CS311: Computational Theory

Lecture 6: CONTEXT-FREE GRAMMARS – Ch 2

Dr. Manal Helal, Spring 2014.

http://moodle.manalhelal.com

Lecture Learning Objectives

1. *Understand Context-Free Grammars and their applications.*

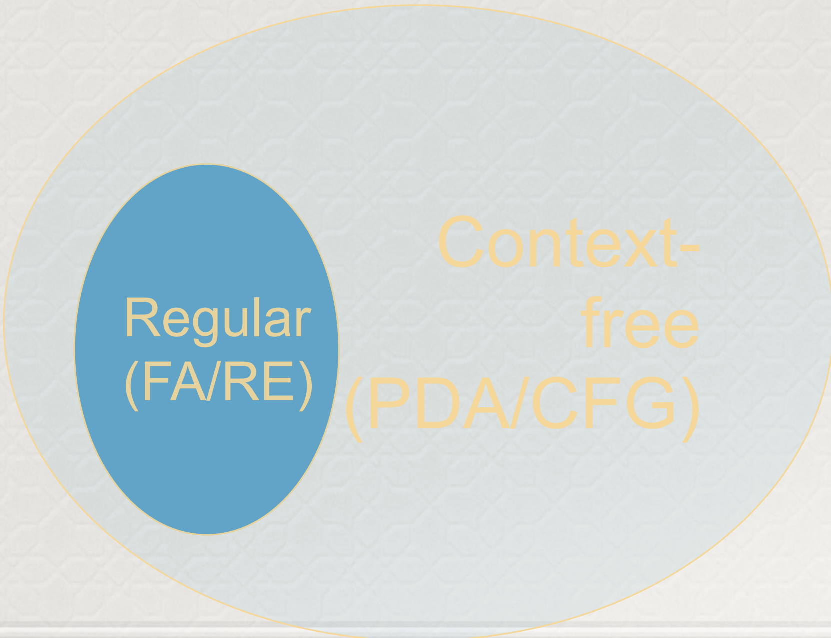
Not all languages are regular

- *So what happens to the languages which are not regular?*

- *Can we still come up with a language recognizer?*
  - *i.e., something that will accept (or reject) strings that belong (or do not belong) to the language?*

Context-Free Languages

- *A language class larger than the class of regular languages*
- *Supports natural, recursive notation called "context-free grammar"*
- *Applications:*
  - *Parse trees, compilers*
  - *XML*



An Example

- *A palindrome is a word that reads identical from both ends*
  - *E.g., madam, redivider, malayalam, 010010010*
- *Let L = { w | w is a binary palindrome}*
- *Is L regular?*
  - *No.*
  - *Proof:*
    - *Let w=0^N10^N* *(assuming N to be the p/l constant)*
    - *By Pumping lemma, w can be rewritten as xyz, such that xy^kz is also L (for any k≥0)*
    - *But |xy|≤N and y≠ε*
    - *==> $y=0^+$*
    - *==> xy^kz will NOT be in L for k=0*
    - *==> Contradiction*

But the language of palindromes…

is a CFL, because it supports recursive substitution (in the form of a CFG)

- This is because we can construct a "grammar" like this:

Productions

1. A ==> ε
2. A ==> 0
3. A ==> 1
4. A ==> 0A0
5. A ==> 1A1

Terminal

Variable or non-terminal

Same as:
A => 0A0 | 1A1 | 0 | 1 | ε

How does this grammar work?

How does the CFG for palindromes work?

An input string belongs to the language (i.e., accepted) iff it can be generated by the CFG

- *<u>Example:</u> w=01110*
- *G can generate w as follows:*

G:
A => 0A0 | 1A1 | 0 | 1 | ε

1. *A => 0A0*
2. *=> 01A10*
3. *=> 01110*

<u>Generating a string from a grammar:</u>

1. Pick and choose a sequence of productions that would allow us to generate the string.
2. At every step, substitute one variable with one of its productions.

Context-Free Grammar: Definition

- *A context-free grammar G=(V,∑,R,S), where:*
  - *V: set of variables or non-terminals*
  - *T: set of terminals (= alphabet U {ε})*
  - *P: set of productions, each of which is of the form*
    V ==> α_1 | α_2 | …
    - *Where each α_i is an arbitrary string of variables and terminals*
  - *S ==> start variable*

CFG for the language of binary palindromes:

G=({A},{0,1},P,A)

P: A ==> 0 A 0 | 1 A 1 | 0 | 1 | ε

More examples

- *Parenthesis matching in code*
- *Syntax checking*
- *In scenarios where there is a general need for:*
  - *Matching a symbol with another symbol, or*
  - *Matching a count of one symbol with that of another symbol, or*
  - *Recursively substituting one symbol with a string of other symbols*

Example #2

- *Language of balanced paranthesis*

  e.g., ()((((())))((()))….
- *CFG?*

G:
S => (S) | SS | ε

How would you "interpret" the string "(((()))()())" using this grammar?

Example #3

- *A grammar for L = $\{0^m 1^n \mid m \geq n\}$*

- *CFG?*

G:
S => 0S1 | A
A => 0A | ε

How would you interpret the string "00000111" using this grammar?

Example #4

*A program containing **if-then(-else)** statements*

* **if** Condition **then** Statement **else** Statement*

* (Or)*

* **if** Condition **then** Statement*

CFG?

Example #5

⟨SENTENCE⟩ → ⟨NOUN-PHRASE⟩⟨VERB-PHRASE⟩
⟨NOUN-PHRASE⟩ → ⟨CMPLX-NOUN⟩ | ⟨CMPLX-NOUN⟩⟨PREP-PHRASE⟩
⟨VERB-PHRASE⟩ → ⟨CMPLX-VERB⟩ | ⟨CMPLX-VERB⟩⟨PREP-PHRASE⟩
⟨PREP-PHRASE⟩ → ⟨PREP⟩⟨CMPLX-NOUN⟩
⟨CMPLX-NOUN⟩ → ⟨ARTICLE⟩⟨NOUN⟩
⟨CMPLX-VERB⟩ → ⟨VERB⟩ | ⟨VERB⟩⟨NOUN-PHRASE⟩
⟨ARTICLE⟩ → `a` | `the`
⟨NOUN⟩ → `boy` | `girl` | `flower`
⟨VERB⟩ → `touches` | `likes` | `sees`
⟨PREP⟩ → `with`

Example #5 Derviation

a boy sees

⟨SENTENCE⟩ ⇒ ⟨NOUN-PHRASE⟩⟨VERB-PHRASE⟩

⇒ ⟨CMPLX-NOUN⟩⟨VERB-PHRASE⟩

⇒ ⟨ARTICLE⟩⟨NOUN⟩⟨VERB-PHRASE⟩

⇒ a ⟨NOUN⟩⟨VERB-PHRASE⟩

⇒ a by ⟨VERB-PHRASE⟩

⇒ a by ⟨CMPLX-VERB⟩

⇒ a by ⟨VERB⟩

⇒ a by sees

More examples

- $L_1 = \{0^n \mid n \geq 0\}$
- $L_2 = \{0^n \mid n \geq 1\}$
- $L_3=\{0^i1^j2^k \mid i=j \text{ or } j=k, \text{ where } i,j,k \geq 0\}$
- $L_4=\{0^i1^j2^k \mid i=j \text{ or } i=k, \text{ where } i,j,k \geq 1\}$

Applications of CFLs & CFGs

- *Compilers use parsers for syntactic checking*
- *Parsers can be expressed as CFGs*
  1. *Balancing paranthesis:*
     - ➢ *B ==> BB | (B) | Statement*
     - ➢ *Statement ==> …*
  2. *If-then-else:*
     - ➢ *S ==> SS | if Condition then Statement else Statement | if Condition then Statement | Statement*
     - ➢ *Condition ==> …*
     - ➢ *Statement ==> …*
  - *C paranthesis matching { … }*
  - *Pascal begin-end matching*
  - *YACC (Yet Another Compiler-Compiler)*

More applications

- *Markup languages*
  - *Nested Tag Matching*
    - *HTML*
      - *<html> …<p> … … </p> … </html>*
    - *XML*
      - *<PC> … <MODEL> … </MODEL> .. <RAM> … </RAM> … </PC>*

Tag-Markup Languages

*Roll ==> **<ROLL>** Class Students **</ROLL>***

*Class ==> **<CLASS>** Text **</CLASS>***

Text ==> Char Text | Char

*Char ==> **a | b | … | z | A | B | .. | Z***

Students ==> Student Students | ε

*Student ==> **<STUD>** Text **</STUD>***

Here, the left hand side of each production denotes one non-terminals (e.g., "Roll", "Class", etc.)

Those symbols on the right hand side for which no productions (i.e., substitutions) are defined are terminals (e.g., 'a', 'b', '|', '<', '>', "ROLL", etc.)

Structure of a production

head	derivation	body
A	========>	$\alpha_1 \mid \alpha_2 \mid \ldots \mid \alpha_k$

The above is same as:

1. A ==> α_1
2. A ==> α_2
3. A ==> α_3

…

K. A ==> α_k

CFG conventions

- *Terminal symbols <== a, b, c…*
- *Non-terminal symbols <== A,B,C, …*
- *Terminal or non-terminal symbols <== X,Y,Z*
- *Terminal strings <== w, x, y, z*
- *Arbitrary strings of terminals and non-terminals <==* α, β, γ, ..

Syntactic Expressions in Programming Languages

*result = a*b + score + 10 * distance + c*

terminals variables Operators are also terminals

Regular languages have only terminals

- *Reg expression = [a-z][a-z0-1]**
- *If we allow only letters a & b, and 0 & 1 for constants (for simplification)*
  - *Regular expression = (a+b)(a+b+0+1)**

String membership

How to say if a string belong to the language defined by a CFG?

1. *Derivation*
   - *Head to body*
2. *Recursive inference*
   - *Body to head*

Example:

- *w = 01110*
- *Is w a palindrome?*

Both are equivalent forms

G:
A => 0A0 | 1A1 | 0 | 1 | ε

A => 0A0
=> 01A10
=> 01110

Simple Expressions…

- *We can write a CFG for accepting simple expressions*
- *G = (V,∑,R,S)*
  - *V = {E,F}*
  - *∑ = {0,1,a,b,+,*,(,)}*
  - *S = {E}*
  - *R:*
    - *E ==> E+E | E*E | (E) | F*
    - *F ==> aF | bF | 0F | 1F | a | b | 0 | 1*

Generalization of derivation

- *Derivation is head ==> body*

- *A==>X* (A derives X in a single step)
- *A ==>$^*_G$ X* (A derives X in a multiple steps)

- ***Transitivity:***

  IFA ==>$^*_G$B, and B ==>$^*_G$C, THEN A ==>$^*_G$ C

Context-Free Language

- *The language of a CFG, G=(V,∑,R,S), denoted by L(G), is the set of terminal strings that have a derivation from the start variable S.*
  - $L(G) = \{ w \text{ in } T^* \mid S \Longrightarrow^*_G w \}$

Left-most & Right-most Derivation Styles

G:
E => E+E | E*E | (E) | F
F => aF | bF | 0F | 1F | ε

Derive the string a(ab+10) from G:*

$E \overset{*}{\Longrightarrow}_G a*(ab+10)$

Left-most derivation:

Always substitute leftmost variable

- E
- ==> E * E
- ==> F * E
- ==> a * E
- ==> a * (E)
- ==> a * (E + E)
- ==> a * (F + E)
- ==> a * (aF + E)
- ==> a * (abF + E)
- ==> a * (ab + E)
- ==> a * (ab + F)
- ==> a * (ab + 1F)
- ==> a * (ab + 10F)
- ==> a * (ab + 10)

Right-most derivation:

Always substitute rightmost variable

- E
- ==> E * E
- ==> E * (E)
- ==> E * (E + E)
- ==> E * (E + F)
- ==> E * (E + 1F)
- ==> E * (E + 10F)
- ==> E * (E + 10)
- ==> E * (F + 10)
- ==> E * (aF + 10)
- ==> E * (abF + 0)
- ==> E * (ab + 10)
- ==> F * (ab + 10)
- ==> aF * (ab + 10)
- ==> a * (ab + 10)

Leftmost vs. Rightmost derivations

Q1) For every leftmost derivation, there is a rightmost derivation, and vice versa. True or False?

True - will use parse trees to prove this

Q2) Does every word generated by a CFG have a leftmost and a rightmost derivation?

Yes – easy to prove (reverse direction)

Q3) Could there be words which have more than one leftmost (or rightmost) derivation?

Yes – depending on the grammar

How to prove that your CFGs are correct?

(using induction)

CFG & CFL

G_{pal}:
A => 0A0 | 1A1 | 0 | 1 | ε

- *Theorem:* *A string w in (0+1)* is in $L(G_{pal})$, if and only if, w is a palindrome.*

- *Proof:*
  - *Use induction*
    - *on string length for the IF part*
    - *On length of derivation for the ONLY IF part*

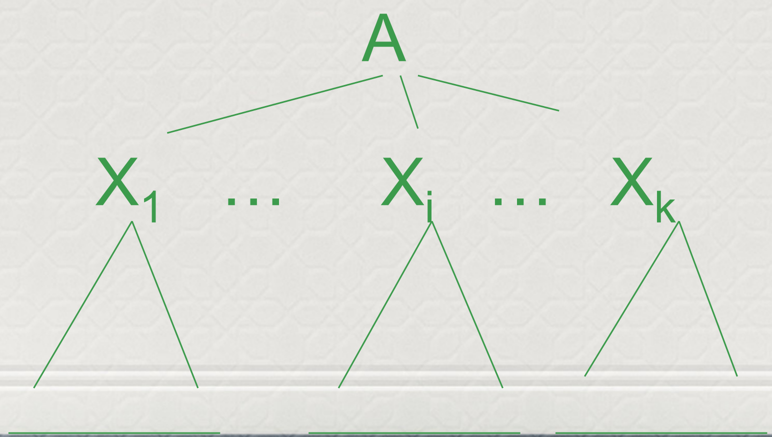
Parse trees

Parse Trees

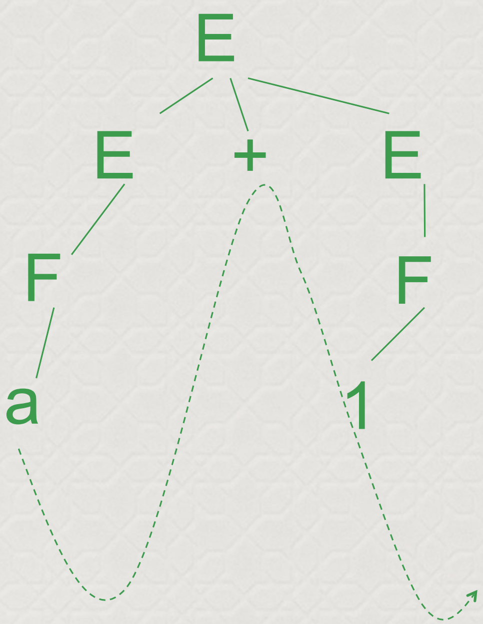
- *Each CFG can be represented using a parse tree:*
  - *Each internal node is labeled by a variable in V*
  - *Each leaf is terminal symbol*
  - *For a production, $A ==> X_1X_2 \ldots X_k$, then any internal node labeled A has k children which are labeled from $X_1, X_2, \ldots X_k$ from left to right*

Parse tree for production and all other subsequent productions:

$A ==> X_1..X_i..X_k$

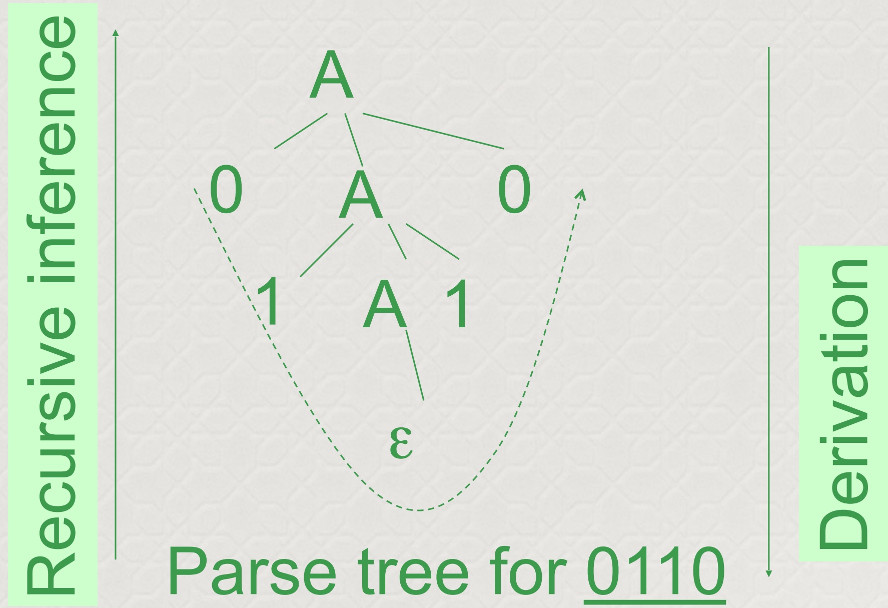


Examples



Parse tree for a + 1

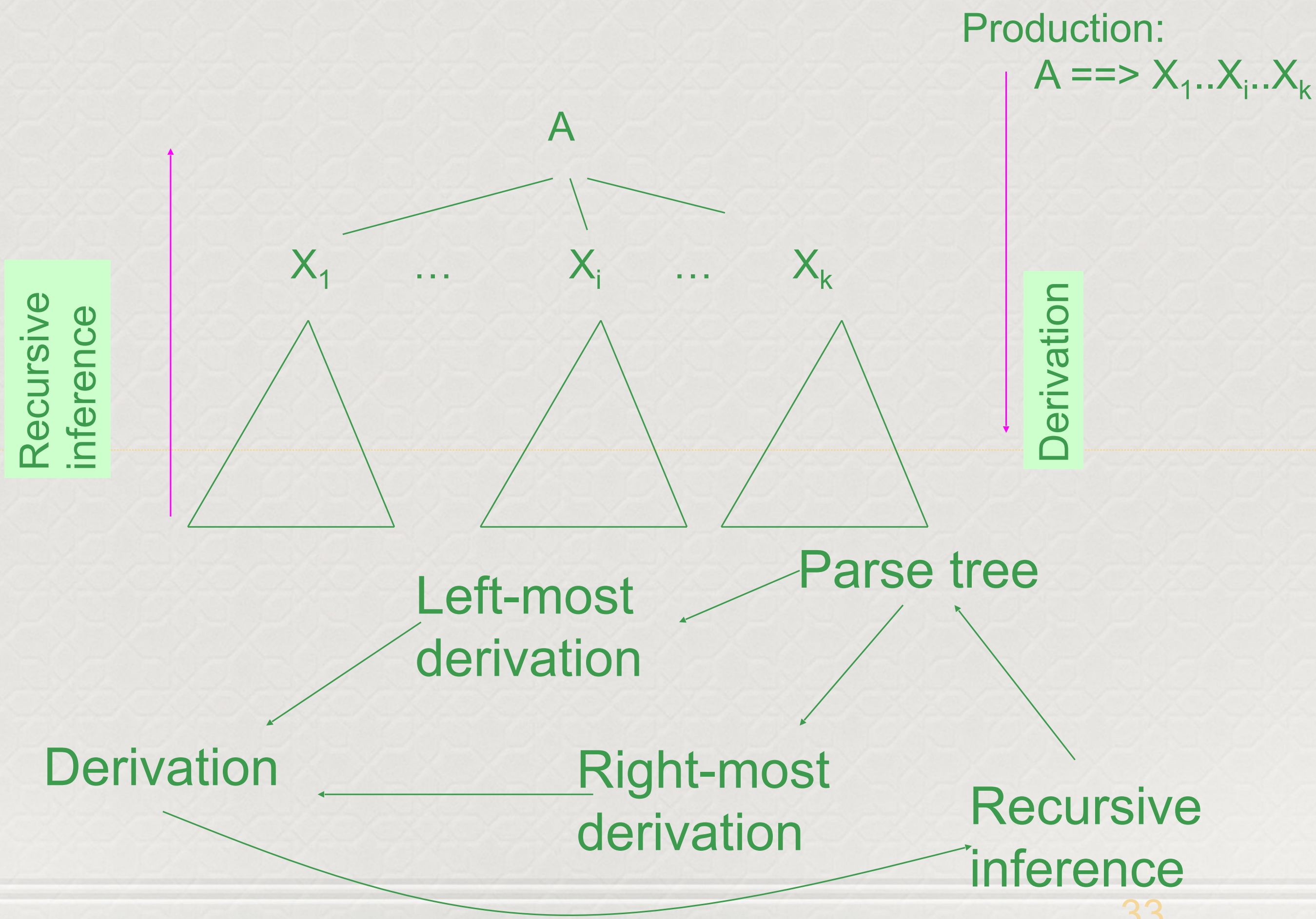
G:
E => E+E | E*E | (E) | F
F => aF | bF | 0F | 1F | 0 | 1 | a | b



Parse tree for 0110

G:
A => 0A0 | 1A1 | 0 | 1 | ε

Parse Trees, Derivations, and Recursive Inferences



Interchangeability of different CFG representations

- *Parse tree ==> left-most derivation*
  - *DFS left to right*
- *Parse tree ==> right-most derivation*
  - *DFS right to left*
- *==> left-most derivation == right-most derivation*
- *Derivation ==> Recursive inference*
  - *Reverse the order of productions*
- *Recursive inference ==> Parse trees*
  - *bottom-up traversal of parse tree*

Ambiguity in CFGs and CFLs

Ambiguity in CFGs

- *A CFG is said to be **ambiguous** if there exists a string which has more than one left-most derivation*

Example:

S ==> AS | ε

A ==> A1 | 0A1 | 01

Input string: 00111

Can be derived in two ways

LM derivation #1:

S => AS

=> 0A1S

=>0A11S

=> 00111S

=> 00111

LM derivation #2:

S => AS

=> A1S

=> 0A11S

=> 00111S

=> 00111

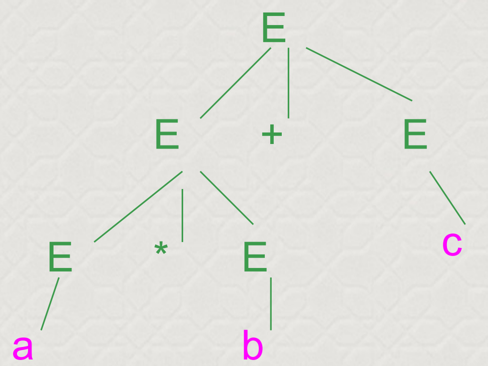
Why does ambiguity matter?

E ==> E + E | E * E | (E) | a | b | c | 0 | 1

Values are different !!!

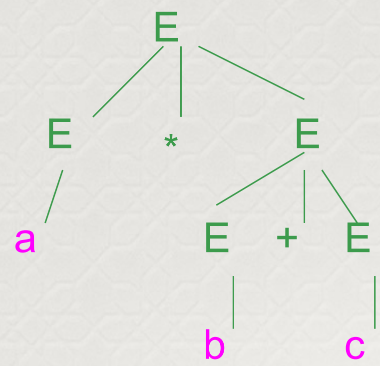
*string = a * b + c*

- LM derivation #1:
  - E => E + E => E * E + E ==>* a * b + c



→ (a*b)+c

- LM derivation #2
  - E => E * E => a * E => a * E + E ==>* a * b + c



→ a*(b+c)

The calculated value depends on which of the two parse trees is actually used.

Connection between CFLs and RLs

Removing Ambiguity in Expression Evaluations

- *It MAY be possible to remove ambiguity for some CFLs*
  - *E.g.,, in a CFG for expression evaluation by imposing rules & restrictions such as precedence*
  - *This would imply rewrite of the grammar*

- *Precedence: (), * , +*

Modified unambiguous version:

E => E + T | T
T => T * F | F
F => I | (E)
I => a | b | c | 0 | 1

Ambiguous version:

E ==> E + E | E * E | (E) | a | b | c | 0 | 1

How will this avoid ambiguity?

Inherently Ambiguous CFLs

- *However, for some languages, it may not be possible to remove ambiguity*

- *A CFL is said to be **inherently ambiguous** if every CFG that describes it is 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 inherently ambiguous*
- *Why?*

Input string: $a^n b^n c^n d^n$

Chomsky Normal Form

- *A context-free grammar $G = (V,\Sigma,R,S)$ is in Chomsky normal form if every rule is of the form*
- *$A \rightarrow BC$ or $A \rightarrow x$*
- *with variables $A \in V$ and $B,C \in V \setminus \{S\}$, and $x \in \Sigma$ For the start variable S we also allow the rule $S \rightarrow \varepsilon$*
- *Advantage: Grammars in this form are far easier to analyse.*

Theorem 2.9

- *Every context-free language can be described by a grammar in Chomsky normal form.*
- *Outline of Proof:*
  - *We rewrite every CFG in Chomsky normal form.*
  - *We do this by replacing, one-by-one, every rule that is not ‘Chomsky’.*
  - *We have to take care of: Starting Symbol, ε symbol, all other violating rules.*

Example of Chomsky NF

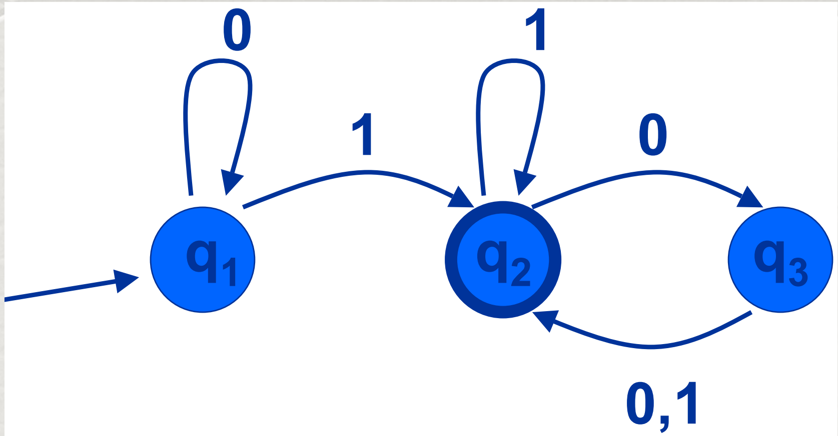
- *Initial grammar:* $S \rightarrow aSb \mid \varepsilon$
- *In Chomsky normal form:*
  - $S_0 \rightarrow \varepsilon \mid T_a T_b \mid T_a X$
  - $X \rightarrow S T_b$
  - $S \rightarrow T_a T_b \mid T_a X$
  - $T_a \rightarrow a$
  - $T_b \rightarrow b$

RL ⊆ CFL

- *Every regular language can be expressed by a context-free grammar.*
- *Proof Idea:*
  - *Given a DFA $M = (Q,\sum,\delta,q_0,F)$, we construct a corresponding CF grammar $GM = (V, \sum,R,S)$ with $V = Q$ and $S = q_0$*
  - *Rules of GM:*
    - $q_i \longrightarrow x\ \delta(q_i,x)$ *for all* $q_i \in V$ *and all* $x \in \sum$
    - $q_i \longrightarrow \varepsilon$ *for all* $q_i \in F$

Example RL ⊆ CFL

- *The DFA*



- *leads to the context-free grammar*
- *$G_M = (Q, \sum, R, q_1)$ with the rules*
  - $q_1 \rightarrow 0q_1 \mid 1q_2$
  - $q_2 \rightarrow 0q_3 \mid 1q_2 \mid \varepsilon$
  - $q_3 \rightarrow 0q_2 \mid 1q_2$

Summary

- *Context-free grammars*
- *Context-free languages*
- *Productions, derivations, recursive inference, parse trees*
- *Left-most & right-most derivations*
- *Ambiguous grammars*
- *Removing ambiguity*
- *CFL/CFG applications*
    - *parsers, markup languages*