each sentence

**Naive parsing algorithm: Search and backtrack**

1. If the sentential form consists of only $S$, we are done
2. Search right side of a rule that matches part of the sentence

   Success?
   - Replace the right side by the corresponding left side
   - Repeat step 1 (next phase)

   Failure?
   - Go back to the previous phase
   - Try the next possibility (backtracking)



## Example of naive parsing

- Grammar: $S \to a\,S\,b \mid a\,A \qquad A \to a\,A \mid a$
- Language: $\{a^m b^n \mid n \geq 0, m > n+1\}$
- Parsing:

$$
\begin{array}{lllllll}
aaab & \Leftarrow & Aaab & \Leftarrow & AAab & \Leftarrow & AAAb \not\Leftarrow \\
 & & & \Leftarrow & AaAb & \Leftarrow & AAAb \not\Leftarrow \\
 & & & & & \Leftarrow & AAb \not\Leftarrow \\
 & & & & & \Leftarrow & ASb \not\Leftarrow \\
 & \Leftarrow & aAab & \Leftarrow & AAab & \not\Leftarrow & \\
 & & & \Leftarrow & aAAb & \Leftarrow & \cdots \not\Leftarrow \\
 & \Leftarrow & aaAb & \Leftarrow & AaAb & \not\Leftarrow & \\
 & & & \Leftarrow & aAAb & \not\Leftarrow & \\
 & & & \Leftarrow & aAb & \Leftarrow & AAb \not\Leftarrow \\
 & & & & & \Leftarrow & Ab \not\Leftarrow \\
 & & & & & \Leftarrow & Sb \not\Leftarrow \\
 & & & \Leftarrow & aSb & \Leftarrow & S \checkmark
\end{array}
$$

- The complexity of this algorithm is not acceptable! Why?
  - It becomes even worse if there is also $\lambda$ on the right side



## Contents





## Normal forms of grammars

Known: A language can have several grammars
- Some are more suitable for parsing than others

Normal form: restriction on the right side of the rules
- Example: never $a\,A\,b$ but always $A\,B$; never $\lambda$
- Simpler to analyze
- More efficient parsing algorithms

Automatic transformation to normal form:
- Through *step-by-step transformation* of the grammar

## Examples of these grammar transformations

Take $G:\quad S \to aS \mid Ab \qquad A \to cSb \mid \lambda$

- Adding a new start symbol $S'$ results in:

$$G_1:\quad S' \to S \quad S \to aS \mid Ab \qquad A \to cSb \mid \lambda$$

(rationale: the new start symbol will not occur recursively)

- Inlining the $S$-rule in $A$ results in:

$$G_3:\quad S \to aS \mid Ab \qquad A \to caSb \mid cAbb \mid \lambda$$



## Simple transformations of grammars

Let $G = \langle V, \Sigma, P, S\rangle$ be a given grammar.

**Lemma: Add start symbol**

Take $S' \notin V$, a new symbol. Let
$G_1 := \langle V \cup \{S'\}, \Sigma, P \cup \{S' \to S\}, S'\rangle$.
Then $\mathcal{L}(G_1) = \mathcal{L}(G)$.

**Lemma: "Inlining" rules**

Suppose that $A \to uBv$ and $B \to w_1 \mid w_2 \mid \ldots \mid w_n$ in $P$.
Let $G_3 := \langle V, \Sigma, P_3, S\rangle$ with

$$P_3 = (P \setminus \{A \to uBv\}) \cup \{A \to uw_1v \mid uw_2v \mid \cdots \mid uw_nv\}\ .$$

Then $\mathcal{L}(G_3) = \mathcal{L}(G)$.



## Chomsky normal form (CNF)

**Definition**

$G = \langle V, \Sigma, P, S\rangle$ is in *Chomsky normal form* if all rules are of one of the following forms:

- $A \to BC$, ór
- $A \to a$, ór
- $S \to \lambda$

met $B, C \in V \setminus \{S\}$ (so $S$ is not recursive)

For example for: $S \to aSb \mid aA$
$A \to aA \mid a$

this is a Chomsky normal form: $S \to S_1S_2 \mid S_1A$
$S_1 \to a$
$S_2 \to S_3S_4$
$S_3 \to S_1S_2 \mid S_1A$
$S_4 \to b$
$A \to S_1A \mid a$



## Transformation to CNF

- Advantages Chomsky normal form:
  - Derivations are non-contracting (words do not get shorter)
  - Derivation of $w$ at most $2n-1$ steps ($n = |w|$)
  - Derivation tree is binary, depth at most $n$
  - Efficient parsing algorithm
- Necessary steps to achieve CNF:
  1. Eliminate recursion in start symbol
  2. Eliminate $\lambda$-rules (except for $S$)
  3. Eliminate "chain-rules" $A \to B$
  4. Finalize rules $A \to w$ with $|w| \geq 2$ and $w \neq BC$
- Clean up:
  5. Eliminate non-terminating variables
  6. Eliminate unreachable variables

## Non-contracting grammar: eliminate null-rules

- Example
  - Grammar with null-rule:

$$G_1: \quad S \to SaB \mid aB \qquad B \to bB \mid \lambda$$

  - Equivalent grammar without null-rule (non-contracting):

$$G_2: \quad S \to SaB \mid Sa \mid aB \mid a \qquad B \to bB \mid b$$

- Transformation-idea
  1. Assume that the start symbol is non-recursive
  2. Collect all variables $L$ with a derivation to $\lambda$;
  3. For all rules $A \to w$:
     - If $w$ is equal to $w_1 A_1 w_2 A_2 w_3 \cdots w_k A_k w_{k+1}$ (with $A_i \in L$)
     - Add a rule $A \to w_1 w_2 \cdots w_k w_{k+1}$
  4. Remove all null-rules except $S \to \lambda$



## Step 1: Collect variables with $\lambda$-derivation

**Algorithm: backwards starting from null-rules**

$Null := \{A \in V \mid (A \to \lambda) \in P\}$
$Prev := \emptyset$
**while** $(Prev \neq Null)$**:**
  $Prev := Null$
  $Null := Null \cup \{A \in V \mid \exists w \in Null^* : (A \to w) \in P\}$

- Example grammar:

$S \to A\,C\,A$ $\qquad$ $B \to b\,B \mid b$
$A \to a\,A\,a \mid B \mid C$ $\qquad$ $C \to c\,C \mid \lambda$

- Iterations algorithm:

| *Round* | *Null* | *Prev* |
|---|---|---|
| 0 | $\{C\}$ | $\emptyset$ |
| 1 | $\{A, C\}$ | $\{C\}$ |
| 2 | $\{S, A, C\}$ | $\{A, C\}$ |
| 3 | $\{S, A, C\}$ | $\{S, A, C\}$ |



## Step 2 and 3: Eliminate variables with $\lambda$-derivation

- Example grammar:

$S \to A\,C\,A$
$A \to a\,A\,a \mid B \mid C$
$B \to b\,B \mid b$
$C \to c\,C \mid \lambda$

- Step 2: Remove all combinations of variables $\{S, A, C\}$:

$S \to \lambda \mid A \mid C \mid A\,C \mid C\,A \mid A\,A \mid A\,C\,A$
$A \to a\,a \mid a\,A\,a \mid B \mid \lambda \mid C$
$B \to b\,B \mid b$
$C \to c \mid c\,C \mid \lambda$

- Step 3: Remove rules $A \to \lambda$ (except $S \to \lambda$)



## Eliminate chain-rules

- Example
  - Grammar with chain-rules:
    $G_1: \quad S \to A \mid aB \quad A \to Ab \mid B \quad B \to AA \mid b$
  - Equivalent grammar without chain-rule:
    $G_2: \quad S \to Ab \mid AA \mid b \mid aB \quad A \to Ab \mid AA \mid b \quad B \to AA \mid b$
- Transformation-idea
  1. Assume that the grammar is non-contracting
  2. Collect the reachable variables $Chain(A) = \{B \in V \mid A \overset{*}{\Rightarrow} B\}$
  3. Create rule $A \to w$ for all $B \in Chain(A)$ and $B \to w$ $(w \notin V)$

## Step 1: compute chain-variables

**Algorithm: forwards from chain-rules**

$Chain(A) := \{A\}$
$Prev := \emptyset$
**while** $(Chain(A) \neq Prev)$**:**
  $New := Chain(A) \setminus Prev$
  $Prev := Chain(A)$
  $Chain(A) := Chain(A) \cup \{C \in V \mid \exists B \in New : (B \to C) \in P\}$

- Example grammar:
  $S \to \lambda \mid A \mid C \mid A\,C \mid C\,A \mid A\,A \mid A\,C\,A$ $\quad B \to b\,B \mid b$
  $A \to a\,a \mid a\,A\,a \mid B \mid C$ $\quad C \to c \mid c\,C$
- Result of algorithm: $Chain(S) = \{S, A, C, B\}$
  $Chain(A) = \{A, B, C\}$
  $Chain(B) = \{B\}$
  $Chain(C) = \{C\}$



## Step 2: Create rules without chain variables

- Grammar:

| | *Chain* |
|---|---|
| $S \to \lambda \mid A \mid C \mid A\,C \mid C\,A \mid A\,A \mid A\,C\,A$ | $\{S, A, B, C\}$ |
| $A \to a\,a \mid a\,A\,a \mid B \mid C$ | $\{A, B, C\}$ |
| $B \to b\,B \mid b$ | $\{B\}$ |
| $C \to c \mid c\,C$ | $\{C\}$ |

- Step 2: Rule $A \to w$ for all $B \in Chain(A)$ and $B \to w$ $(w \notin V)$

$S \to \lambda \mid A\,C \mid C\,A \mid A\,A \mid A\,C\,A \mid a\,a \mid a\,A\,a \mid b\,B \mid b \mid c \mid c\,C$
$A \to a\,a \mid a\,A\,a \mid b\,B \mid b \mid c \mid c\,C$
$B \to b\,B \mid b$
$C \to c \mid c\,C$



## Finalizing right sides

- Example
  - Unfinalized grammar: $S \to a\,A\,b$
    $A \to a\,A\,b \mid a\,b$
  - Chomsky normal form: $S \to D_1\,C_1$
    $A \to D_1\,C_1 \mid D_1\,D_2$
    $C_1 \to A\,D_2$
    $D_1 \to a$
    $D_2 \to b$
- Transformation-idea: Replace each rule $A \to w$ with $|w| \geq 2$:
  - Suppose that $w = a_1 a_2 \cdots a_n$ with $a_i \in V \cup \Sigma$ for all $i \in [1, n]$, and
    - For $a_i \in V$: name $D_i = a_i$ ;
    - For $a_i \in \Sigma$: choose a new $D_i$ and define $D_i \to a_i$
  - Name $C_1 = A$ and choose new variables $C_2, \ldots, C_{n-1}$;
  - Define $C_i \to D_i C_{i+1}$ for $i \in [1, n-2]$, and $C_{n-1} \to D_{n-1} D_n$



## Construction of Chomsky normal form

**Theorem**

For every grammar $G$ a grammar $G'$ exists in Chomsky normal form such that $L(G') = L(G)$

**Proof**

Construct $G'$ in steps:

1. Transform $G$ into $G_1$ by adding a new start symbol
2. Transform $G_1$ into $G_2$ by eliminating null-rules
3. Transform $G_2$ into $G_3$ by eliminating chain-rules
4. Transform $G_3$ into $G'$ by finalizing right sides

Every transformation results in an equivalent grammar.

## Useless variables

- Which variables can *not* be used in a derivation?

$$
\begin{aligned}
S &\to AC \mid BS \mid B \\
A &\to aA \mid aF \\
B &\to CF \mid b \\
C &\to cC \mid D \\
D &\to aD \mid BD \mid C \\
E &\to aA \mid BSA \\
F &\to bD \mid b
\end{aligned}
$$

- $C$ and $D$ do not *terminate*; so
$$
\begin{aligned}
S &\to BS \mid B \\
A &\to aA \mid aF \\
B &\to b \\
E &\to aA \mid BSA \\
F &\to b
\end{aligned}
$$
- $A$, $E$ and $F$ not *reachable*; so
$$
\begin{aligned}
S &\to BS \mid B \\
B &\to b
\end{aligned}
$$



## Detection of useless variables

**Algorithm: detection of termination (backwards)**

$term := \{A \in V \mid \exists w \in \Sigma^* : (A \to w) \in P\}$
$prev := \emptyset$
**while** ($term \neq prev$)**:**
    $prev := term$
    $term := term \cup \{A \in V \mid \exists w \in (term \cup \Sigma)^* : (A \to w) \in P\}$

**Algorithm: detecting of reachability (forwards)**

$reach := \{S\}$
$new := \{S\}$
**while** ($new \neq \emptyset$)**:**
    $prev := reach$
    $reach := reach \cup \{C \in V \mid \exists B \in new : (B \to w) \in P, C \in w\}$
    $new := reach \setminus prev$

First clean up $V \setminus term$, then $V' \setminus reach$ (why?)



## Contents





## Cocke-Younger-Kasami (CYK) parsing

Starting point: Grammar in CNF; word $w$ to be parsed
- Suppose that $w = a_1 a_2 \cdots a_n$ (where $n \geq 0$ and $a_i \in V \cup \Sigma$)
- Notation: sub-word $w_{ij} = a_i \cdots a_j$ (with $1 \leq i \leq j \leq n$)

Basic idea: Dynamic Programming:
Consider increasingly longer sub-words
- $X_{ij} \subseteq V$: set of variables with a derivation to $w_{ij}$
- These can be built step-by-step (iterate over $j - i$)

Algorithm: Bottom-up

**for** $len$ **in range**$(1, n)$**:**
    **for** $i$ **in range**$(1, n - len + 1)$**:**
        $j := i + len - 1$
        **if** $len == 1$**:**
            $X_{ij} := \{A \mid (A \to a_{ii}) \in P\}$
        **else:**
            $X_{ij} := \{A \mid (A \to BC) \in P, B \in X_{ik}, C \in X_{k+1,j}\}$

## Example CYK parsing

- Grammar of $\{a^i b^i \mid i > 0\}$ (in Chomsky normal form):

  $S \to A\,T \mid A\,B$ $\quad$ $A \to a$

  $T \to X\,B$ $\quad$ $B \to b$

  $X \to A\,T \mid A\,B$

- Parsing of word $w = a\,a\,a\,b\,b\,b$:

$X_{ij}$-matrix:

| | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|
| 1 | $\{A\}$ | $\emptyset$ | $\emptyset$ | $\emptyset$ | $\emptyset$ | $\{S, X\}$ |
| 2 | | $\{A\}$ | $\emptyset$ | $\emptyset$ | $\{S, X\}$ | $\{T\}$ |
| 3 | | | $\{A\}$ | $\{S, X\}$ | $\{T\}$ | $\emptyset$ |
| 4 | | | | $\{B\}$ | $\emptyset$ | $\emptyset$ |
| 5 | | | | | $\{B\}$ | $\emptyset$ |
| 6 | | | | | | $\{B\}$ |



## Derivation trees in the CYK-algorithm

- The CYK-algorithm decides whether a word is in the langage
- This is *not* yet the same as parsing
- The derivation tree is implicitly in the $X_{ij}$-matrix: