SYMBOLIC COMPUTING

Rewrite rules

- $\boxed{\textit{Symbolic computing}}$:

  Strings over an alphabet, jointly represent data and action.

  There are ***no states.***

- The operational engine (analogous to Turing's transition function)

  is the set of ***rewrite-rules,*** also called ***productions.***

Rewrite rules

- ***Symbolic computing***:

  Strings over an alphabet, jointly represent data and action.

  There are ***no states.***

- The operational engine (analogous to Turing's transition function)

  is the set of ***rewrite-rules,*** also called ***productions.***

- A ***rewrite-rule*** is of the form $z \to y$

  where z, y are strings.

- z is the ***source*** of the production, and y its ***target.***

- A finite set of rewrite rules is a ***rewrite system.***

A familiar example of rewriting

$$\begin{aligned}
0 \wedge 0 &\rightarrow 0 \\
0 \wedge 1 &\rightarrow 0 \\
1 \wedge 0 &\rightarrow 0 \\
1 \wedge 1 &\rightarrow 1
\end{aligned}$$

A familiar example of rewriting

$$
\begin{array}{llll}
0 \wedge 0 \rightarrow 0 & \qquad & 0 \vee 0 \rightarrow 0 \\
0 \wedge 1 \rightarrow 0 & \qquad & 0 \vee 1 \rightarrow 1 \\
1 \wedge 0 \rightarrow 0 & \qquad & 1 \vee 0 \rightarrow 1 \\
1 \wedge 1 \rightarrow 1 & \qquad & 1 \vee 1 \rightarrow 1
\end{array}
$$

A familiar example of rewriting

$0 \wedge 0 \rightarrow 0$	$0 \vee 0 \rightarrow 0$	$-0 \rightarrow 1$
$0 \wedge 1 \rightarrow 0$	$0 \vee 1 \rightarrow 1$	$-1 \rightarrow 0$
$1 \wedge 0 \rightarrow 0$	$1 \vee 0 \rightarrow 1$	
$1 \wedge 1 \rightarrow 1$	$1 \vee 1 \rightarrow 1$	

A familiar example of rewriting

$0 \wedge 0 \to 0$	$0 \vee 0 \to 0$	$-0 \to 1$	$(0) \to 0$
$0 \wedge 1 \to 0$	$0 \vee 1 \to 1$	$-1 \to 0$	$(1) \to 1$
$1 \wedge 0 \to 0$	$1 \vee 0 \to 1$		
$1 \wedge 1 \to 1$	$1 \vee 1 \to 1$		

Reductions and derivations

- Given a rewrite system R,
  we say that w **reduces to** w', and write $\boxed{w \Rightarrow_R w'}$,
  if w' is w with substring u replaced by u',
  here $u \to u'$ is a rule.

  We omit the subscript R when clear.
- Reductions are analogous to the yield relation
  between machine's configurations.

Reductions and derivations

- Given a rewrite system R,
  we say that w **reduces to** w', and write $\boxed{w \Rightarrow_R w'}$,
  if w' is w with substring u replaced by u',
  here $u \to u'$ is a rule.

  We omit the subscript R when clear.

- Reductions are analogous to the yield relation
  between machine's configurations.

- A **derivation** in R is a sequence

$$w_0,\ w_1,\ w_2, \ldots w_k$$

  where $w_i \in \Gamma$ and $w_i \Rightarrow_R w_{i+1}$ for $i < k$.

  This derivation is ***of*** w_k ***from*** w_0 .

Reductions and derivations

- Given a rewrite system R,
  we say that w ***reduces to*** w', and write $\boxed{w \Rightarrow_R w'}$,
  if w' is w with substring u replaced by u',
  here $u \rightarrow u'$ is a rule.

  We omit the subscript R when clear.
- Reductions are analogous to the yield relation
  between machine's configurations.
- A ***derivation*** in R is a sequence

  $$w_0,\ w_1,\ w_2, \ldots w_k$$

  where $w_i \in \Gamma$ and $w_i \Rightarrow_R w_{i+1}$ for $i < k$.

  This derivation is ***of*** w_k ***from*** w_0 **.**
- Derivations are analogous to computation traces of machines.

Recap: The reflexive-transitive closure

- If $R : A \longrightarrow A$ then the $\boxed{\textit{reflexive-transitive closure}}$ of R is the mapping $R^* : A \to A$ defined by: xR^*z ***iff*** $x = y_0\ (R)\ y_1$

Recap: The reflexive-transitive closure

- If $R: A \longrightarrow A$ then the $\boxed{\textit{reflexive-transitive closure}}$ of R is the mapping $R^*: A \to A$ defined by: xR^*z ***iff*** $x = y_0 \; (R) \; y_1$
- A generative definition of R^* :
  - $x \; (R^*) \; x$
  - If $x \; (R) \; y$ and $y \; (R^*) \; z$ then $x \; (R^*) \; z$.

Recap: The reflexive-transitive closure

- If $R : A \longrightarrow A$ then the $\boxed{\textit{reflexive-transitive closure}}$ of R is the mapping $R^* : A \to A$ defined by: xR^*z ***iff*** $x = y_0 \, (R) \, y_1$
- A generative definition of R^* :
  - $x \, (R^*) \, x$
  - If $x \, (R) \, y$ and $y \, (R^*) \, z$ then $x \, (R^*) \, z$.
- So w_k is derived from w_0 as above exactly when $w_0 \Rightarrow^* w_k$.

Example

A derivation in our boolean rewrite-system:

$$
\begin{array}{ll}
\multicolumn{2}{l}{((0) \wedge (1)) \vee (1)} \\
\quad \Rightarrow & (0 \wedge (1)) \vee (1) \\
\quad \Rightarrow & (0 \wedge 1) \vee (1) \\
\quad \Rightarrow & (0) \vee (1) \\
\quad \Rightarrow & 0 \vee (1) \\
\quad \Rightarrow & 0 \vee 1 \\
\quad \Rightarrow & 1
\end{array}
$$

Example

A derivation in our boolean rewrite-system:

$$
\begin{aligned}
&((0) \wedge (1)) \vee (1) \\
&\qquad \Rightarrow \quad (0 \wedge (1)) \vee (1) \\
&\qquad \Rightarrow \quad (0 \wedge 1) \vee (1) \\
&\qquad \Rightarrow \quad (0) \vee (1) \\
&\qquad \Rightarrow \quad 0 \vee (1) \\
&\qquad \Rightarrow \quad 0 \vee 1 \\
&\qquad \Rightarrow \quad 1
\end{aligned}
$$

- Here we ended up with the ***irreducible*** string 1, which cannot be reduced further.

Grammars

- Rewrite systems can be transducers, acceptors, or ***generators.***
- A rewrite system that generates a language is a ***grammar.***
- A $\boxed{\textbf{\textit{grammar}}}$ consists of

Grammars

- Rewrite systems can be transducers, acceptors, or ***generators.***
- A rewrite system that generates a language is a ***grammar.***
- A ***grammar*** consists of
  - An input alphabet Σ. (We say that G is ***over*** Σ).
  - A finite set V of fresh symbols (not in Σ), dubbed ***variables***. (We write Γ for $\Sigma \cup V$.)
  - A distinguished ***initial-variable***. Default: $\mathtt{S}$.

Grammars

- Rewrite systems can be transducers, acceptors, or ***generators.***
- A rewrite system that generates a language is a ***grammar.***
- A ***grammar*** consists of
  - An input alphabet Σ. (We say that G is ***over*** Σ).
  - A finite set V of fresh symbols (not in Σ), dubbed ***variables***. (We write Γ for $\Sigma \cup V$.)
  - A distinguished ***initial-variable***. Default: $\mathtt{S}$.
  - A finite set R of rewrite rules, called ***productions.***

  These are of the form $w \to t$

  where w has ***at least one non-terminal.***

Examples

Take $\Sigma = \{\mathtt{a}, \mathtt{b}\}$ and $V = \{\mathbf{S}\}$.

1. Two productions: $\mathbf{S} \to \mathtt{a}$ and $\mathbf{S} \to \mathtt{bb}$.

Examples

Take $\Sigma = \{\mathtt{a}, \mathtt{b}\}$ and $V = \{\mathtt{S}\}$.

1. Two productions: $\mathtt{S} \to \mathtt{a}$ and $\mathtt{S} \to \mathtt{bb}$.
2. Two productions: $\mathtt{S} \to \varepsilon$ and $\mathtt{S} \to \mathtt{aS}$

Examples

Take $\Sigma = \{\mathtt{a}, \mathtt{b}\}$ and $V = \{\mathtt{S}\}$.

1. Two productions: $\mathtt{S} \to \mathtt{a}$ and $\mathtt{S} \to \mathtt{bb}$.
2. Two productions: $\mathtt{S} \to \varepsilon$ and $\mathtt{S} \to \mathtt{aS}$
3. A non-example: rewrite rules $\mathtt{a} \to \mathtt{ab}$ and $\mathtt{b} \to \mathtt{ba}$.

Each grammar generates a language

- Let $G = (\Sigma, V, \mathtt{S}, R)$ be a grammar.
  $w \in \Sigma^*$ is $\boxed{\textit{derived}}$ in G if
  it is derived from $\mathtt{S}$.
- The $\boxed{\textit{language generated by } G}$ is
  $$\mathcal{L}(G) = \{w \in \Sigma^* \mid S \Rightarrow^* w\}$$

Examples

- Grammar G has productions $S \rightarrow \mathtt{a}$ and $S \rightarrow \mathtt{b}$.
  $\mathcal{L}(G) = \{\mathtt{a}, \mathtt{b}\}$.

Examples

- Grammar G has productions $S \to \mathtt{a}$ and $S \to \mathtt{b}$.
  $\mathcal{L}(G) = \{\mathtt{a}, \mathtt{b}\}$.
- Grammar G has productions $S \to \mathtt{a}S$ and $S \to \mathtt{b}$.
- Some derivations:

$$S \Rightarrow \mathtt{b}$$

$$S \Rightarrow \mathtt{a}S \Rightarrow \mathtt{ab}$$

$$S \Rightarrow \mathtt{a}S \Rightarrow \mathtt{aa}S \Rightarrow \mathtt{aab}$$

Examples

- Grammar G has productions $S \to \mathtt{a}$ and $S \to \mathtt{b}$.
  $\mathcal{L}(G) = \{\mathtt{a}, \mathtt{b}\}$.
- Grammar G has productions $S \to \mathtt{a}S$ and $S \to \mathtt{b}$.
- Some derivations:

$$S \Rightarrow \mathtt{b}$$

$$S \Rightarrow \mathtt{a}S \Rightarrow \mathtt{ab}$$

$$S \Rightarrow \mathtt{a}S \Rightarrow \mathtt{aa}S \Rightarrow \mathtt{aab}$$

- $\mathcal{L}(G) = \{\mathtt{a}^n \cdot \mathtt{b} \mid n \geqslant 0\} = \mathcal{L}(\mathtt{a}^* \cdot \mathtt{b})$.

Examples

- Grammar G has productions $S \to \mathtt{a}$ and $S \to \mathtt{b}$.
  $\mathcal{L}(G) = \{\mathtt{a}, \mathtt{b}\}$.
- Grammar G has productions $S \to \mathtt{a}S$ and $S \to \mathtt{b}$.
- Some derivations:

$$S \Rightarrow \mathtt{b}$$

$$S \Rightarrow \mathtt{a}S \Rightarrow \mathtt{ab}$$

$$S \Rightarrow \mathtt{a}S \Rightarrow \mathtt{aa}S \Rightarrow \mathtt{aab}$$

- $\mathcal{L}(G) = \{\mathrm{a}^n \cdot \mathrm{b} \mid n \geqslant 0\} = \mathcal{L}(\mathrm{a}^* \cdot \mathrm{b})$.
- How to formally prove this?

Examples

- Grammar G has productions $S \to \mathtt{a}$ and $S \to \mathtt{b}$.

  $\mathcal{L}(G) = \{\mathtt{a}, \mathtt{b}\}$.
- Grammar G has productions $S \to \mathtt{a}S$ and $S \to \mathtt{b}$.
- Some derivations:

$$S \Rightarrow \mathtt{b}$$

$$S \Rightarrow \mathtt{a}S \Rightarrow \mathtt{ab}$$

$$S \Rightarrow \mathtt{a}S \Rightarrow \mathtt{aa}S \Rightarrow \mathtt{aab}$$

- $\mathcal{L}(G) = \{\mathtt{a}^n \cdot \mathtt{b} \mid n \geqslant 0\} = \mathcal{L}(\mathtt{a}^* \cdot \mathtt{b})$.
  - By induction every string $\mathtt{a}^n$ is generated.
  - By induction $S \Rightarrow_G^{n+1} w$ implies that w is either $\mathtt{a}^n\mathtt{b}$ or $\mathtt{a}^{n+1}S$.

More examples

- G's productions are $S \to \mathtt{a}S$, $S \to S\mathtt{b}$ and $S \to \varepsilon$.

More examples

- G's productions are $S \to \mathtt{a}S$, $S \to S\mathtt{b}$ and $S \to \varepsilon$.
- $\mathcal{L}(G) = ?$

More examples

- G's productions are $S \to \mathtt{a}S$, $S \to S\mathtt{b}$ and $S \to \varepsilon$.
- $\mathcal{L}(G) = \mathcal{L}(\mathtt{a}^*\mathtt{b}^*)$

More examples

- G's productions are $S \to \mathtt{a}S$, $S \to S\mathtt{b}$ and $S \to \varepsilon$.
- $\mathcal{L}(G) = \mathcal{L}(\mathtt{a}^*\mathtt{b}^*)$
- $S \to \mathtt{a}S\mathtt{b}$ and $S \to \varepsilon$.

More examples

- G's productions are $S \to \mathrm{a}S$, $S \to S\mathrm{b}$ and $S \to \varepsilon$.
- $\mathcal{L}(G) = \mathcal{L}(\mathrm{a}^*\mathrm{b}^*)$
- $S \to \mathrm{a}S\mathrm{b}$ and $S \to \varepsilon$.
- Some derivations:

$$S \Rightarrow \varepsilon$$

$$S \Rightarrow \mathrm{a}S\mathrm{b} \Rightarrow \mathbf{ab}$$

$$S \Rightarrow \mathrm{a}S\mathrm{b} \Rightarrow \mathbf{aa}S\mathbf{bb} \Rightarrow \mathbf{aabb}$$

$$S \Rightarrow \mathrm{a}S\mathrm{b} \Rightarrow \mathbf{aa}S\mathbf{bb} \Rightarrow \mathbf{aaa}S\mathbf{bbb} \Rightarrow \mathbf{aaabbb}$$

More examples

- G's productions are $S \to \mathrm{a}S$, $S \to S\mathrm{b}$ and $S \to \varepsilon$.
- $\mathcal{L}(G) = \mathcal{L}(\mathrm{a}^*\mathrm{b}^*)$
- $S \to \mathrm{a}S\mathrm{b}$ and $S \to \varepsilon$.
- Some derivations:

$$S \Rightarrow \varepsilon$$

$$S \Rightarrow \mathrm{a}S\mathrm{b} \Rightarrow \mathbf{ab}$$

$$S \Rightarrow \mathrm{a}S\mathrm{b} \Rightarrow \mathbf{aa}S\mathbf{bb} \Rightarrow \mathbf{aabb}$$

$$S \Rightarrow \mathrm{a}S\mathrm{b} \Rightarrow \mathbf{aa}S\mathbf{bb} \Rightarrow \mathbf{aaa}S\mathbf{bbb} \Rightarrow \mathbf{aaabbb}$$

- $\mathcal{L}(G) \;\; = ?$

More examples

- G's productions are $S \to \mathrm{a}S$, $S \to S\mathrm{b}$ and $S \to \varepsilon$.
- $\mathcal{L}(G) = \mathcal{L}(\mathrm{a}^*\mathrm{b}^*)$
- $S \to \mathrm{a}S\mathrm{b}$ and $S \to \varepsilon$.
- Some derivations:

$$S \Rightarrow \varepsilon$$
$$S \Rightarrow \mathrm{a}S\mathrm{b} \Rightarrow \mathbf{ab}$$
$$S \Rightarrow \mathrm{a}S\mathrm{b} \Rightarrow \mathbf{aa}S\mathbf{bb} \Rightarrow \mathbf{aabb}$$
$$S \Rightarrow \mathrm{a}S\mathrm{b} \Rightarrow \mathbf{aa}S\mathbf{bb} \Rightarrow \mathbf{aaa}S\mathbf{bbb} \Rightarrow \mathbf{aaabbb}$$

- $\mathcal{L}(G) = \{\mathrm{a}^n\mathrm{b}^n \mid n \geqslant 0\}$. A non-regular language!

CONTEXT FREE GRAMMARS

Context-free grammars

- A $\boxed{\textit{context-free grammar (CFG)}}$ is a grammar where every source is ***a single non-terminal***.
- All grammars we've seen so far are context-free.
- A language generated by a CFG is a $\boxed{\textit{context-free language (CFL).}}$
- Context-free grammars are also called $\boxed{\textit{inductive grammars.}}$
- A convention: bundle rules with a common source as in $S \rightarrow \mathtt{a}\,S\,\mathtt{b} \mid \varepsilon$.
  The vertical line abbreviates ***"or".***

Example: palindromes

- Let P be the initial non-terminal.
- Productions:

$$\begin{array}{rcl} P & \rightarrow & \mathtt{a}P\mathtt{a} \\ P & \rightarrow & \mathtt{b}P\mathtt{b} \\ P & \rightarrow & \mathtt{a} \\ P & \rightarrow & \mathtt{b} \\ P & \rightarrow & \varepsilon \end{array}$$

- In BNF format: $P \rightarrow \mathtt{a}P\mathtt{a} \mid \mathtt{b}P\mathtt{b} \mid \mathtt{a} \mid \mathtt{b} \mid \varepsilon$

- Similar grammar for palindromes over the entire Latin alphabet. We have then $2 \cdot 26 + 1 = 53$ productions.
- Using the more economical grammar

$$\begin{aligned} P &\rightarrow LPL \mid L \mid \varepsilon \\ L &\rightarrow \mathtt{a} \mid \mathtt{b} \mid \cdots \mid \mathtt{z} \end{aligned}$$

is wrong, because the two L's in LPS should be the same.

- But we can use a modular description of the correct grammar above:

$$P \rightarrow \sigma P \sigma \mid \sigma \mid \varepsilon \quad (\sigma \in \Sigma)$$

CFLs for natural languages

- *The bone ate the dog* is grammatically correct English
  The dog the bone ate is not
- There is a context-free grammar that generates
  exactly the grammatically correct sentences in English!
- Not 100% for all languages, more sophisticated formalisms are needed.

An example for English

- Alphabet Σ consists of the six “symbols”:

  `dog`, `apple`, `eats`, `loves`, `big`, and `green`.

An example for English

- Alphabet Σ consists of the six “symbols”:

  `dog`, `apple`, `eats`, `loves`, `big`, and `green`.

- Nonterminals:

  S for sentences,

  P for noun-phrases

  N for nouns

  V for verbs

  A for adjectives.

An example for English

- Alphabet Σ consists of the six "symbols":

  `dog`, `apple`, `eats`, `loves`, `big`, and `green`.

- The productions are

$$
\begin{aligned}
S &\rightarrow PVP \\
P &\rightarrow N \mid AP \\
N &\rightarrow \texttt{dog} \mid \texttt{apple} \\
V &\rightarrow \texttt{eats} \mid \texttt{loves} \\
A &\rightarrow \texttt{big} \mid \texttt{green}
\end{aligned}
$$

- This grammar generates *big dog eats green apple*
  and *big green big apple loves green dog*
  but not *eats big dog apple loves*.

An example for English

- Alphabet Σ consists of the six "symbols":

  `dog`, `apple`, `eats`, `loves`, `big`, and `green`.

- The productions are

$$\begin{array}{rcl} S & \to & PVP \\ P & \to & N \mid AP \\ N & \to & \texttt{dog} \mid \texttt{apple} \\ V & \to & \texttt{eats} \mid \texttt{loves} \\ A & \to & \texttt{big} \mid \texttt{green} \end{array}$$

- This grammar generates *big dog eats green apple*
  and *big green big apple loves green dog*
  but not *eats big dog apple loves*.

The CF-Factoring Theorem

- Intuitively clear: context-free productions guarantee a separation

  between descendents of one occurrence of a variable
  and descendents of another.

- That is:

  CF-Factoring Theorem.
  Let $G = (\Sigma, N, S, R)$ *be a CFG,* $\Gamma = \Sigma \cup N$.
  For strings $u_0, u_1 \in \Gamma^*$ *, if* $u_0 \cdot u_1 \Rightarrow^* v$
  then $v = v_0 \cdot v_1$ *where* $u_0 \Rightarrow^* v_0$ *and* $u_1 \Rightarrow^* v_1$.

- We prove by induction on n that if $u_0 \cdot u_1 \Rightarrow^n v$
  then the conclusion above holds.

Symmetries in CFL

- CFGs often generate languages with symmetries (eg palindromes!).
- The language of balanced parentheses, e.g. $(())()$ is balanced, $(()($ is not.
- The alphabet: just left- and right-parentheses: $($ and $)$,
- Productions: $S \to SS \mid (S) \mid \varepsilon$

A CFG is a generative definition

- Each CFG describes a generative process:
  A variable X names the language generated from X.

A CFG is a generative definition

- Each CFG describes a generative process:
  A variable X names the language generated from X.
- Here's a CFG $G_{a=b}$ that generates $\{w \in \Sigma^* \mid \#_a(w) = \#_b(w)\}$

A CFG is a generative definition

- Each CFG describes a generative process:
  A variable X names the language generated from X.
- Here's a CFG $G_{a=b}$ that generates $\{w \in \Sigma^* \mid \#_a(w) = \#_b(w)\}$
  - Let A name $\{w \in \Sigma^* \mid \#_a(w) = \#_b(w) + 1\}$,
    and B name $\{w \in \Sigma^* \mid \#_b(w) = \#_a(w) + 1\}$.

A CFG is a generative definition

- Each CFG describes a generative process:
  A variable X names the language generated from X.
- Here's a CFG $G_{a=b}$ that generates $\{w \in \Sigma^* \mid \#_a(w) = \#_b(w)\}$
  - Let A name $\{w \in \Sigma^* \mid \#_a(w) = \#_b(w) + 1\}$,
    and B name $\{w \in \Sigma^* \mid \#_b(w) = \#_a(w) + 1\}$.
  - The productions of $G_{a=b}$ are

$$\begin{array}{lcl} S & \to & \varepsilon \mid \mathtt{a}B \mid \mathtt{b}A \\ A & \to & \mathtt{a}\,S \mid \mathtt{b}\,A\,A \\ B & \to & \mathtt{b}\,S \mid \mathtt{a}\,B\,B \end{array}$$

A CFG is a generative definition

- Each CFG describes a generative process:
  A variable X names the language generated from X.
- Here's a CFG $G_{a=b}$ that generates $\{w \in \Sigma^* \mid \#_a(w) = \#_b(w)\}$
  - Let A name $\{w \in \Sigma^* \mid \#_a(w) = \#_b(w) + 1\}$,
    and B name $\{w \in \Sigma^* \mid \#_b(w) = \#_a(w) + 1\}$.
  - The productions of $G_{a=b}$ are

$$
\begin{aligned}
S &\to \varepsilon \mid \mathtt{a}B \mid \mathtt{b}A \\
A &\to \mathtt{a}\,S \mid \mathtt{b}\,A\,A \\
B &\to \mathtt{b}\,S \mid \mathtt{a}\,B\,B
\end{aligned}
$$

  - $\mathcal{L}(G_{a=b}) = \{w \in \Sigma^* \mid \#_a(w) = \#_b(w)\}$

A CFG is a generative definition

- Each CFG describes a generative process:
  A variable X names the language generated from X.
- Here's a CFG $G_{a=b}$ that generates $\{w \in \Sigma^* \mid \#_a(w) = \#_b(w)\}$
  - Let A name $\{w \in \Sigma^* \mid \#_a(w) = \#_b(w) + 1\}$,
    and B name $\{w \in \Sigma^* \mid \#_b(w) = \#_a(w) + 1\}$.
  - The productions of $G_{a=b}$ are

$$\begin{aligned} S &\to \varepsilon \mid \mathtt{a}B \mid \mathtt{b}A \\ A &\to \mathtt{a}\,S \mid \mathtt{b}\,A\,A \\ B &\to \mathtt{b}\,S \mid \mathtt{a}\,B\,B \end{aligned}$$

  - $\mathcal{L}(G_{a=b}) = \{w \in \Sigma^* \mid \#_a(w) = \#_b(w)\}$
- Exercise: The grammar with productions $S \to \mathtt{b} \mid \mathtt{a}SS$ generates the strings with $\#_b > \#_a$ but $\#_b \leqslant \#_a$ for all proper-prefixes.

A grammar that is not context-free

- Let $\Sigma = \{\mathtt{a}, \mathtt{bc}\}$. We shall see later that
  $$L_{a=b=c} = \{w \in \Sigma^* \mid \#_a(w) = \#_b(w) = \#_c(w)\}$$
  is not CF.

A grammar that is not context-free

- Let $\Sigma = \{\mathtt{a}, \mathtt{bc}\}$. We shall see later that
  $$L_{a=b=c} = \{w \in \Sigma^* \mid \#_a(w) = \#_b(w) = \#_c(w)\ \}$$
  is not CF.

- Consider the grammar

$$S \ \rightarrow \ \varepsilon \mid SABC$$
$$A \rightarrow \mathtt{a}, \qquad B \rightarrow \mathtt{b}, \qquad C \rightarrow \mathtt{c}$$

- It generates the strings $(\mathtt{abc})^n$.

A grammar that is not context-free

- Let $\Sigma = \{\mathtt{a}, \mathtt{bc}\}$. We shall see later that
$$L_{a=b=c} = \{w \in \Sigma^* \mid \#_a(w) = \#_b(w) = \#_c(w)\ \}$$
is not CF.

- Consider the grammar
$$S \;\to\; \varepsilon \mid SABC$$
$$A \to \mathtt{a}, \qquad B \to \mathtt{b}, \qquad C \to \mathtt{c}$$

- It generates the strings $(\mathtt{abc})^n$.

- Add the productions $AB \;\to\; BA,\;\; AC \to CA \qquad BC \;\to CB$.
$BA \;\to\; AB,\;\; CA \to AC \qquad CB \;\to BC$.
Yes, these are ***not*** context-free!

A grammar that is not context-free

- Let $\Sigma = \{\mathtt{a}, \mathtt{bc}\}$. We shall see later that
  $$L_{a=b=c} = \{w \in \Sigma^* \mid \#_a(w) = \#_b(w) = \#_c(w)\ \}$$
  is not CF.

- Consider the grammar

$$S \ \rightarrow \ \varepsilon \mid SABC$$
$$A \rightarrow \mathtt{a}, \qquad B \rightarrow \mathtt{b}, \qquad C \rightarrow \mathtt{c}$$

- It generates the strings $(\mathtt{abc})^n$.

- Add the productions $AB \ \rightarrow \ BA,\ \ AC \rightarrow CA \qquad BC \ \rightarrow CB$.
  $BA \ \rightarrow \ AB,\ \ CA \rightarrow AC \qquad CB \ \rightarrow BC$.
  Yes, these are ***not*** context-free!

- This extended grammar generates $L_{a=b=c}$

Multiple symmetries

- $\{\mathtt{a}^n\mathtt{b}^n\mathtt{c}^k \mid n, k \geqslant 0\}$
- $\{\mathtt{a}^n\mathtt{b}^n\mathtt{a}^k\mathtt{b}^k \mid n, k \geqslant 0\}$
- $\{\mathtt{a}^n\mathtt{b}^{n+k}\mathtt{a}^k \mid n, k \geqslant 0\}$
- $\{\mathtt{a}^n\mathtt{b}^k\mathtt{c}^{n+k} \mid n, k \geqslant 0\}$
- $\{\mathtt{a}^n\mathtt{b}^k\mathtt{a}^k\mathtt{b}^n \mid n, k \geqslant 0\}$
- $\{\mathtt{a}^n\mathtt{b}^{n+k}\mathtt{c}^{k+m}\mathtt{d}^m \mid n, k, m \geqslant 0\}$

REGULAR LANGUAGES ARE CONTEXT-FREE

- We use the generative definition of the strictly regular languages. Their definition as strictly-regular languages is simplifying this.

- We use the generative definition of the strictly regular languages. Their definition as strictly-regular languages is simplifying this.
- Recall that the strictly-regular languages over Σ are generated by:

1. The trivial languages $\emptyset,\ \{\boldsymbol{\varepsilon}\},\ \{\boldsymbol{\sigma}\}\ \ (\sigma \in \Sigma)$ are regular.
2. The union, concatenation, and star of regular languages are regular.

- We use the generative definition of the strictly regular languages. Their definition as strictly-regular languages is simplifying this.
- Recall that the strictly-regular languages over Σ are generated by:

1. The trivial languages $\emptyset,\ \{\boldsymbol{\varepsilon}\},\ \{\boldsymbol{\sigma}\}\ \ (\sigma \in \Sigma)$ are regular.
2. The union, concatenation, and star of regular languages are regular.

- We use the generative definition of the strictly regular languages. Their definition as strictly-regular languages is simplifying this.
- Recall that the strictly-regular languages over Σ are generated by:

  1. The trivial languages $\emptyset,\ \{\boldsymbol{\varepsilon}\},\ \{\boldsymbol{\sigma}\}\ \ (\sigma \in \Sigma)$ are regular.
  2. The union, concatenation, and star of regular languages are regular.

- So by induction on the collection of regular languages they are all CF.

The regular languages are CF

- $\emptyset$:

The regular languages are CF

- $\emptyset$: Generated by the CFG $S \to S$.
- $\{\varepsilon\}$:

The regular languages are CF

- $\emptyset$: Generated by the CFG $S \to S$.
- $\{\varepsilon\}$: Generated by $S \to \varepsilon$.
- $\{\mathtt{a}\}$:

Closure under union, concatenation, star

Refer to CFGs and the languages they generated:

$L_0 = \mathcal{L}(G_0)$ and $L_1 = \mathcal{L}(G_1)$ where $G_i = (\Sigma, V_i, S_i, R_i)$.

We may assume that G_0 and G_1 have no variable in common: renaming a grammar's variables
does not change the language generated.

Closure under union

- $L_0 \cup L_1$ is generated by $(\Sigma,\ V \cup V' + S,\ S,\ R)$
  where S is a fresh nonterminal
  and R is $R_0 \cup R_1$ augmented with the production $S \to S_0 \mid S$

Closure under union

- $L_0 \cup L_1$ is generated by $(\Sigma,\ V \cup V' + S,\ S,\ R)$
  where S is a fresh nonterminal
  and R is $R_0 \cup R_1$ augmented with the production $S \to S_0 \mid S$
- G generates each $w \in L_0 \cup L_1$.

Closure under union

- $L_0 \cup L_1$ is generated by $(\Sigma,\ V \cup V' + S,\ S,\ R)$
  where S is a fresh nonterminal
  and R is $R_0 \cup R_1$ augmented with the production $S \to S_0 \mid S$
- G generates each $w \in L_0 \cup L_1$.
- Conversely, a derivation D in G for $S \Rightarrow_G w$
  must start with $S \to S_0$ or $S \to S_1$ and proceed with
  either a derivation in G_0 or a derivation in G_1 ,
  since $V_0 \cap V_1 = \emptyset$.

Closure under concatenation

- In this section we let L_0, L_1 be CFLs generated by CFGs $G_0 = (\Sigma, N_0, S_0, R_0)$ and $G_1 = (\Sigma, N_1, S_1, R_1)$ respectively,
  with no non-terminals in common ($N_0 \cap N_1 = \emptyset$).
- Let N be $N_0 \cup N_1$ + a fresh nonterminal S.

Closure under concatenation

- In this section we let L_0, L_1 be CFLs generated by CFGs $G_0 = (\Sigma, N_0, S_0, R_0)$ and $G_1 = (\Sigma, N_1, S_1, R_1)$ respectively,
  with no non-terminals in common ($N_0 \cap N_1 = \emptyset$).
- Let N be $N_0 \cup N_1$ + a fresh nonterminal S.
- A grammar generating $L_0 \cup L_1$:
  (Σ, N, S, R) where R is $R_0 \cup R_0$ augmented with the production $S \rightarrow S_0 \mid S_1$.

Closure under concatenation

- In this section we let L_0, L_1 be CFLs generated by CFGs $G_0 = (\Sigma, N_0, S_0, R_0)$ and $G_1 = (\Sigma, N_1, S_1, R_1)$ respectively,
with no non-terminals in common ($N_0 \cap N_1 = \emptyset$).
- Let N be $N_0 \cup N_1$ + a fresh nonterminal S.
- A grammar generating $L_0 \cup L_1$:
(Σ, N, S, R) where R is $R_0 \cup R_0$ augmented with the production $S \to S_0 \mid S_1$.
- A grammar generating $L_0 \cdot L_1$: (Σ, N, S, R) where R is $R_0 \cup R_0$ augmented with the production $S \to S_0 S_1$.

Regular languages are context-free

- The trivial finite languages are CF.
- The CFLs are closed under union, concatenation and star.
- By induction on the definition of regular languages:

  Theorem. *Every regular language is CF*

- But not every CFL is regular: $\{\mathtt{a}^n\mathtt{b}^n \mid n \geqslant 0\}$ is CF.

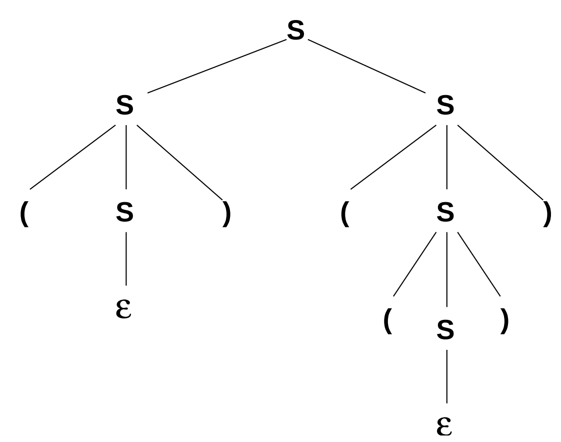
DERIVATIONS AND REPETITIONS

Parse-trees

- Recall CFG for balanced parentheses: $S \to \varepsilon \mid SS \mid (S)$
- A derivation for the string $()(())$:

$$S \Rightarrow SS \Rightarrow S(S) \Rightarrow (S)(S) \Rightarrow ()(S) \Rightarrow ()((S)) \Rightarrow ()(())$$

- Represented as a tree with terminals for leaves and variables for internal nodes:



- This is a $\boxed{\textbf{derivation tree}}$ ***of*** w ***in*** G (aka $\boxed{\textbf{parse tree}}$).

Leftmost-derivations

- Derivations for the same parse-tree are said to be $\boxed{\textit{equivalent.}}$
- Example:
  In addition to $S \Rightarrow SS \Rightarrow S(S) \Rightarrow (S)(S) \Rightarrow ()(S) \Rightarrow ()((S)) \Rightarrow ()(())$
  we also have $S \Rightarrow SS \Rightarrow (S)S \Rightarrow ()S \Rightarrow ()(S) \Rightarrow ()((S)) \Rightarrow ()(())$
- The latter is the $\boxed{\textit{leftmost derivation}}$ for the tree.
- Generally: The leftmost derivation for a parse-tree expands at each step the ***leftmost*** variable.

Example

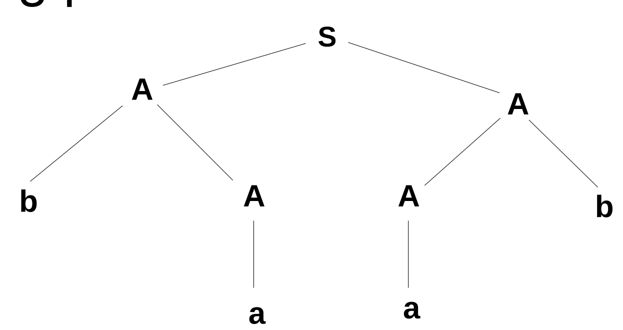
- G is

$$S \to AAS \mid \varepsilon, \qquad A \to \mathtt{b}A \mid A\mathtt{b} \mid \mathtt{a}$$

- Here is a derivation of **`baab`** :

$$\begin{aligned} S &\Rightarrow_G AA \\ &\Rightarrow_G bAA \\ &\Rightarrow_G bAAb \\ &\Rightarrow_G bAab \\ &\Rightarrow_G baab \end{aligned}$$

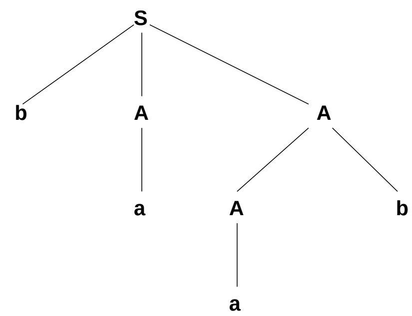
- The corresponding parse-tree is



- The leftmost derivation for this tree is

$$S \Rightarrow_G AA \Rightarrow_G \mathtt{b}AA \Rightarrow_G \mathtt{ba}A \Rightarrow_G \mathtt{ba}A\mathtt{b} \Rightarrow_G \mathtt{baab}$$

- A different parse-tree for the same string:

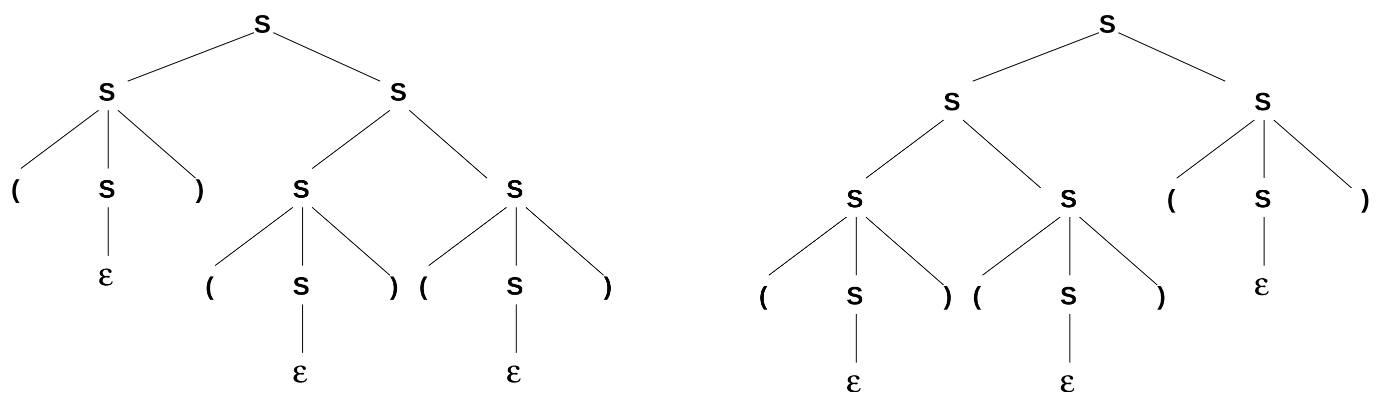


The leftmost derivation for that tree:

$$S \Rightarrow_G \mathtt{b}AA \Rightarrow_G \mathtt{ba}A \Rightarrow_G \mathtt{ba}A\mathtt{b} \Rightarrow_G \mathtt{baab}$$

Ambiguous grammars

- A parse-tree usually represents several derivations.
  Can a grammar have different parse-*trees* for the same string?
- We have already seen one: $S \to SS \mid (S) \mid \varepsilon$.



- And natural languages are full of ambiguities:

  Jane welcomed the man with a dog

  Jane welcomed the man with a dog

Inherently ambiguous CFLs

- There are non-ambiguous grammars to generate the above.
- A CFL is $\boxed{\textit{inherently ambiguous}}$ if
  all grammars generating it are ambiguous.
- Example (no proof):

$$\{\mathrm{a}^n\mathrm{b}^n\mathrm{c}^k \mid n, k \geqslant 0\} \cup \{\mathrm{a}^k\mathrm{b}^n\mathrm{c}^n \mid n, k \geqslant 0\}$$

- Any grammar generating this language has at least two derivations
  for, say, `aabbcc`
- Remark: The intersection is not CF at all.

Parse-trees

- Computation traces capture the nature of procedural computing by a mathematical machine.
- But a formal derivation by a grammar G conveys an order that is not part of the intended generative process.

- Recall CFG for balanced parentheses: $S \rightarrow \varepsilon \mid SS \mid (S)$

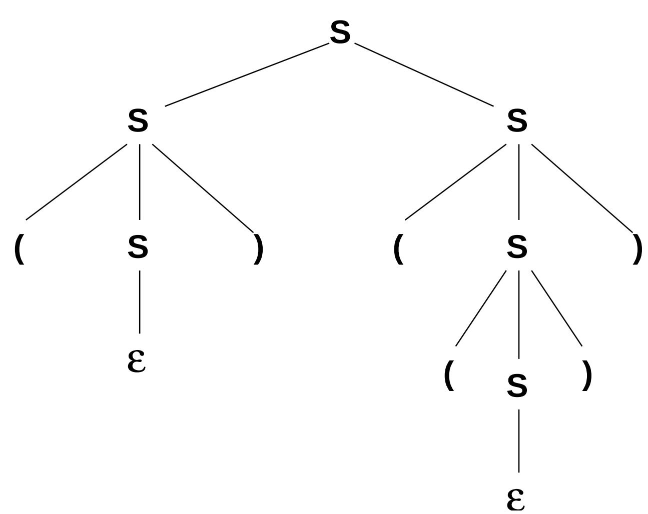
- Recall CFG for balanced parentheses: $S \to \varepsilon \mid SS \mid (S)$
- A derivation for the string $()(())$:

$$S \Rightarrow SS \Rightarrow S(S) \Rightarrow (S)(S) \Rightarrow ()(S) \Rightarrow ()((S)) \Rightarrow ()(())$$

- Recall CFG for balanced parentheses: $S \to \varepsilon \mid SS \mid (S)$
- A derivation for the string $()(())$:

$$S \Rightarrow SS \Rightarrow S(S) \Rightarrow (S)(S) \Rightarrow ()(S) \Rightarrow ()((S)) \Rightarrow ()(())$$

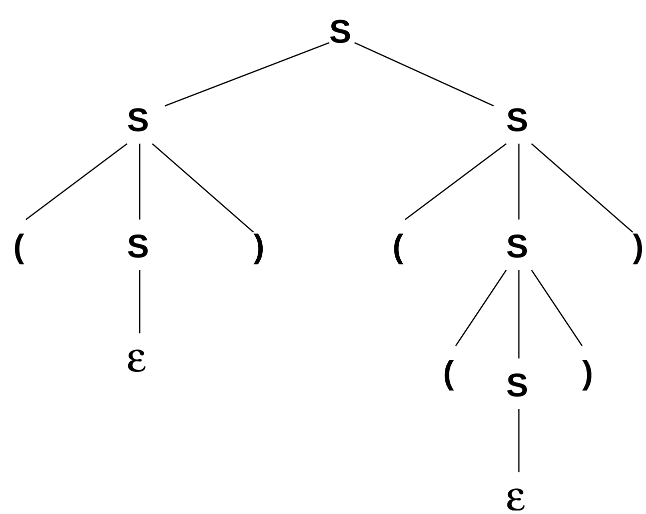
- Represented as a tree with terminals for leaves and variables for internal nodes:



- Recall CFG for balanced parentheses: $S \to \varepsilon \mid SS \mid (S)$
- A derivation for the string $()(())$:

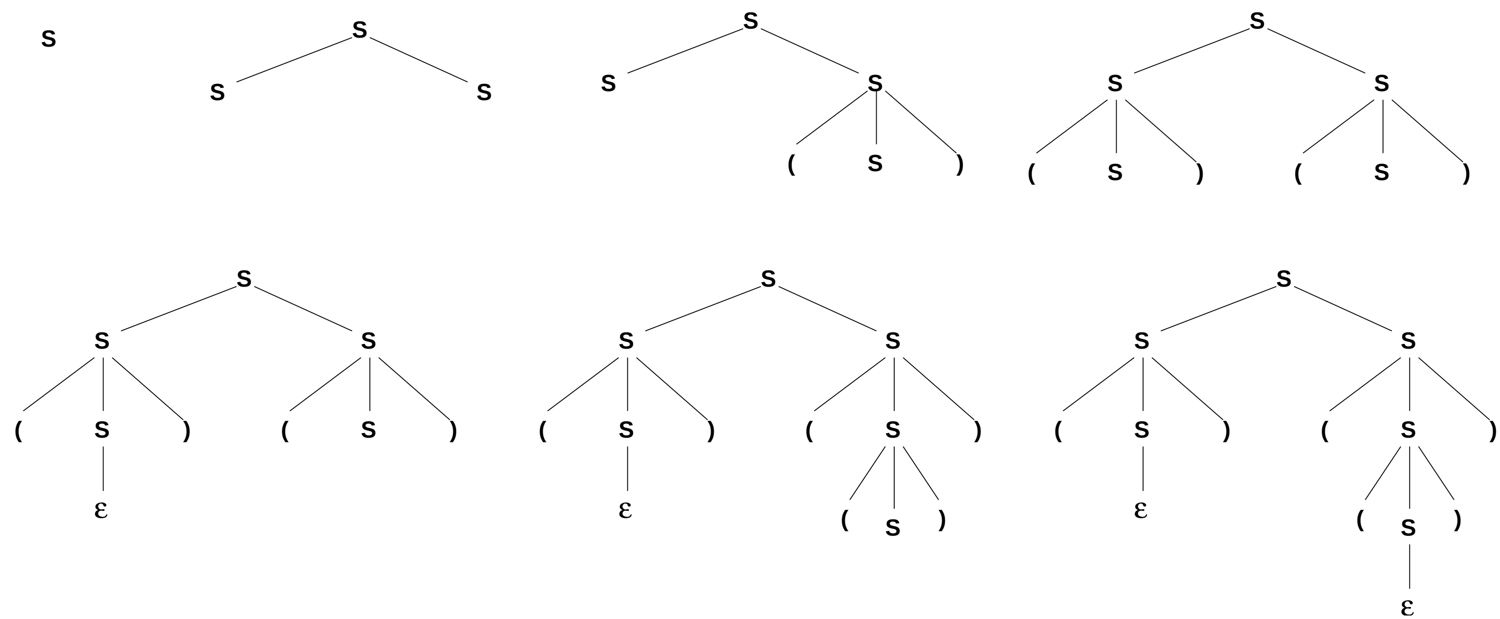
  $$S \Rightarrow SS \Rightarrow S(S) \Rightarrow (S)(S) \Rightarrow ()(S) \Rightarrow ()((S)) \Rightarrow ()(())$$

- Represented as a tree with terminals for leaves
  and variables for internal nodes:

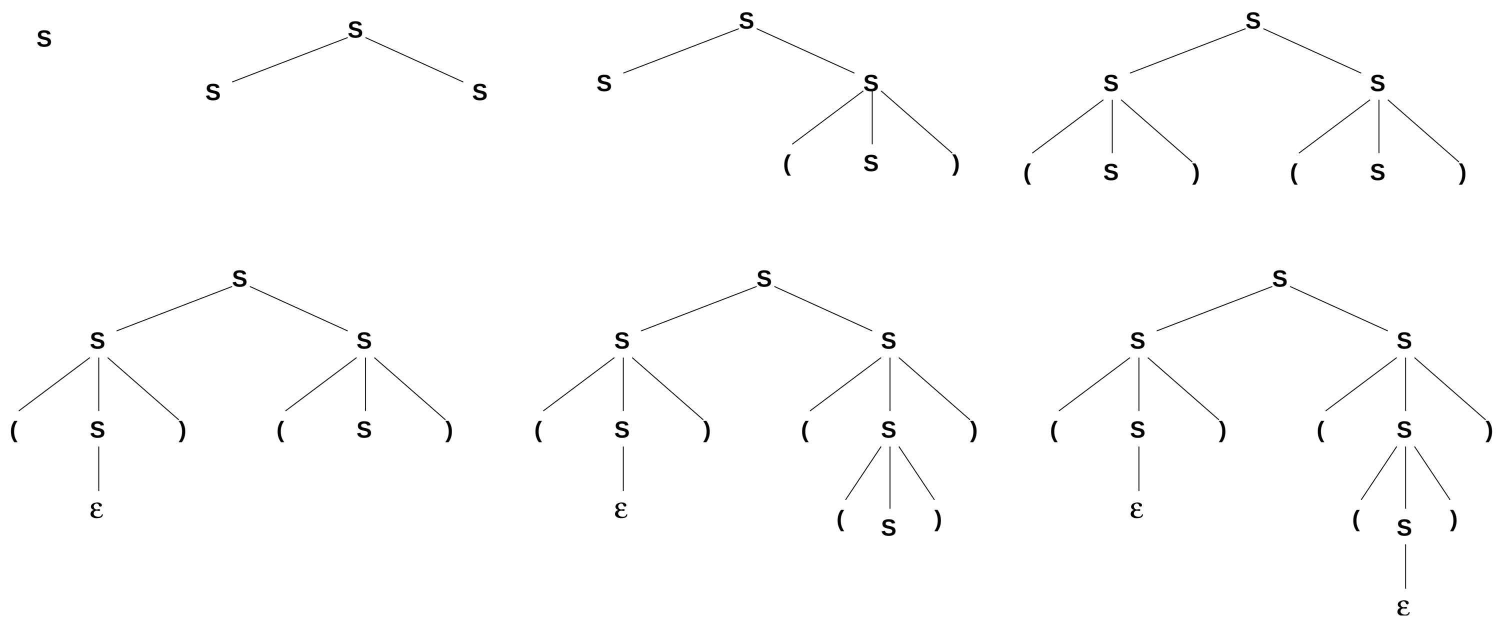


- This is a ***parse-tree,*** or ***pars-tree***
  (of the grammar G for the string w).

- The parse-tree can be built using the derivation above:

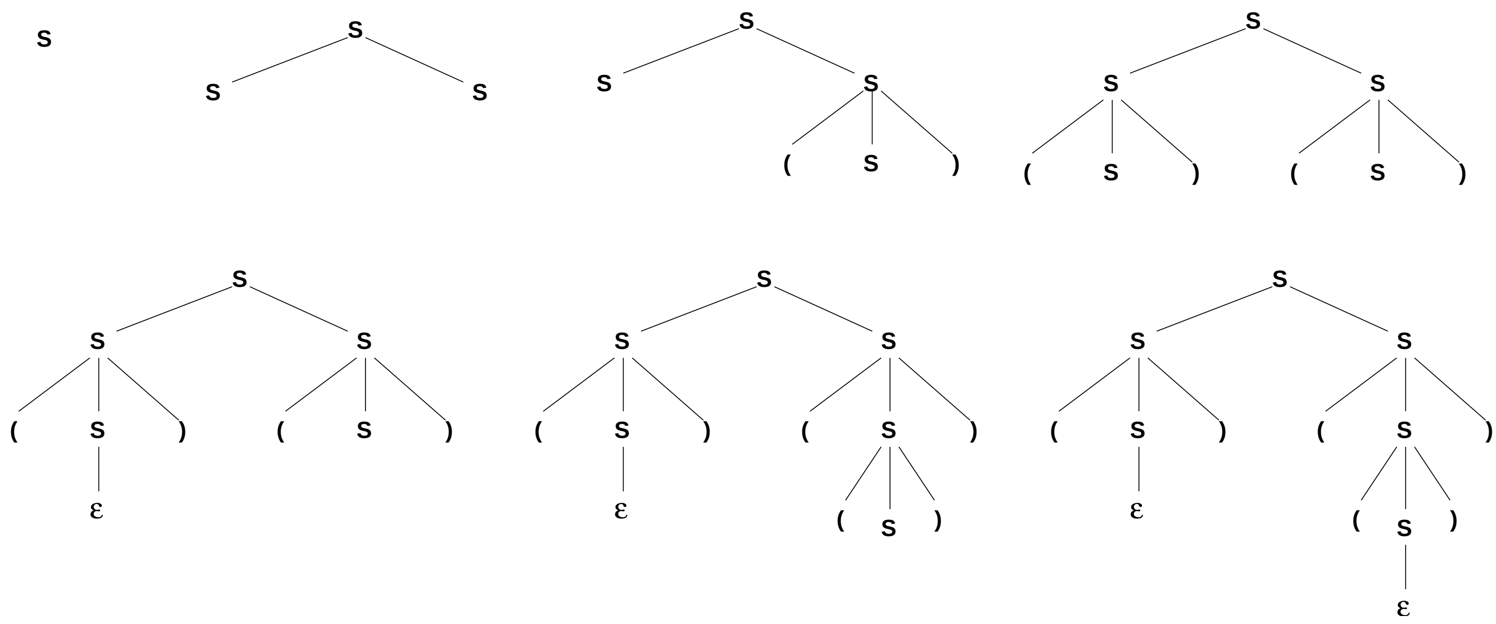


- The parse-tree can be built using the derivation above:



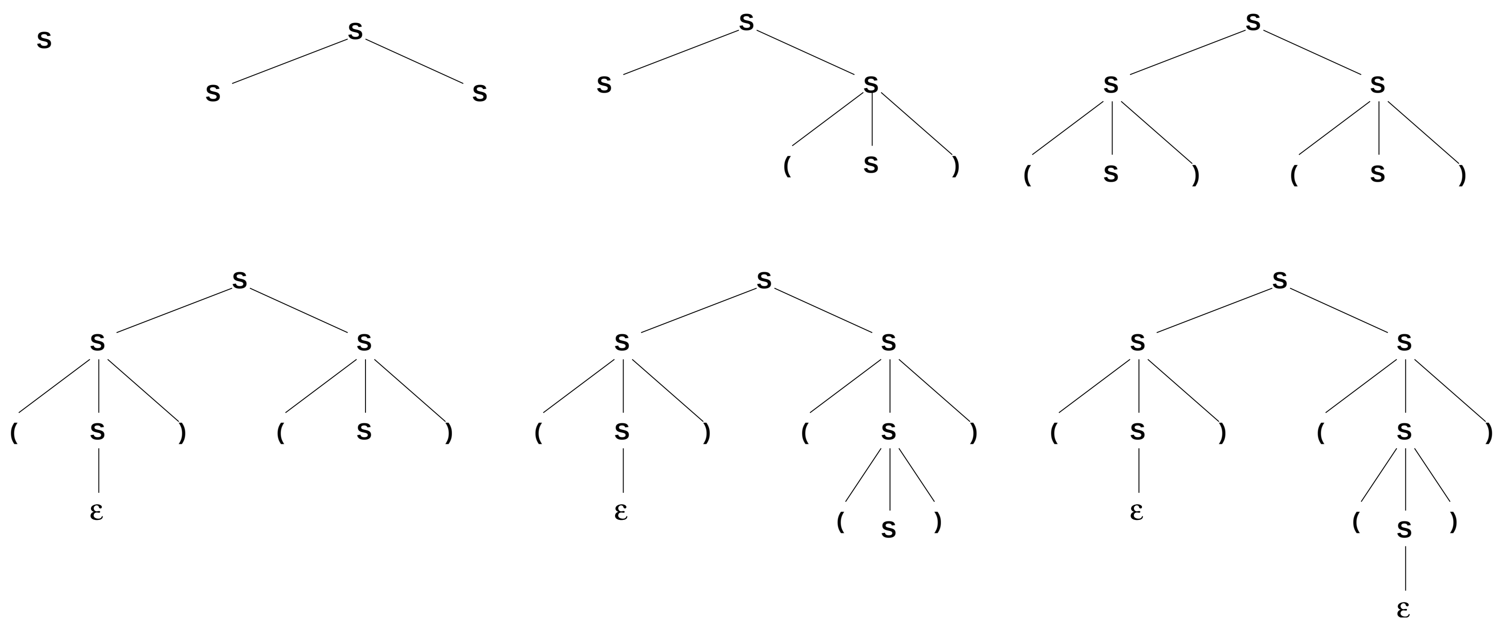
- The parse-tree represents the essential features of a derivation, abstracting away from the sequential listing of the steps.

- The parse-tree can be built using the derivation above:



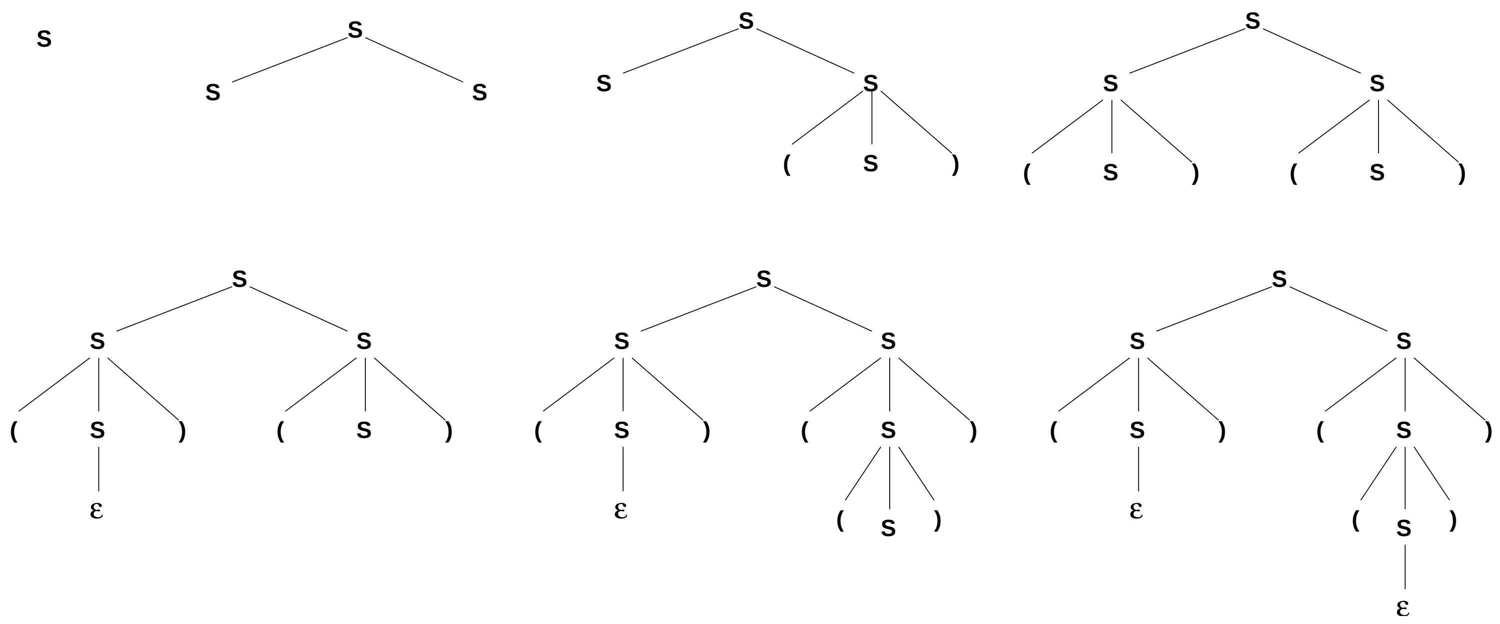
- The parse-tree represents the essential features of a derivation, abstracting away from the sequential listing of the steps.
- This is analogous to a set $\{a_1, \ldots a_k\}$, which abstracts away from the order of the list $a_1 \ldots, a_k$.

- The parse-tree can be built using the derivation above:



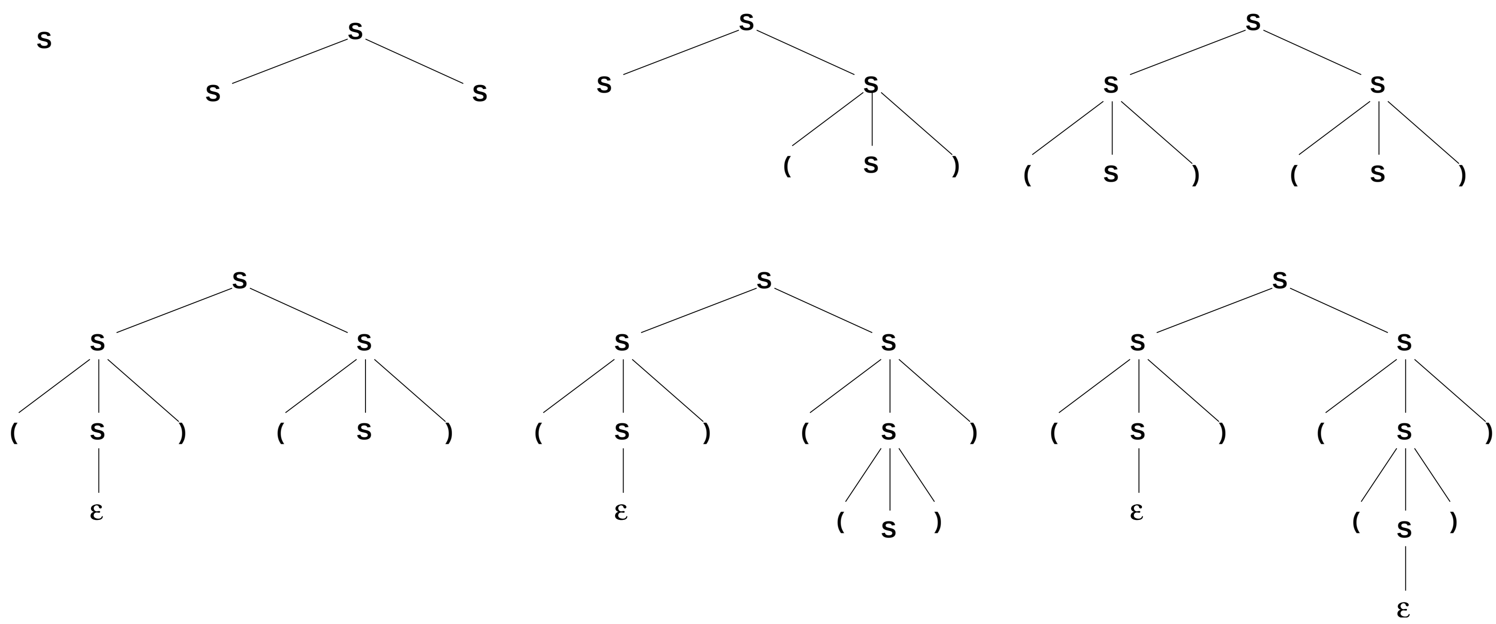
- The parse-tree represents the essential features of a derivation, abstracting away from the sequential listing of the steps.
- This is analogous to a set $\{a_1, \ldots a_k\}$, which abstracts away from the order of the list $a_1 \ldots, a_k$.
- This is why parse-trees are also called *“abstract syntax-trees”.*

- The parse-tree can be built using the derivation above:



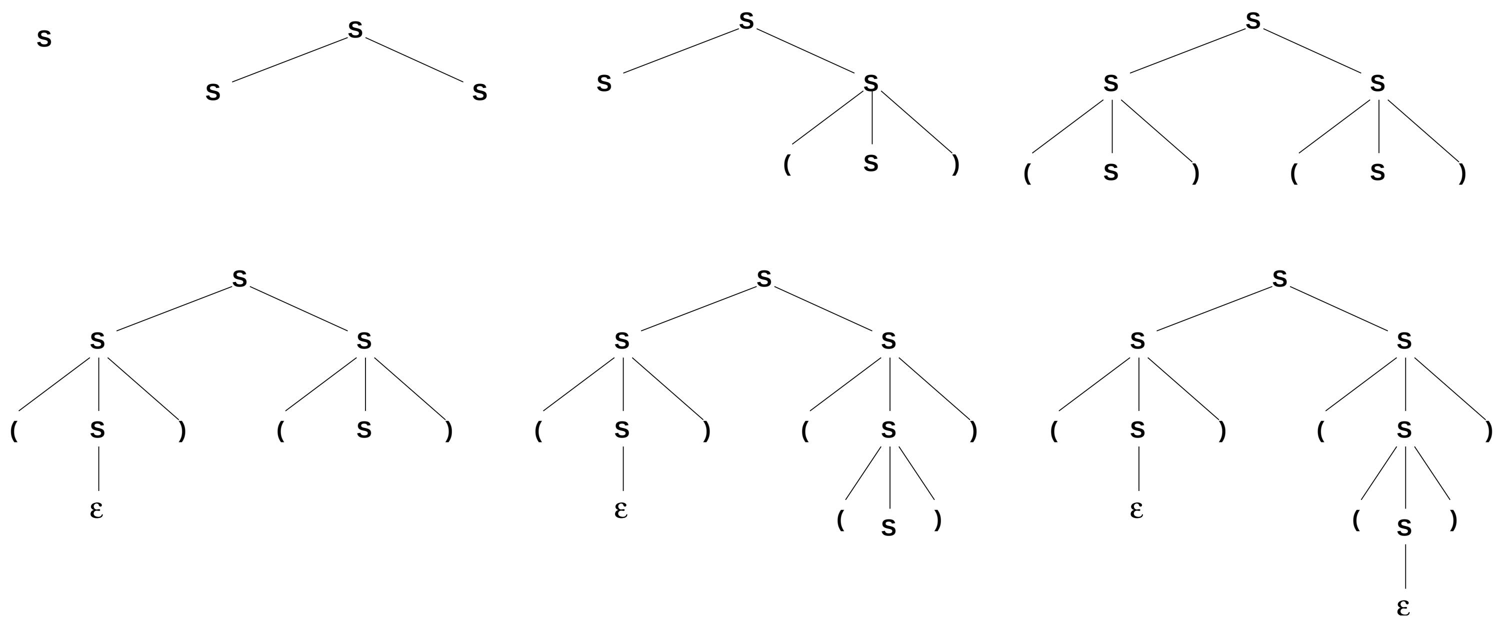
- The parse-tree represents the essential features of a derivation, abstracting away from the sequential listing of the steps.
- Different derivations for the same tree are ***equivalent.***

- The parse-tree can be built using the derivation above:



- The parse-tree represents the essential features of a derivation, abstracting away from the sequential listing of the steps.
- Different derivations for the same tree are $\boxed{\textit{equivalent.}}$
- E.g. besides $S \Rightarrow SS \Rightarrow S(S) \Rightarrow (S)(S) \Rightarrow ()(S) \Rightarrow ()((S)) \Rightarrow ()(())$
  we also have $S \Rightarrow SS \Rightarrow (S)S \Rightarrow ()S \Rightarrow ()(S) \Rightarrow ()((S)) \Rightarrow ()(())$

- The parse-tree can be built using the derivation above:



- The parse-tree represents the essential features of a derivation, abstracting away from the sequential listing of the steps.
- Different derivations for the same tree are ***equivalent.***
- E.g. besides $S \Rightarrow SS \Rightarrow S(S) \Rightarrow (S)(S) \Rightarrow ()(S) \Rightarrow ()((S)) \Rightarrow ()(())$
  we also have $S \Rightarrow SS \Rightarrow (S)S \Rightarrow ()S \Rightarrow ()(S) \Rightarrow ()((S)) \Rightarrow ()(())$

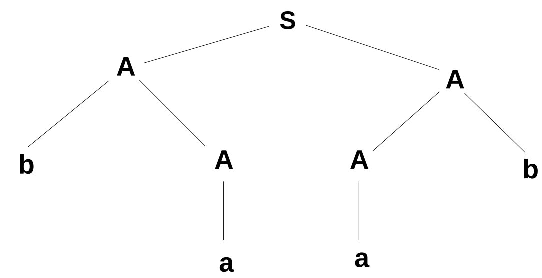
- The latter is the $\boxed{\textit{leftmost-derivation}}$ for the tree, obtained by repeatedly expanding the leftmost variable.

Example

- Grammar G : $\quad S \to AA \mid \mathtt{b}AA, \qquad A \to \mathtt{b}A \mid A\mathtt{b} \mid \mathtt{a}$
- A derivation of **`baab`** :

$$S \Rightarrow_G AA \Rightarrow_G \mathtt{b}AA \Rightarrow_G \mathtt{b}AA\mathtt{b} \Rightarrow_G \mathtt{b}\,A\mathtt{a}\,\mathtt{b} \Rightarrow_G \mathtt{baab}$$

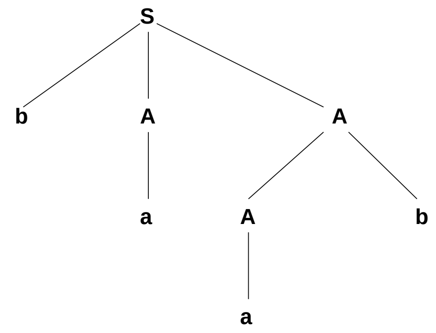
- The corresponding parse-tree:



- The leftmost derivation for this is

$$S \Rightarrow_G AA \Rightarrow_G \mathtt{b}AA \Rightarrow_G \mathtt{ba}A \Rightarrow_G \mathtt{ba}A\mathtt{b} \Rightarrow_G \mathtt{baab}$$

A different parse-tree for the same string:

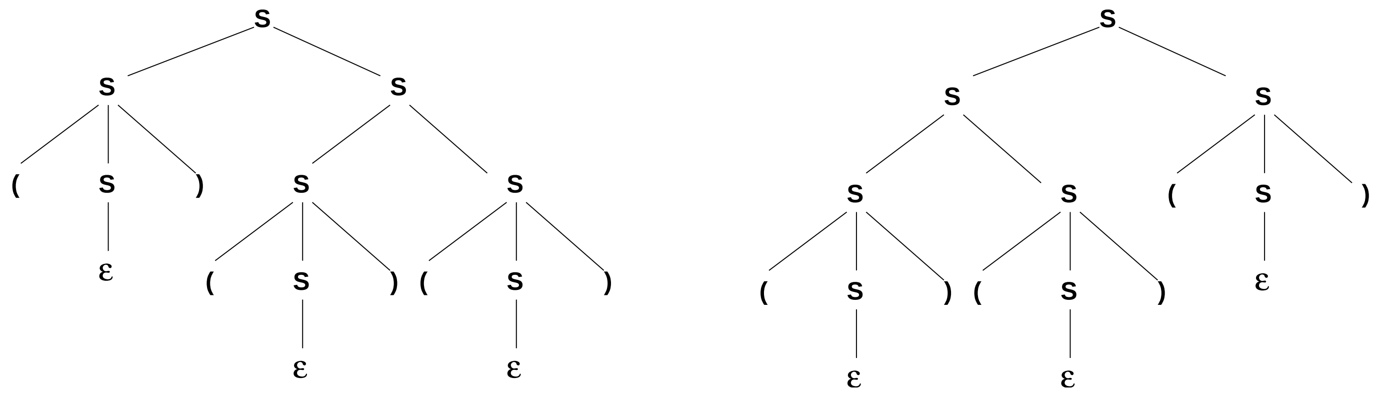


The leftmost derivation for this parse-tree:

$$S \Rightarrow_G \mathtt{b}AA \Rightarrow_G \mathtt{ba}A \Rightarrow_G \mathtt{ba}A\mathtt{b} \Rightarrow_G \mathtt{baab}$$

Ambiguous grammars

- A parse-tree usually represents several derivations.
  Can a grammar have different parse-trees for the same string?
- We have already seen one: $S \to SS \mid (S) \mid \varepsilon$.



- And natural languages are full of ambiguities:

  Jane welcomed the speaker with a smile
  Jane welcomed the speaker with a smile

Inherently ambiguous CFLs

- There are non-ambiguous grammars to generate the above. That's good!
- A CFL is $\boxed{\textit{inherently ambiguous}}$
  if ***all*** grammars generating it are ambiguous.

  ıExample (no proof):

$$\{a^n b^n c^k \mid n, k \geqslant 0\} \cup \{a^k b^n c^n \mid n, k \geqslant 0\}$$

- Any grammar for this language will have at least two derivations for every string $a^n b^n c^n$.

REPEATED PARSING PATTERNS

Dual-clipping in CFLs

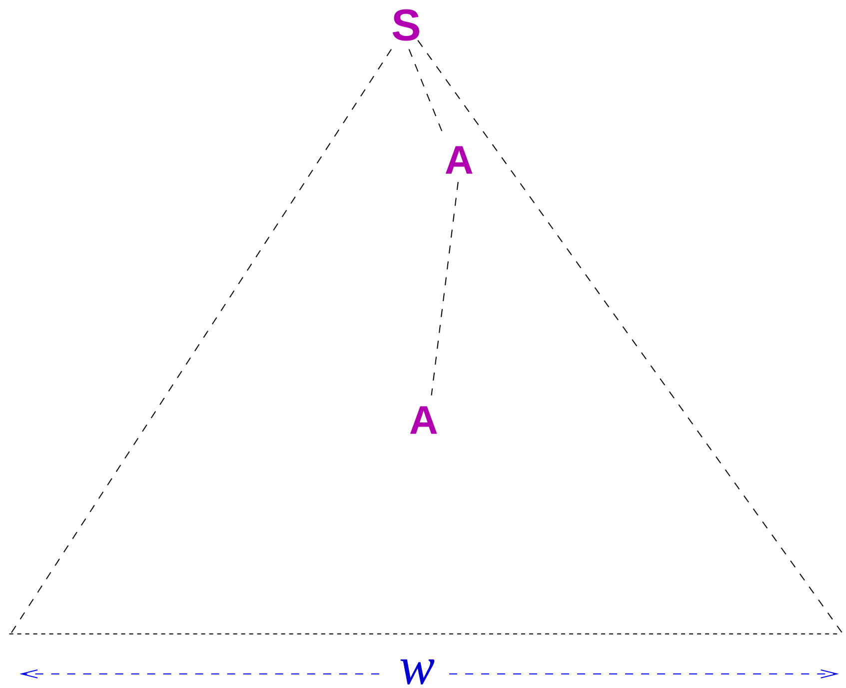
- The Clipping Theorem is based on the observation that if M is a k-state DFA then any trace of M of length $\geqslant k$ has some state q repeating.

Dual-clipping in CFLs

- The Clipping Theorem is based on the observation that if M is a k-state DFA then any trace of M of length $\geqslant k$ has some state q repeating.
- This does not work as stated for for CFLs. Why?

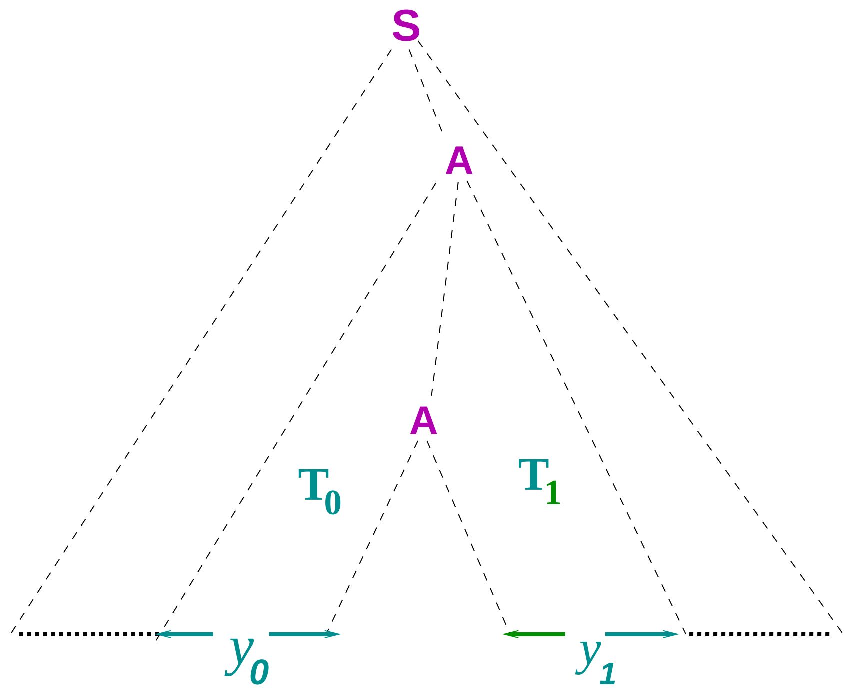
Dual-clipping in CFLs

- The Clipping Theorem is based on the observation that if M is a k-state DFA then any trace of M of length $\geqslant k$ has some state q repeating.
- This does not work as stated for for CFLs. Why?
- A DFA accepts a string w by a textual scan, but a CFG generates w by a parse-tree for it.
  Here the repetition is "vertical":
  A variable repeats on a ***branch*** of the parse-tree.


S
A
A
w

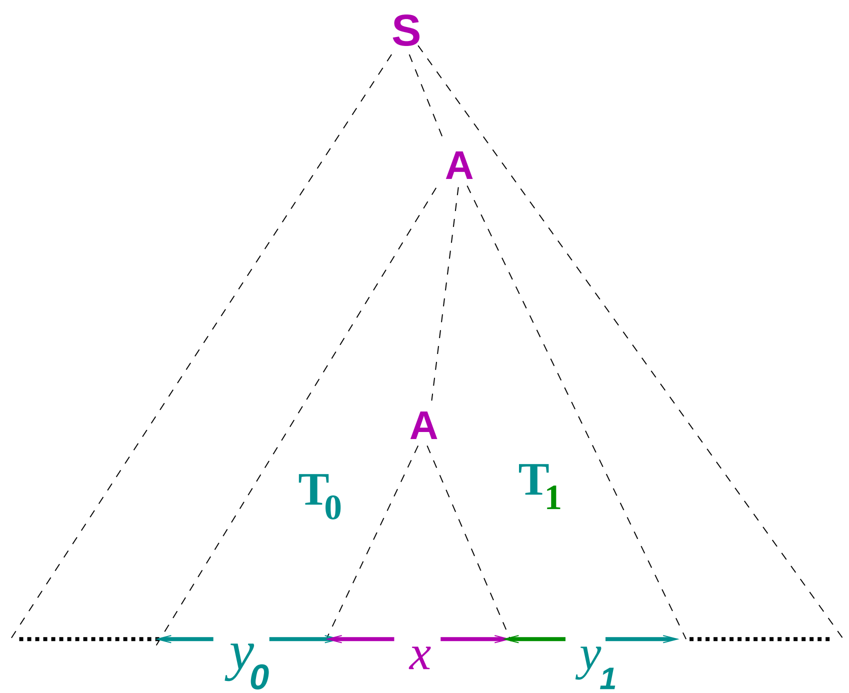
Dual-Clipping for CFLs

- The portions of the parse-tree generated by the upper **A**, but not the lower one, can be "clipped-off" the tree:



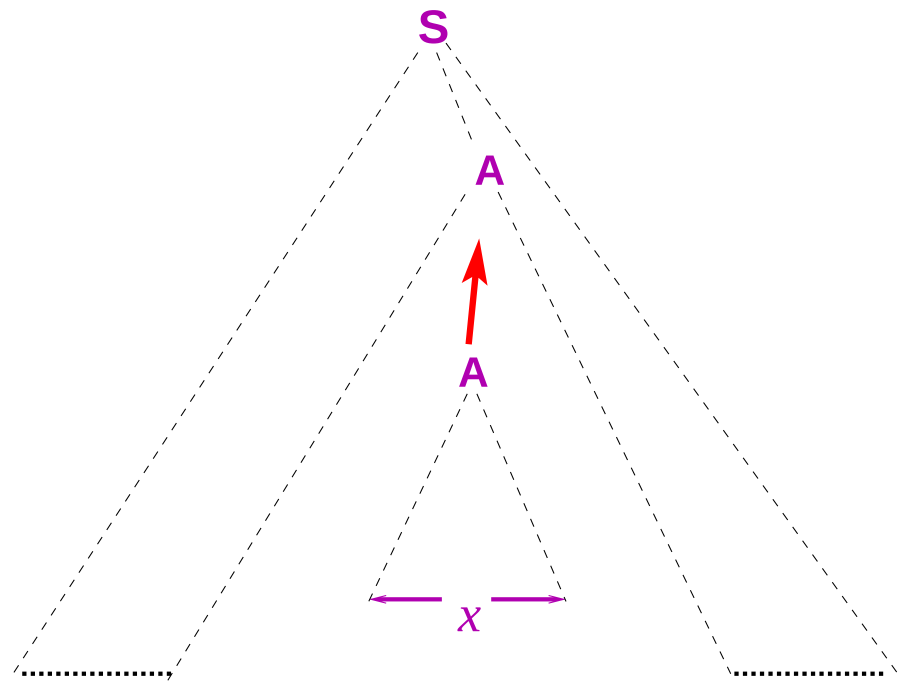
Dual-Clipping for CFLs

- The portion generated from the lower **A** remains:



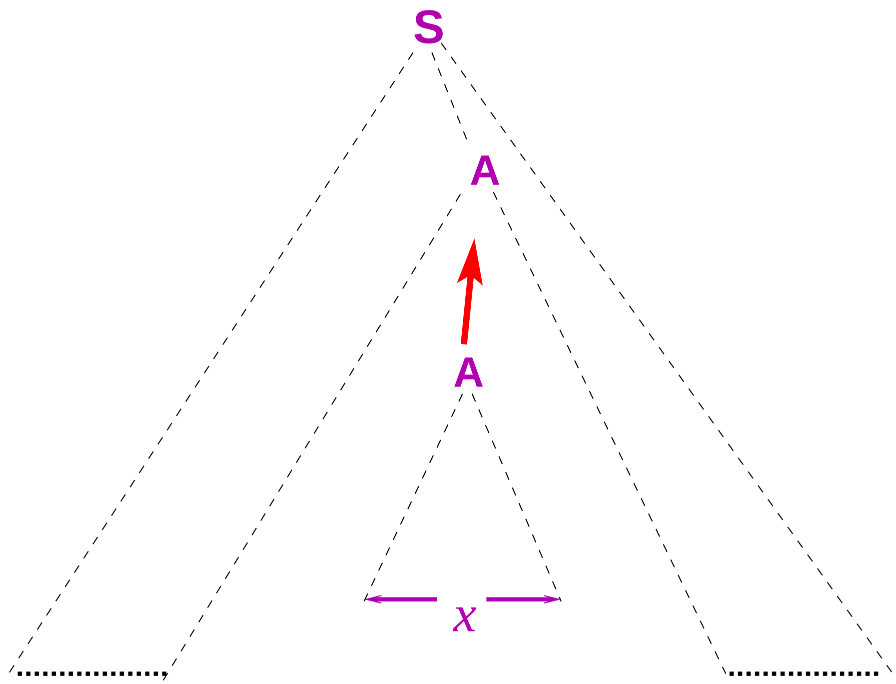
Dual-Clipping for CFLs

- The lower **A** can be identified with the upper one, by lifting the subtree it generates:



Dual-Clipping for CFLs

- The lower **A** can be identified with the upper one, by lifting the subtree it generates:



Dual-clipping: The framework

- **Dual-clipping Theorem for CFLs** (informal summary)

  If L is a CFL then

  every sufficiently long $w \in L$ has two disjoint substrings,
  not both empty, and not too far apart,
  that can be clipped off w to yield a string $w' \in L$.

Dual-clipping: The framework

- **Dual-clipping Theorem for CFLs** (informal summary)

  If L is a CFL then

  every sufficiently long $w \in L$ has two disjoint substrings,
  not both empty, and not too far apart,
  that can be clipped off w to yield a string $w' \in L$.

- Core idea: variable repeating on a branch.

Dual-clipping: The framework

- **Dual-clipping Theorem for CFLs** (informal summary)

  If L is a CFL then

  every sufficiently long $w \in L$ has two disjoint substrings,
  not both empty, and not too far apart,
  that can be clipped off w to yield a string $w' \in L$.

- Core idea: variable repeating on a branch.

- We'll also need to

  1. Give conditions that guarantee such a repetition

Dual-clipping: The framework

- **Dual-clipping Theorem for CFLs** (informal summary)

  If L is a CFL then

  every sufficiently long $w \in L$ has two disjoint substrings,
  not both empty, and not too far apart,
  that can be clipped off w to yield a string $w' \in L$.

- Core idea: variable repeating on a branch.

- We'll also need to

  1. Give conditions that guarantee such a repetition
  2. Ensure that the clipping obtained is non-empty

Dual-clipping: The framework

- **Dual-clipping Theorem for CFLs** (informal summary)

  If L is a CFL then

  every sufficiently long $w \in L$ has two disjoint substrings,
  not both empty, and not too far apart,
  that can be clipped off w to yield a string $w' \in L$.

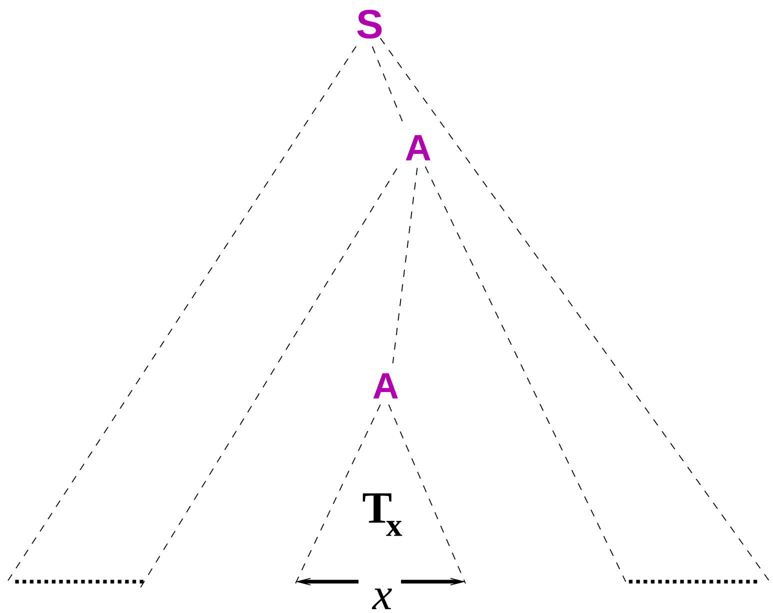
- Core idea: variable repeating on a branch.

- We'll also need to

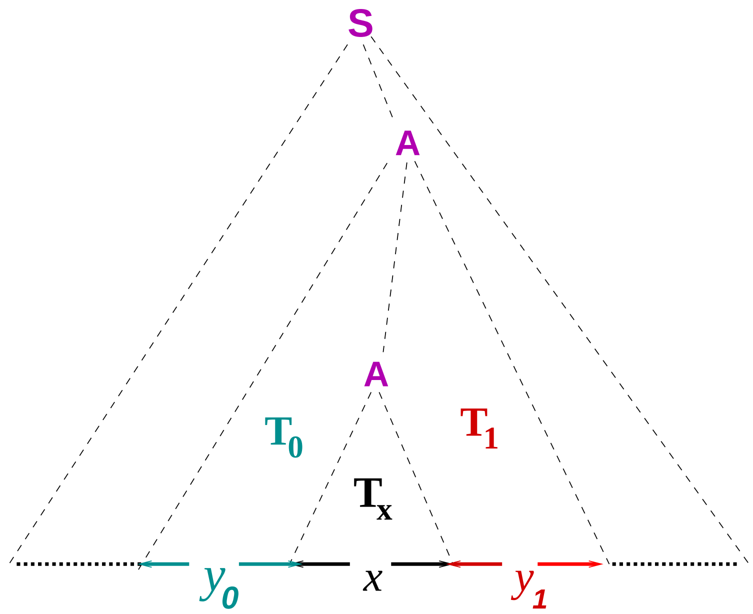
  1. Give conditions that guarantee such a repetition
  2. Ensure that the clipping obtained is non-empty
  3. Obtain two clipped substrings that are "not too far apart".

A repeated variable on a branch

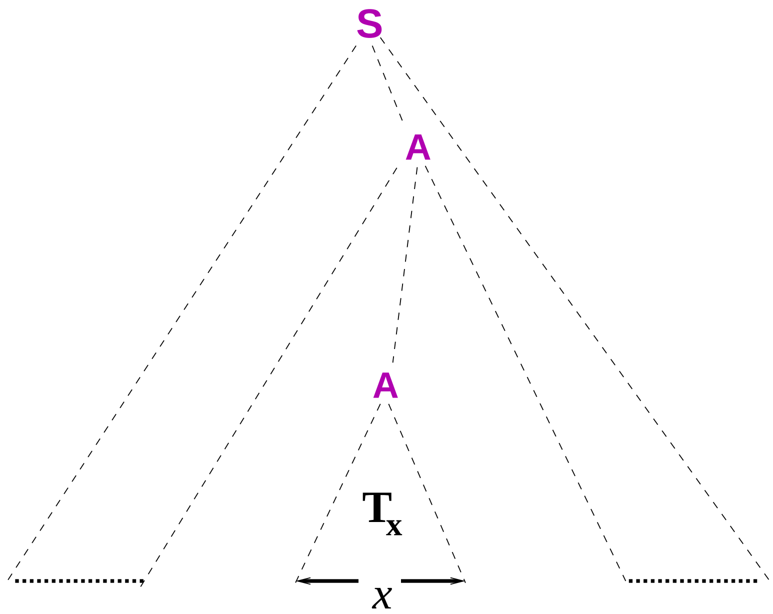
- Suppose T is a parse-tree of a CFG G for w with variable A repeating on a branch.



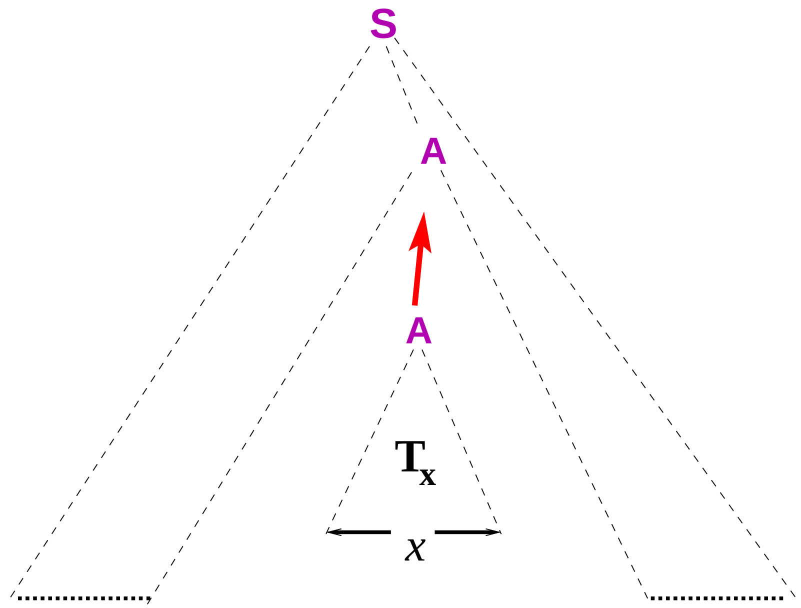
- The lower occurrence of A generates a substring x.



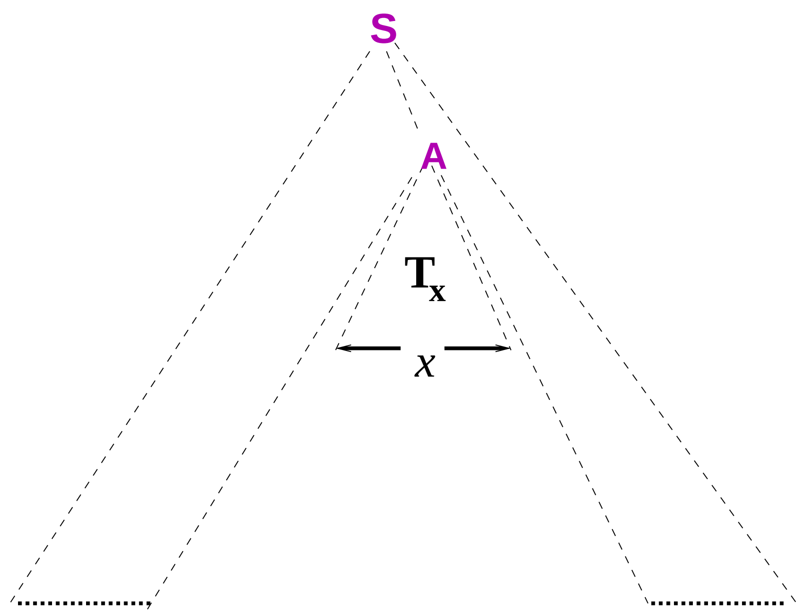
- The upper occurrence of A generates a substring $y_0 \, x \, y_1$.



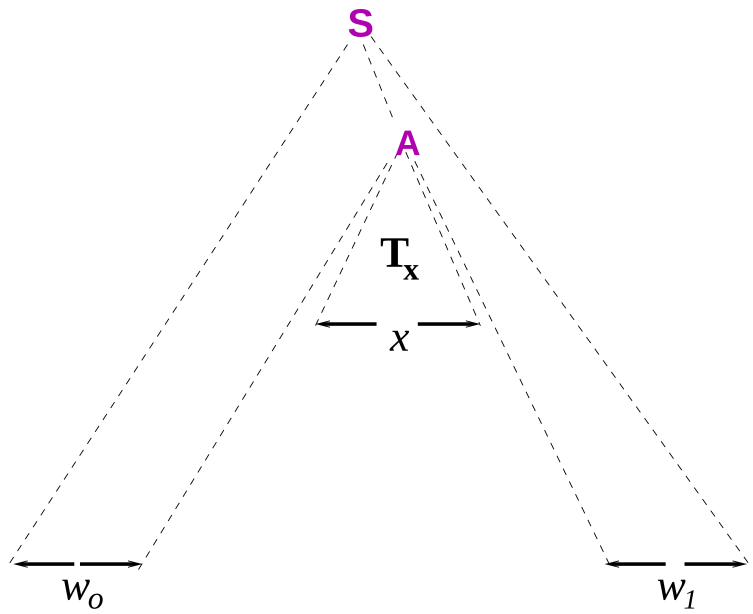
- Eliminating y_0 and y_1 yields a parse-tree except for the branch-segment between the two occurrences of A.



- So lifting the derivation from the lower occurrence of A ...



- ... results in a parse-tree for the input string with the substrings y_0 and y_1 clipped off.



- Naming the "outer" substrings of the input w_0 and w_1 , the input w is $w_0 \cdot y_0 \cdot x \cdot y_1 \cdots w_1$ for some w_0, w_1, and the resulting (clipped) string, $w_0 \cdot x \cdot w_1,$ is also in L.

Ensuring a repeated variable

- Let m be the number of variables of G .
- So there are at least $m+1$ variables on the branch for just m different variables in G.
- Some variable must be repeating!

Deriving a long string requires repetition

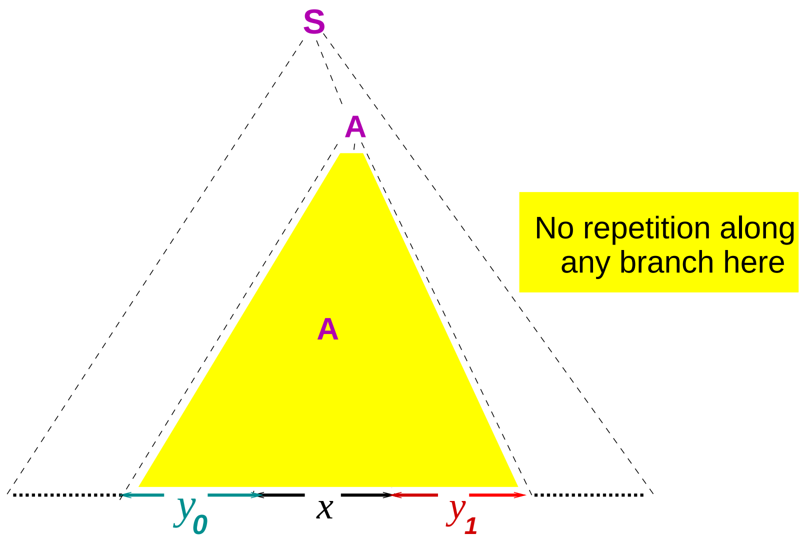
- Say that a production $X \to \sigma_1 \cdots \sigma_\ell$ has ***length*** ℓ and that the $\boxed{\textit{degree}}$ of a grammar is the maximal length of its productions.
- A binary tree of height h has $\leqslant 2^h$ leaves.
  Generally, a tree of degree d has $\leqslant d^h$ leaves.
- For a grammar of degree d and m variables
  any string with a parse-tree of height $\leqslant m$ is d^m.
- So a parse-tree for a string of length $> d^m$ must have a branch with $> m$ variables, which therefore has a variable repeating.

Ensuring non-vacuous clipping

- What if the clipped y_0, y_1 are both empty?
- Then we obtained a smaller parse-tree for w !
- If we just start with a parse-tree of G for w
  with a minimal number of nodes (no smaller parse-tree for w)
  then at least one of y_0, y_1 is non-empty.

Bounding $y_0 \cdot x \cdot y_1$

- Claim: There must be a $y_0 \cdot x \cdot y_1$ of length $\leqslant d^m$.
- Take a lower-most pair of a variable repeating:
  then no nonterminal repeats on a branch under the upper A.



- Then $|y_0 \cdot x \cdot y_1| \leqslant k$.

Dual-clipping Theorem for CFLs (Formal statement)

- ***Theorem.*** Let G be a CFG over Σ with m variables and of degree d (= all targets of length $\leqslant d$).
  - If $w \in \mathcal{L}(G)$ has length $\geqslant k = d^m$
  - then it has a substring p of length $\leqslant k$, with disjoint substrings y_0, y_1 not both empty, s.t. w' obtained from w by removing y_0 and y_1 is also in L.

Dual-clipping Theorem for CFLs (Formal statement)

- ***Theorem.*** Let G be a CFG over Σ with m variables and of degree d (= all targets of length $\leqslant d$).
  - If $w \in \mathcal{L}(G)$ has length $\geqslant k = d^m$
  - then it has a substring p of length $\leqslant k$, with disjoint substrings y_0, y_1 not both empty, s.t. w' obtained from w by removing y_0 and y_1 is also in L.

- Stated explicitly:
  w has some partition $w = w_0 \cdot y_0 \cdot x \cdot y_1 \cdot w_1$,
  with y_0, y_1 not both empty and $|y_0 \cdot x \cdot y_1| \leqslant k$,
  s.t. $w_0 \cdot x \cdot w_1 \in L$.

A Dual-clipping Property

- We rephrase the Dual-clipping Theorem in terms of a language property.
- Say that a language L has the **_Dual-clipping Property_** if
  there is a k such that
  every $w \in L$ of length $\geqslant k$
  has a substring $y_0 \cdot x \cdot y_1$ of length $\leqslant k$ with $y_0 y_1 \neq \varepsilon$,
  for which the string w' obtained from w
  by removing y_0 and y_1 is also in L.

A Dual-clipping Property

- We rephrase the Dual-clipping Theorem in terms of a language property.
- Say that a language L has the $\boxed{\textit{Dual-clipping Property}}$ if
  there is a k such that
  every $w \in L$ of length $\geqslant k$
  has a substring $y_0 \cdot x \cdot y_1$ of length $\leqslant k$ with $y_0 y_1 \neq \varepsilon$,
  for which the string w' obtained from w
  by removing y_0 and y_1 is also in L.
- The Dual-Clipping Theorem for CFLs states that
  every CFL has the Dual-Clipping Property.
- Consequently, if a language L fails this property,
  then it is not CF.

Failing Dual-Clipping

- L **fails** the Dual-clipping Property when
  - *For every* k *there is a* $w \in L$ *of length* $\geqslant k$ *s.t.* *for every substring* $y_0 \cdot x \cdot h_1$ *of* w *of length* $\leqslant k$ *with* $y_0 y_1 \neq \varepsilon$ *the string* w' *obtained from* w *by removing* y_0 *and* y_1 *is not in* L.

Example: an-bn-cn

- Let $L = \{\mathrm{a}^n \mathrm{b}^n \mathrm{c}^n \mid n \geqslant 0\}$.
  We show that L is not CF.

Example: an-bn-cn

- Let $L = \{\mathrm{a}^n\mathrm{b}^n\mathrm{c}^n \mid n \geqslant 0\}$.
  We show that L is not CF.
- Let $L = \mathcal{L}(G)$, G a CFG with clipping constant k .

Example: an-bn-cn

- Let $L = \{\mathrm{a}^n\mathrm{b}^n\mathrm{c}^n \mid n \geqslant 0\}$.
  We show that L is not CF.
- Let $L = \mathcal{L}(G)$, G a CFG with clipping constant k .
- Take $w = \mathrm{a}^k\mathrm{b}^k\mathrm{c}^k \in L$. We have $w \in L$ with $|w| \geqslant k$.

Example: an-bn-cn

- Let $L = \{\mathrm{a}^n\mathrm{b}^n\mathrm{c}^n \mid n \geqslant 0\}$.
  We show that L is not CF.
- Let $L = \mathcal{L}(G)$, G a CFG with clipping constant k .
- Take $w = \mathrm{a}^k\mathrm{b}^k\mathrm{c}^k \in L$. We have $w \in L$ with $|w| \geqslant k$.
- By Dual-Clipping we can clip off w some y_0, y_1
  within a k-long substring p of w , getting $w' \in L$.

Example: an-bn-cn

- Let $L = \{\mathrm{a}^n\mathrm{b}^n\mathrm{c}^n \mid n \geqslant 0\}$.
  We show that L is not CF.
- Let $L = \mathcal{L}(G)$, G a CFG with clipping constant k .
- Take $w = \mathrm{a}^k\mathrm{b}^k\mathrm{c}^k \in L$. We have $w \in L$ with $|w| \geqslant k$.
- By Dual-Clipping we can clip off w some y_0, y_1
  within a k-long substring p of w , getting $w' \in L$.
- But this is impossible:
  Since $|p| \leqslant k$ it cannot have more than 2 of the 3 letters,
  so w' cannot have an equal number of letters.

Example: an-bn-cn

- Let $L = \{\mathrm{a}^n\mathrm{b}^n\mathrm{c}^n \mid n \geqslant 0\}$.
  We show that L is not CF.
- Let $L = \mathcal{L}(G)$, G a CFG with clipping constant k .
- Take $w = \mathrm{a}^k\mathrm{b}^k\mathrm{c}^k \in L$. We have $w \in L$ with $|w| \geqslant k$.
- By Dual-Clipping we can clip off w some y_0, y_1
  within a k-long substring p of w , getting $w' \in L$.
- But this is impossible:
  Since $|p| \leqslant k$ it cannot have more than 2 of the 3 letters,
  so w' cannot have an equal number of letters.
- Conclusion: L is not CF.

Steps in the contrarian game

Note the order of choices in this "contrarian" proof by contradiction:

1. G is ***given to us,*** with its clipping constant.

Steps in the contrarian game

Note the order of choices in this “contrarian” proof by contradiction:

1. G is ***given to us,*** with its clipping constant.
2. ***We*** can choose a $w \in L$ of length $\geqslant k$.

Steps in the contrarian game

Note the order of choices in this “contrarian” proof by contradiction:

1. G is ***given to us,*** with its clipping constant.
2. ***We*** can choose a $w \in L$ of length $\geqslant k$.
3. Substring p and its factorization $py_0 \cdot x \cdot y_1$ are unknown, i.e. ***given to us.***

Steps in the contrarian game

Note the order of choices in this “contrarian” proof by contradiction:

1. G is ***given to us,*** with its clipping constant.
2. ***We*** can choose a $w \in L$ of length $\geqslant k$.
3. Substring p and its factorization $py_0 \cdot x \cdot y_1$ are unknown,
   i.e. ***given to us.***
4. We must show that ***whatever they are***
   the resulting string w' is $\notin L$.

Same proof articulated as failures

We can articulate proofs like this directly by showing failure of Dual-Clipping,

Same proof articulated as failures

We can articulate proofs like this directly by showing failure of Dual-Clipping,

- Given to us an unknown $k > 0$,
  we choose $w = \mathrm{a}^k \mathrm{b}^k \mathrm{c}^k$. We have $w \in L$ and $|w| \geqslant k$.

Same proof articulated as failures

We can articulate proofs like this directly by showing failure of Dual-Clipping,

- Given to us an unknown $k > 0$,
  we choose $w = \mathtt{a}^k\mathtt{b}^k\mathtt{c}^k$. We have $w \in L$ and $|w| \geqslant k$.
- Then given to us that an unknown substring
  $p = y_0 \cdot x \cdot y_1$ of length $\leqslant k$
  we observe that it can have at most two of $\mathtt{a}, \mathtt{b}, \mathtt{c}$.

Same proof articulated as failures

We can articulate proofs like this directly by showing failure of Dual-Clipping,

- Given to us an unknown $k > 0$,
  we choose $w = \mathtt{a}^k\mathtt{b}^k\mathtt{c}^k$. We have $w \in L$ and $|w| \geqslant k$.
- Then given to us that an unknown substring
  $p = y_0 \cdot x \cdot y_1$ of length $\leqslant k$
  we observe that it can have at most two of $\mathtt{a}, \mathtt{b}, \mathtt{c}$.
- So removing y_0 and y_1 yields a string not in L.

Same proof articulated as failures

We can articulate proofs like this directly by showing failure of Dual-Clipping,

- Given to us an unknown $k > 0$,
  we choose $w = \mathtt{a}^k\mathtt{b}^k\mathtt{c}^k$. We have $w \in L$ and $|w| \geqslant k$.
- Then given to us that an unknown substring
  $p = y_0 \cdot x \cdot y_1$ of length $\leqslant k$
  we observe that it can have at most two of $\mathtt{a}, \mathtt{b}, \mathtt{c}$.
- So removing y_0 and y_1 yields a string not in L.
- Since L fails the Dual-clipping Property, it is not CF.

The intersection of CFLs

Now that we have a non-CF language,
we can show that the intersection of CFL ***need not be CF!!***

- $$\begin{aligned} L_{ab} &= \{\mathsf{a}^n\mathsf{b}^n\mathsf{c}^k \mid n,k \geqslant 0\} \text{ is CF} \\ L_{bc} &= \{\mathsf{a}^k\mathsf{b}^n\mathsf{c}^n \mid n,k \geqslant 0\} \text{ is CF} \end{aligned}$$
- But their intersection
$$L_{ab} \cap L_{bc} = \{\mathsf{a}^n\mathsf{b}^n\mathsf{c}^n \mid n \geqslant 0\}$$
is not CF.

The complement of a CFL

The complement of a CFL ***need not be CF***.

- Reason: The collection of CFLs is closed under union.
  If it were closed under complement then it would be closed under intersection.
- $-(A \cap B) = -A \cup -B$ so $A \cap B = -(-A \cup -B)$
- ***Specific example:*** The Mahi-mahi Language is not CF.
  But its complement is!

Example: Alternating equals

- $\{\mathtt{a}^i\mathtt{b}^j\mathtt{c}^i \mid i,j \geqslant 0\}$ is CF. So is $\{\mathtt{a}^i\mathtt{b}^j\mathtt{c}^j\mathtt{d}^i \mid i,j \geqslant 0\}$

Example: Alternating equals

- $\{\mathtt{a}^i\mathtt{b}^j\mathtt{c}^i \mid i,j \geqslant 0\}$ is CF. So is $\{\mathtt{a}^i\mathtt{b}^j\mathtt{c}^j\mathtt{d}^i \mid i,j \geqslant 0\}$
- But $L = \{\mathtt{a}^i\mathtt{b}^j\mathtt{c}^i\mathtt{d}^j \mid i,j \geqslant 0\}$ is not.

Example: Alternating equals

- $\{\mathrm{a}^i\mathrm{b}^j\mathrm{c}^i \mid i,j \geqslant 0\}$ is CF. So is $\{\mathrm{a}^i\mathrm{b}^j\mathrm{c}^j\mathrm{d}^i \mid i,j \geqslant 0\}$
- But $L = \{\mathrm{a}^i\mathrm{b}^j\mathrm{c}^i\mathrm{d}^j \mid i,j \geqslant 0\}$ is not.
- Given $k > 0$ take $w = \mathrm{a}^k\mathrm{b}^k\mathrm{c}^k\mathrm{d}^k \in L$. $w \in L,\ |w| \geqslant k$.

Example: Alternating equals

- $\{a^ib^jc^i \mid i,j \geqslant 0\}$ is CF. So is $\{a^ib^jc^jd^i \mid i,j \geqslant 0\}$
- But $L = \{a^ib^jc^id^j \mid i,j \geqslant 0\}$ is not.
- Given $k > 0$ take $w = a^kb^kc^kd^k \in L$. $w \in L,\ |w| \geqslant k$.
- If $p = y_0 \cdot x \cdot y_1$ is a substring, $y_1y_1z \neq \varepsilon$
  let w' be obtained from w by removing y_0, y_1 .

Example: Alternating equals

- $\{a^i b^j c^i \mid i, j \geqslant 0\}$ is CF. So is $\{a^i b^j c^j d^i \mid i, j \geqslant 0\}$
- But $L = \{a^i b^j c^i d^j \mid i, j \geqslant 0\}$ is not.
- Given $k > 0$ take $w = a^k b^k c^k d^k \in L$. $w \in L$, $|w| \geqslant k$.
- If $p = y_0 \cdot x \cdot y_1$ is a substring, $y_1 y_1 z \neq \varepsilon$
  let w' be obtained from w by removing y_0, y_1 .
- Since p can span at most two adjacent blocks,
  removing y_0, y_1 deletes some letter (`a`,`b`,`c`, or `d`)
  without deleting any corresponding one (`c`, `d`, `a`, or `b`, respectively).
- So $w' \notin L$.

Example: Alternating equals

- $\{\mathrm{a}^i\mathrm{b}^j\mathrm{c}^i \mid i,j \geqslant 0\}$ is CF. So is $\{\mathrm{a}^i\mathrm{b}^j\mathrm{c}^j\mathrm{d}^i \mid i,j \geqslant 0\}$
- But $L = \{\mathrm{a}^i\mathrm{b}^j\mathrm{c}^i\mathrm{d}^j \mid i,j \geqslant 0\}$ is not.
- Given $k > 0$ take $w = \mathrm{a}^k\mathrm{b}^k\mathrm{c}^k\mathrm{d}^k \in L$. $w \in L$, $|w| \geqslant k$.
- If $p = y_0 \cdot x \cdot y_1$ is a substring, $y_1 y_1 z \neq \varepsilon$
  let w' be obtained from w by removing y_0, y_1 .
- Since p can span at most two adjacent blocks,
  removing y_0, y_1 deletes some letter (`a`,`b`,`c`, or `d`)
  without deleting any corresponding one (`c`, `d`, `a`, or `b`, respectively).
- So $w' \notin L$.
- L fails the dual-clipping property, and cannot be CF.

The mahi-mahi language

- We already proved that the language $L = \{r \cdot r \mid r \in \{\mathtt{a}, \mathtt{b}\}^*\}$ is not regular. We now prove that it is not even CF.
- Let L is generated by a CFG G with clipping constant $k = d^m$.
- Try $w = \mathtt{a}^k \mathtt{b} \mathtt{a}^k \mathtt{b}$.
- *Not working!* since we might have
  y_0 in the first block of $\mathtt{a}$'s
  and y_1 in the second!
- Solution: Push the two blocks apart: $w = \mathtt{a}^k \mathtt{b}^k \mathtt{a}^k \mathtt{b}^k \in L$.
- By Dual-clipping, w has a substring of length $\leqslant k$
  of the form $y_0 \cdot x \cdot y_1$ with $y_0 y_1 \neq \varepsilon$
  s.t. w with y_0 and y_1 removed is still in L.

- Cases for $p = y_0 \cdot x \cdot y_1$:

  1. p in first half of w.
     Then $w' = \mathtt{a}^i\mathtt{b}^j\mathtt{a}^k\mathtt{b}^k$ with $i + j < 2k$.
     First half of w' ends with $\mathtt{a}$, second with $\mathtt{b}$. $w' \notin L$!
  2. p in second half of w.
     Then $w' = \mathtt{a}^k\mathtt{b}^k\mathtt{a}^i\mathtt{b}^j$ with $i + j < 2k$.
     First half of w' starts with $\mathtt{a}$, second with $\mathtt{b}$.
     $w' \notin L$!
  3. p in $\mathtt{b}^k\mathtt{a}^k$. Then $w' = \mathtt{a}^k\mathtt{b}^i\mathtt{a}^j\mathtt{b}^k$ with $i + j < 2k$.
     First half of w' has more $\mathtt{a}$'s than the second,
     or second half has more $\mathtt{b}$'s than the first (or both).
     $w' \notin L$!

- In any case $w' \notin L$. So L is not CF.

SPECIAL TYPES OF CFGs

Regular grammars

- Regular languages are generated by ***regular grammars,*** a special type of CFGs.
- $\boxed{\textit{Regular grammars}}$ have only productions of the forms

$$
\begin{aligned}
A &\rightarrow \varepsilon \\
A &\rightarrow \sigma \qquad (\sigma \in \Sigma) \\
A &\rightarrow \sigma B
\end{aligned}
$$

- Simulate a DFA $M = (\Sigma, Q, s, A, \delta)$ by a CFG G:
  - G's nonterminals are the states of M (let's underline them).
  - Its initial nonterminal is $\underline{s}$.

– A transition-rule $\underline{q} \xrightarrow{\sigma} \underline{p}$
becomes the production $\underline{q} \rightarrow \sigma\underline{p}$.

– Acceptance by M becomes releasing an output by G:
$\underline{a} \rightarrow \varepsilon$ for each $a \in A$.

- Above are ***right-regular*** (aka right-***linear***) grammars.
Left-regular grammars have $A \rightarrow B\sigma$ instead.

Expanding grammars

- We'll say that a CFG is ***spreading*** if its productions are all of one of two forms.
  - ***Terminal:*** $A \to \sigma$, or
  - ***Spread:*** $A \to z$ where $|z| \geqslant 2$.

- ***Observation.*** Suppose G is a spreading CFG,
  nd D is a derivation in G of a string $x \in \Gamma^*$.
  Then D has at most $c(x) = |x| + \#_\Sigma(x)$ steps,
  where $\#_\Sigma(x)$ is the number of terminals in x.

- In particular, if $x \in \Sigma$ (i.e. all terminals), then $c(x) = 2 \cdot |x|$.

- Example: Let G be the spreading CFG $S \to QS \mid \mathtt{b},\ Q \to \mathtt{a}$.
  Consider the derivation $S \Rightarrow QS \Rightarrow QQS \Rightarrow QQ\mathtt{b} \to \mathtt{a}Q\mathtt{b}$, which has 4 steps.
  Here x is $\mathtt{a}Q\mathtt{b}$, a string of length 3 with 2 non-terminals.
  And indeed $c(x) = |x| + \#_\Sigma(x) = 3{+}2 = 5 \geqslant 4$.

- Observation's proof by induction on the length of D: For spreading CFG G, if $x \Rightarrow_G y$ then $c(y) < c(x)$.

Conversion to spreading CFGs

- ***Lemma.*** Every CFG G that does not generate ε can be converted into an equivalent spreading CFG.
- Need to eliminate productions of the form
  - $Q \to \varepsilon$ (ε productions) and
  - $Q \to R$ (Stagnant productions)
- Let's eliminate first all ε-productions.

Eliminating ε-productions

- To drop $Q \rightarrow \varepsilon$ must be compensated so as to preserve equivalence w/ G.
- For each production $R \rightarrow z$ and each z' obtained from z by deleting some Q's, add the production $R \rightarrow z'$.
- Example 1: To $R \rightarrow Q\mathtt{a}Q$ add $R \rightarrow \mathtt{a}Q$, $R \rightarrow Q\mathtt{a}$ and $R \rightarrow Q$.
- Example 2: To $R \rightarrow Q$ add $R \rightarrow \varepsilon$
- We might add new ε-productions and stagnant-productions.

- But for now we don't worry about stagnant productions, and we won't need to return to $Q \to \varepsilon$.

  So process repeated $\leqslant m =$ number of non-terminals.

Eliminating stagnant-productions

- To drop $Q \to P$ must compensate to preserve equival with G.
- ε -productions no longer present.
- For each production $P \to z$ add $Q \to z$.
- Example 1: To $P \to PQ$ add $Q \to PQ$.
- Example 2: to $P \to R$ add $Q \to R$.
- No ε-prods are generated: none to start with.
- New stagnant prods are possible, but we wont return to $Q \to P$. So process repeated $\leqslant m^2$ times.

Decision algorithm for spreading grammars

- ***Theorem.*** If G is an spreading grammar,
  then there is an algorithm that for input $w \in \Sigma^*$
  tells whether w is derived in G.
- Algorithm: Go through all the possible derivations of length $\leqslant 2 \cdot |w|$
  and check if one of them yields w.
- Little problem: This takes time exponential in the size of the input.

Chomsky grammars

- A $\boxed{\textit{Chomsky grammar}}$ is a special type of spreading grammar, using only very special Spread productions.

- Only productions allowed:
  - ***Terminal:*** $A \to \sigma$
  - ***Split:*** $A \to BC$
- Theorem: Every CFG G not generating ε can be converted to an equivalent Chomsky grammar.
- We only need to show this for spreading G.
- We'll use fresh non-terminals.

Spreads converted to Splits

- First covert to a grammar with no terminals in Spreads:
- Replace a Spread production like $Q \to \mathtt{ab}R$ by three: $Q \to \hat{\mathtt{a}}\hat{\mathtt{b}}R$, $\hat{\mathtt{a}} \to \mathtt{a}$ and $\hat{\mathtt{b}} \to \mathtt{b}$.

  Here $\hat{\sigma}$ is a fresh nonterminal ("promising to convert to σ")

  - Now eliminate spreads with long targets:
  - Replace a Spread like $Q \to PRT$ by

    $Q \to PM_{RT}$ and $M_{RT} \to RT$.

  Here M_{RT} is a fresh nonterminal ("promising to convert to RT")

- So $Q \to PRTU$ is replaced by

$$M_{RTU} \to RM_{TU} \quad \text{and} \quad M_{TU} \to TU \quad .$$

- This completes the proof Chomsky's Theorem.

What about ε ?

- If a CFL L contains ε then $L - \{\varepsilon\}$ is also CF (we'll see).
- Now $L - \{\varepsilon\}$ is generated by a Chomsky grammar G.
- Use a fresh start on-terminal S' ,
  and add to G the production $S' \to \varepsilon \mid S$,
  obtaining an "almost-Chomsky" grammar G' generating L.
- G' does not have S' in any target!

A memoization algorithm

- Given a Chomsky grammar C over Σ,
  we give a cubic-time memoization algorithm $\mathcal{A}$ to decide $\mathcal{L}(G)$.
- That is, our algorithm decides, given $w = \sigma_1 \cdots \sigma_n \in \Sigma^*$,
  whether G generates w.
  This is known as the ***Cocke-Younger-Kasami (CYK)*** Algorithm.
- Actual credits:
  - Itiro Sakai (1961)
  - Tadao Kasami (1965)
  - Daniel Younger (1967)

- John Cocke and Jacob Schwartz (1970)

The CYK Algorithm

- $\mathcal{A}$ generates lists $\ell_0 \ldots \ell_n$,
  ℓ_i is the list of pairs (A, u), with $|u| = i$, $A \Rightarrow_G^* u$.
- ℓ_0 is $\{\varepsilon\}$ if Empty is a production of G, and is $\emptyset$ o.w.
- ℓ_1 is given by the Terminal productions of G.
- Obtain ℓ_i for $i \geqslant 2$:
  for each substring u, with $|u| = i$ ($< n$ such strings)
  and each split $u = x \cdot y$ ($< n$ such splits)
  and each production $A \to BC$ (constant number of such productions)
  check whether $(B, x) \in \ell_{|x|}$, $(C, y) \in \ell_{|y|}$.
  (constant time assuming random access into the lists).

- $w \in \mathcal{L}(G)$ iff $(S, w) \in \ell_n$.

Example of CYK

Generating $\mathtt{a}^p\mathtt{c}\mathtt{b}^{p+q}\mathtt{c}\mathtt{a}^q$:

$$
\begin{aligned}
S &\rightarrow LR \\
L &\rightarrow \mathtt{a}L\mathtt{b} \mid \mathtt{c} \\
R &\rightarrow \mathtt{b}R\mathtt{a} \mid \mathtt{c}
\end{aligned}
$$

An equivalent Chomsky grammar:

$$
\begin{array}{lll}
S \rightarrow LR & L \rightarrow AM \mid \mathtt{c} & R \rightarrow BN \mid \mathtt{c} & A \rightarrow \mathtt{a} \\
 & M \rightarrow LB & N \rightarrow RA & B \rightarrow \mathtt{b}
\end{array}
$$

Decide whether `acbbca` is generated.

$$\ell_1: \; A \Rightarrow \texttt{a}, \quad B \Rightarrow \texttt{b}, \quad L \Rightarrow \texttt{c}, \quad R \Rightarrow \texttt{c}$$

$$\ell_2: \; M \rightarrow LB \Rightarrow^* \texttt{cb}$$

$$N \rightarrow RA \Rightarrow^* \texttt{ca}$$

$$\ell_3: \; L \rightarrow AM \Rightarrow^* \texttt{acb}$$

$$R \rightarrow BN \Rightarrow^* \texttt{bca}$$

$$\ell_4: \; M \rightarrow LB \Rightarrow^* \texttt{acbb}$$

$$\ell_5: \; \emptyset$$

$$\ell_6: \; S \rightarrow LR \Rightarrow^* \mathtt{\mathbf{acbbca}}$$

EMPTINESS: *Another application of Chomsky*

- Design an algorithm that, given a device M,
  determine whether $\mathcal{L}(M) = \emptyset$.
- EMPTINESS for automata is decidable:
  For automaton with k states check all w with $|w| \leqslant k$.
- This is exponential time. Can we do better?
- Generate in linear time states that "accept something".
- For CFGs we get an Emptiness algorithm by Chomsky grammars:
  Generate variables that "generate something".
- The construction is virtually the same as for CYK.

NONDETERMINISTIC STACK ACCEPTORS

A missing computation model

generative	**REG**	
operational	**DFA**	

DFA = Deterministic Finite Acceptor

A missing computation model

generative	**REG**	
operational	**NFA**	

NFA = Nondeterministic Finite Acceptor

A missing computation model

generative	**REG**	**CFL**
operational	**NFA**	???

A missing computation model

generative	**REG**	**CFL**
operational	**NFA**	**PDA**

PDA = Push-Down Automata, i.e. nondeterministic finite acceptor

A missing computation model

generative	**REG**	**CFL**
operational	**NFA**	

Why this matters

- The primary computational characterization of:
  - regular languages: by a machine model (DFA)
  - context-free languages: by a symbolic model (CFG)
- But *parsing* for CFLs is important,
  and needs a machine model.
- Next: a characterization of CFLs by a machine model.
- Unfortunately, non-determinism is essential here.

Cautious extension of memory

- Approach: extend automata with an external memory.
- Limiting the space used gives us LBA (and other).

- This turns out to be too powerful.
- Alternative: limit external memory to “single-use”.

Stacks

- A stack is read from the top!
- It is unbounded (like the Turing string)
- But access destroys stored information (single use).

Traditional stack operations

- Push a symbol: $w \mapsto \sigma w$
- Pop a symbol: $\sigma w \mapsto w$
- Represent a stack by a string:
  `edcba` is the stack with `e` at the top, `a` at the bottom.
- The empty string ε represents the empty stack.

A combined stack-operation

- Generalize *push* to a string v_0 :

  $w \mapsto v_0 \cdot w$

- And *pop* to a conditional string-pop u_0:

  $u_0 \cdot w \mapsto w$

  If the top of the stack matches u_0 *then pop that top.*

- Combined to a single operation of Replacing a Top segment of stack:

  $u_0 \cdot x \quad \mapsto \quad v_0 \cdot x$

- Meaning:

  if u_0 matches a top portion of the stack

  then replace it by v_0 **else skip**

- Notation: $u_0 \to v_0$.
- Examples:

$$\varepsilon \to 2 \qquad 2 \to \varepsilon \qquad 1 \to 2 \qquad 1 \to 23$$

$$12 \to 221 \qquad \varepsilon \to 23 \qquad 12 \to \varepsilon$$

A $\boxed{\textit{stack automaton (PDA)}}$ over an alphabet Σ
is a device $M = (\Sigma, Q, s, A, \Gamma, \Delta)$ where

- Q is a set, dubbed ***states***
- $s \in Q$ is distinguished state, dubbed ***initial*** state
- $A \subseteq Q$, the set of ***accepting*** states
- $\Gamma \supseteq \Sigma$ is the ***extended alphabet***
- Δ is a finite set of ***transition rules*** of the form

$q \xrightarrow{\sigma\,(\beta \to \gamma)} p$ where

$$q, p \in Q$$

$$\sigma \in \Sigma_\epsilon = \Sigma \cup \{\varepsilon\}$$

$$\beta, \gamma \in \Gamma^*$$

Using stack as memory: an example

- Task: recognize strings $\mathtt{a}^n\mathtt{b}^n$ ($n \geqslant 1$).
- Initially the stack is empty.
- Phase 1:
  As input is read, $\mathtt{a}$'s are pushed on the stack.
- Phase 2:
  When $\mathtt{b}$ is encountered, start popping $\mathtt{a}$'s.
- Termination:
  Input accepted if stack is empty when input scan completed.

Using a bottom-marker

- Our PDAs do not recognize an empty stack
  (some varieties of PDAs do!)
- The intent of an empty stack is obtained
  by reserving a symbol as bottom-of-stack marker, say \$.
- A PDA as above starts by pushing \$ on the stack,
  and accepts the input if \$ is at the top of the stack
  when completing the scan.

A PDA for $\{\mathrm{a}^n\mathrm{b}^n \mid n > 0\}$

- States: initial s, accepting f, $q =$ pushing phase, $p =$ popping phase
- Transitions:

$$s \xrightarrow{\epsilon\,(\epsilon \to \$)} q \quad \text{(push \$)}$$
$$q \xrightarrow{a\,(\epsilon \to a)} q \quad \text{(reading } a\text{'s push them)}$$
$$q \xrightarrow{b\,(a \to \epsilon)} p \quad \text{(on } b \text{ pop } a \text{ \& switch state)}$$
$$p \xrightarrow{b\,(a \to \epsilon)} p \quad \text{(reading } b\text{'s pop } a\text{'s)}$$
$$p \xrightarrow{\epsilon\,(\$ \to \epsilon)} f \quad \text{(if \$ tops stack accept)}$$

A PDA for $\{\mathtt{a}^n\mathtt{b}^n \mid n > 0\}$

- States: initial s, accepting f, $q =$ pushing phase, $p =$ popping phase
- Transitions:

$$
\begin{array}{lll}
s \xrightarrow{\epsilon\,(\epsilon \to \$)} q & & \text{(push } \$\text{)} \\
q \xrightarrow{a\,(\epsilon \to a)} q & & \text{(reading } a\text{'s push them)} \\
q \xrightarrow{b\,(a \to \epsilon)} p & & \text{(on } b \text{ pop } a \text{ \& switch state)} \\
p \xrightarrow{b\,(a \to \epsilon)} p & & \text{(reading } b\text{'s pop } a\text{'s)} \\
p \xrightarrow{\epsilon\,(\$ \to \epsilon)} f & & \text{(if \$ tops stack accept)}
\end{array}
$$

- If $\$$ is read while some $\mathtt{b}$'s unread ($\#_b > \#_a$) then reading is incomplete, so no acceptance.

A PDA for $\{\mathrm{a}^n\mathrm{b}^n \mid n > 0\}$

- States: initial s, accepting f, $q =$ pushing phase, $p =$ popping phase
- Transitions:

$$s \xrightarrow{\epsilon\,(\epsilon \to \$)} q \quad \text{(push \$)}$$
$$q \xrightarrow{a\,(\epsilon \to a)} q \quad \text{(reading } a\text{'s push them)}$$
$$q \xrightarrow{b\,(a \to \epsilon)} p \quad \text{(on } b \text{ pop } a \text{ \& switch state)}$$
$$p \xrightarrow{b\,(a \to \epsilon)} p \quad \text{(reading } b\text{'s pop } a\text{'s)}$$
$$p \xrightarrow{\epsilon\,(\$ \to \epsilon)} f \quad \text{(if \$ tops stack accept)}$$

- If popping is not completed ($\#_a > \#_b$)
  then $\$$ is not reach, so no accept state.

A PDA for $\{\mathtt{a}^n\mathtt{b}^n \mid n > 0\}$

- States: initial s, accepting f, $q =$ pushing phase, $p =$ popping phase
- Transitions:

$$s \xrightarrow{\epsilon\,(\epsilon \to \$)} q \quad \text{(push } \$\text{)}$$
$$q \xrightarrow{a\,(\epsilon \to a)} q \quad \text{(reading } a\text{'s push them)}$$
$$q \xrightarrow{b\,(a \to \epsilon)} p \quad \text{(on } b \text{ pop } a \text{ \& switch state)}$$
$$p \xrightarrow{b\,(a \to \epsilon)} p \quad \text{(reading } b\text{'s pop } a\text{'s)}$$
$$p \xrightarrow{\epsilon\,(\$ \to \epsilon)} f \quad \text{(if \$ tops stack accept)}$$

- If a `b` is followed by `a`
  then computation aborts: no production for p reading `a`.

Semantics of PDAs

- The semantics of an NFA was given by the transition mapping, i.e. the collection of single-transitions $q \xrightarrow{\sigma} p$, where $p, q \in Q$ and $\sigma \in \Sigma_\epsilon$.
- A PDA $P = (\Sigma, Q, s, a, \Gamma, \delta)$ is an NFA equipped with a stack. An ***extended-state*** of P is a pair (q, α) where $q \in Q$ and $\alpha \in \Gamma^*$.

  α is the contents of the stack, represented (from top to bottom) as a string,

Semantics of PDAs

- The semantics of an NFA was given by the transition mapping,
  i.e. the collection of single-transitions $q \xrightarrow{\sigma} p$,
  where $p, q \in Q$ and $\sigma \in \Sigma_\epsilon$.
- A PDA $P = (\Sigma, Q, s, a, \Gamma, \delta)$ is an NFA equipped with a stack. An ***extended-state*** of P is a pair (q, α) where $q \in Q$ and $\alpha \in \Gamma^*$.
  α is the contents of the stack, represented (from top to bottom) as a string,
- Transition rules $q \xrightarrow{\sigma} p$ can be extended to $\xrightarrow{w}$ for arbitrary $w \in \Sigma_\epsilon^*$:

If $q \xrightarrow{\sigma\,(\alpha \to \beta)} p$ is a transition-rule,

and $(p, \beta \cdot \gamma) \stackrel{w}{\to} (r, \eta)$ then and $(q, \alpha \cdot \gamma) \stackrel{\sigma w}{\to} (r, \eta)$.

Accepted strings and recognized languages

- An input string $w \in \Sigma^*$ is $\boxed{\textit{accepted}}$ by a PDA M if $(s, \varepsilon) \xrightarrow{w} (a, \gamma)$ for some $\gamma \in \Gamma^*$.

Accepted strings and recognized languages

- An input string $w \in \Sigma^*$ is $\boxed{\textit{accepted}}$ by a PDA M if $(s, \varepsilon) \xrightarrow{w} (a, \gamma)$ for some $\gamma \in \Gamma^*$.
- The $\boxed{\textit{language recognized}}$ by M, denoted $\mathcal{L}(M)$, is the set of strings accepted by M.

Example: Palindromes around $\texttt{c}$

- Construct a PDA to recognize $\{w \cdot \mathtt{c} \cdot w^R \mid w \in \{\mathtt{a}, \mathtt{b}\}^*\}$

Example: Palindromes around `c`

- Construct a PDA to recognize $\{w \cdot \mathtt{c} \cdot w^R \mid w \in \{\mathtt{a}, \mathtt{b}\}^*\}$
- **Algorithm:** Push successive input symbols.
  When reading `c` switch to a new state,
  match subsequent input symbols with the top of the stack,
  popping the top.

Example: Palindromes around c

- Construct a PDA to recognize $\{w \cdot \mathtt{c} \cdot w^R \mid w \in \{\mathtt{a},\mathtt{b}\}^*\}$
- **Algorithm:** Push successive input symbols.
  When reading c switch to a new state,
  match subsequent input symbols with the top of the stack,
  popping the top.

$s \xrightarrow{\epsilon\,(\epsilon \to \$)} q$ (place a marker \$ on the stack)

$q \xrightarrow{\sigma\,(\epsilon \to \sigma)} q$ (push next letter)

$q \xrightarrow{\mathtt{c}\,(\epsilon \to \epsilon)} p$ (if c, switch to state p)

$p \xrightarrow{\sigma\,(\sigma \to \epsilon)} p$ (if letter matches stack-op pop it, else abort)

$p \xrightarrow{\epsilon\,(\$ \to \epsilon)} f$ (accept if top is \$)

And if the center is absent?

- $\{w \cdot w^R \mid w \in \{\mathtt{a}, \mathtt{b}\}^*\}$.
- Use nondeterminism!
- Replace $q \xrightarrow{c\,(\epsilon \to \epsilon)} p$ by $q \xrightarrow{\epsilon\,(\epsilon \to \epsilon)} p$.
- The resulting PDA:

$$
\begin{array}{lll}
s \xrightarrow{\epsilon\,(\epsilon \to \$)} q & \\
q \xrightarrow{\sigma\,(\epsilon \to \sigma)} q & (\sigma = \mathtt{a}, \mathtt{b}) \\
q \xrightarrow{\epsilon\,(\epsilon \to \epsilon)} p & \\
p \xrightarrow{\sigma\,(\sigma \to \epsilon)} p & (\sigma = \mathtt{a}, \mathtt{b}) \\
p \xrightarrow{\epsilon\,(\$ \to \epsilon)} f &
\end{array}
$$

Repeated use of nondeterminism

- Consider $\{\mathrm{a}^n\,\mathrm{b}^m \in \Sigma^* \mid m \leqslant n \leqslant 2m\}$
- What stack algorithm would work?

Repeated use of nondeterminism

- $\{\mathtt{a}^n \mathtt{b}^m \in \Sigma^* \mid m \leqslant n \leqslant 2m\}$
- What stack algorithm would work?
- Use four states s, q, p, f, s initial, s, f accepting.
- Transition rules:

$$s \xrightarrow{\epsilon\,(\epsilon \to \$)} q \qquad p \xrightarrow{b\,(a \to \epsilon)} p$$
$$q \xrightarrow{a\,(\epsilon \to a)} q \qquad p \xrightarrow{b\,(aa \to \epsilon)} p$$
$$q \xrightarrow{\epsilon\,(\epsilon \to \epsilon)} p \qquad p \xrightarrow{\epsilon\,(\$ \to \epsilon)} f$$

- M pushes the $\mathtt{a}$'s being read,
  switches nondeterministically to a "$\mathtt{b}$-reading state" p
  which empties the stack while reading $\mathtt{b}$'s,
  popping either a single $\mathtt{a}$ or two tta's at a time.

From CFGs to PDAs

- **THEOREM.** Every CFL is recognized by some PDA.
- For each CFG G we construct a PDA M, so that $\mathcal{L}(G) = \mathcal{L}(M)$.
- Example:
  G is $\mathtt{S} \rightarrow \mathtt{aSb} \mid \varepsilon$.
- Initial idea:
  generate on the stack a random string x,
  then compare x to the input w.
- A marker $\$$ used for stack bottom,
  and completion is then detectable.
- What's wrong here?

Alternating between generation and consumption

- What's wrong? We'd need to apply g's productions deep down the stack.
- But there is no need to wait:

  We can compare the (randomly) generate string
  as soon as feasible.

	Input	Stack	
	aabb	$S\$$	
			generate
	aabb	$aSb\$$	
compare			
	abb	$Sb\$$	
			generate
	abb	$aSbb\$$	
compare			
	bb	$Sbb\$$	
			generate
	bb	$bb\$$	
compare			
	b	$b\$$	
compare			
	ε	$\$$	

Every CFL is recognized by a PDA

- Let $G = (\Sigma, S, R)$ be a CFG.
  We define a PDA M that recognizes $\mathcal{L}(G)$.

Every CFL is recognized by a PDA

- Let $G = (\Sigma, S, R)$ be a CFG.
  We define a PDA M that recognizes $\mathcal{L}(G)$.
- ***States:*** Just three, say s , q and f .
  s initial, f accepting.
- ***Auxiliary symbols:*** Nonterminals of G and fresh $\$$.

Transition rules of the PDA

- ***Initializing*** the stack:

$$s \xrightarrow{\epsilon(\epsilon \to S\$)} q$$

Transition rules of the PDA

- ***Initializing*** the stack:

$$s \xrightarrow{\epsilon(\epsilon \to S\$)} q$$

  - For each production $A \to \alpha$ of G:

$$q \xrightarrow{\epsilon(A \to \alpha)} q$$

I.e., if stack-top is A, may apply this production of G.

Transition rules of the PDA

- ***Initializing*** the stack:

  $$s \xrightarrow{\epsilon(\epsilon \to S\$)} q$$

  - For each production $A \to \alpha$ of G:

  $$q \xrightarrow{\epsilon(A \to \alpha)} q$$

  I.e., if stack-top is A, may apply this production of G.

- For each $\sigma \in \Sigma$: $q \xrightarrow{\sigma\ (\sigma \to \epsilon)} q$.

  I.e., if stack-top is σ *matching current input symbol,* then σ is read off input, and popped off the stack.

- Acceptance: $q \xrightarrow{\epsilon(\$ \to \epsilon)} f$.

Example

- Grammar G: $S \to \mathtt{a}S\mathtt{b} \mid \varepsilon$

- The PDA obtained:

$$s \xrightarrow{\epsilon(\epsilon \to S\$)} q \qquad q \xrightarrow{a(a \to \epsilon)} q$$

$$q \xrightarrow{\epsilon(S \to aSb)} q \qquad q \xrightarrow{b(b \to \epsilon)} q$$

$$q \xrightarrow{\epsilon(S \to \epsilon)} q \qquad q \xrightarrow{\epsilon(\$ \to \epsilon)} f$$

- Here is a derivation of $\texttt{aabb}$ in G:

$$S \rightarrow \mathtt{a}S\mathtt{b} \rightarrow \mathtt{aa}S\mathtt{bb} \rightarrow \mathbf{aabb}$$

- And here is the corresponding trace of P:

$$\begin{array}{lll}
(s, \varepsilon) & \xrightarrow{\epsilon} & (q, S\$) \\
 & \xrightarrow{\epsilon} & (q, \mathtt{a}S\mathtt{b}\$) \\
 & \xrightarrow{\mathtt{a}} & (q, S\mathtt{b}\$) \\
 & \xrightarrow{\epsilon} & (q, \mathtt{a}S\mathtt{bb}\$) \\
 & \xrightarrow{\mathtt{a}} & (q, S\mathtt{bb}\$) \\
 & \xrightarrow{\epsilon} & (q, \mathtt{bb}\$) \\
 & \xrightarrow{\mathtt{b}} & (q, \mathtt{b}\$) \\
 & \xrightarrow{\mathtt{b}} & (q, \$) \\
 & \xrightarrow{\$} & (f, \varepsilon)
\end{array}$$

Converting PDAs to CFGs:
Reminder of NFA $\Rightarrow$ RegExp

- Given an NFA $N = (\Sigma, Q, s, A, \delta)$,
  for each $q, p \in Q$ and $I \subset Q$
  w considered the regular language Z_{qpT} of strings leading
  N from q to p using only intermediate states in T.
- The visual algorithm we defined is akin to calculating
  L_{qpI} for larger and larger I ,
  and for $q, p \notin I$.
- Consider now a PDA $P = (\Sigma, Q, s, A, \Gamma, \delta)$.
  We are again interested, for $q, p \in Q$,
  in the language of strings leading P from q to p .

- Except that in a PDA a configuration is not just a state, but a pair (state,stack), i.e. (q, α) where $q \in Q$ and $\alpha \in \Gamma^*$.
- Problem: No evident bound on stack size.
- Luckily, it suffices to consider the extended-states (q, ε) , i.e. the ones where the stack is empty!

Preparing the ground

- For pairs (q,p) of states let E_{qp} consist of the strings w leading P from (q,ε) to (p,ε):

$$E_{qp} = \{w \in \Sigma^* \mid (q,\varepsilon) \xrightarrow{w} (p,\varepsilon)\ \}$$

- Note that if $(q,\varepsilon) \xrightarrow{w} (p,\varepsilon)$ then $(q,\alpha) \xrightarrow{w} (p,\alpha)$ for any stack α, without even referring to the contents α of the stack.
- For this approach to succeed, we'd need to assume that
  - P uses just single-letter push and pop.
  - P accepts only when the stack is empty.
- A PDA P can be converted into an equivalent one satisfying (1)

by breaking compound $u_0 \to v_0$ into single-letter push and pop.

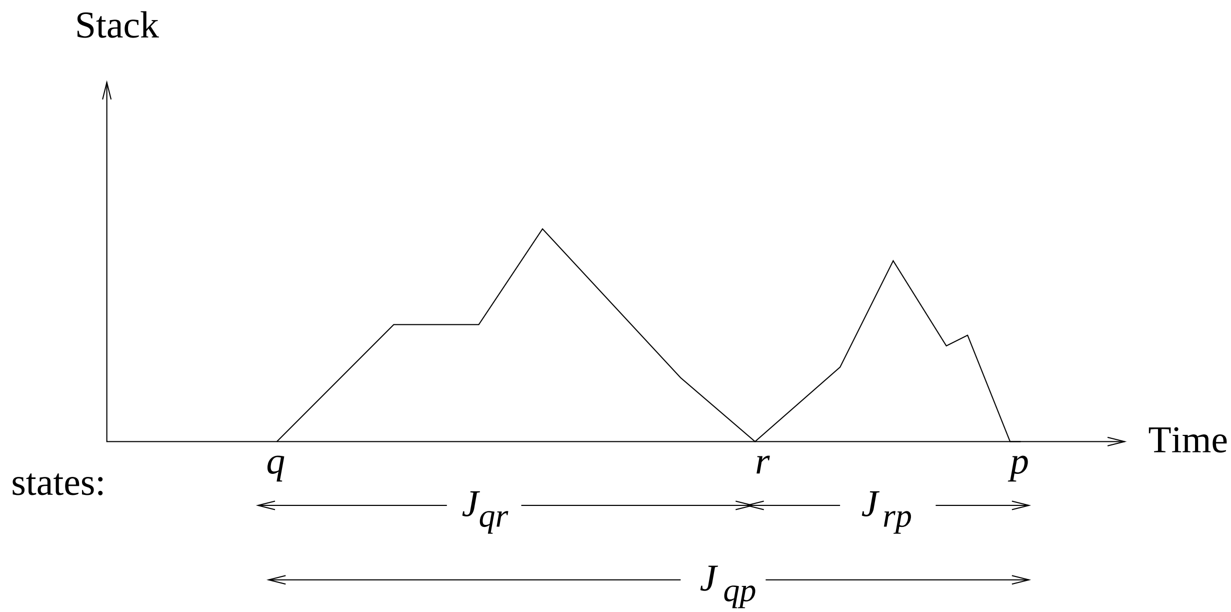
- Moreover, we guarantee empty stack on acceptance by augmenting P with transitions that empty the stack before actually reaching a (new) accepting state.

Generating the languages E_{qp}

- Given the PDA we define a CFG G over Σ :
- Non-terminals for the "journeys":
  J_{qp} (for each pair $q, p \in Q$),
  with the intent that J_{qp} generates the language E_{qp}.
- Initial non-terminal: J_{sa}
- We should have for each $q \in Q$ that $\varepsilon \in L_{qq}$.
  So G includes for each $q \in Q$ the production $A_{qq} \to \varepsilon$.

Splicing journeys

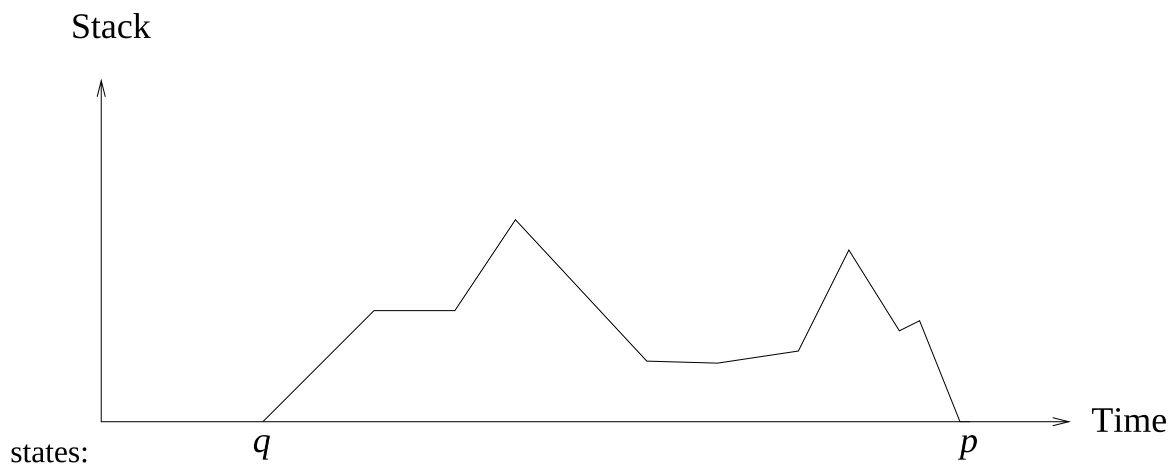
- If $(q, \varepsilon) \xrightarrow{u} (r, \varepsilon) \xrightarrow{v} (p, \varepsilon)$ then $(q, \varepsilon) \xrightarrow{u \cdot v} (p, \varepsilon)$.
- In other words, if we know that
  $J_{qr} \Rightarrow^* u$ and $J_{rp} \Rightarrow^* v$,
  then we should have $J_{qp} \Rightarrow^* u \cdot v$.
- So we include in G the production $J_{qp} \rightarrow J_{qr}\, J_{rp}$



- We include this production for each combination of q, r, p.

Productions for stack operations

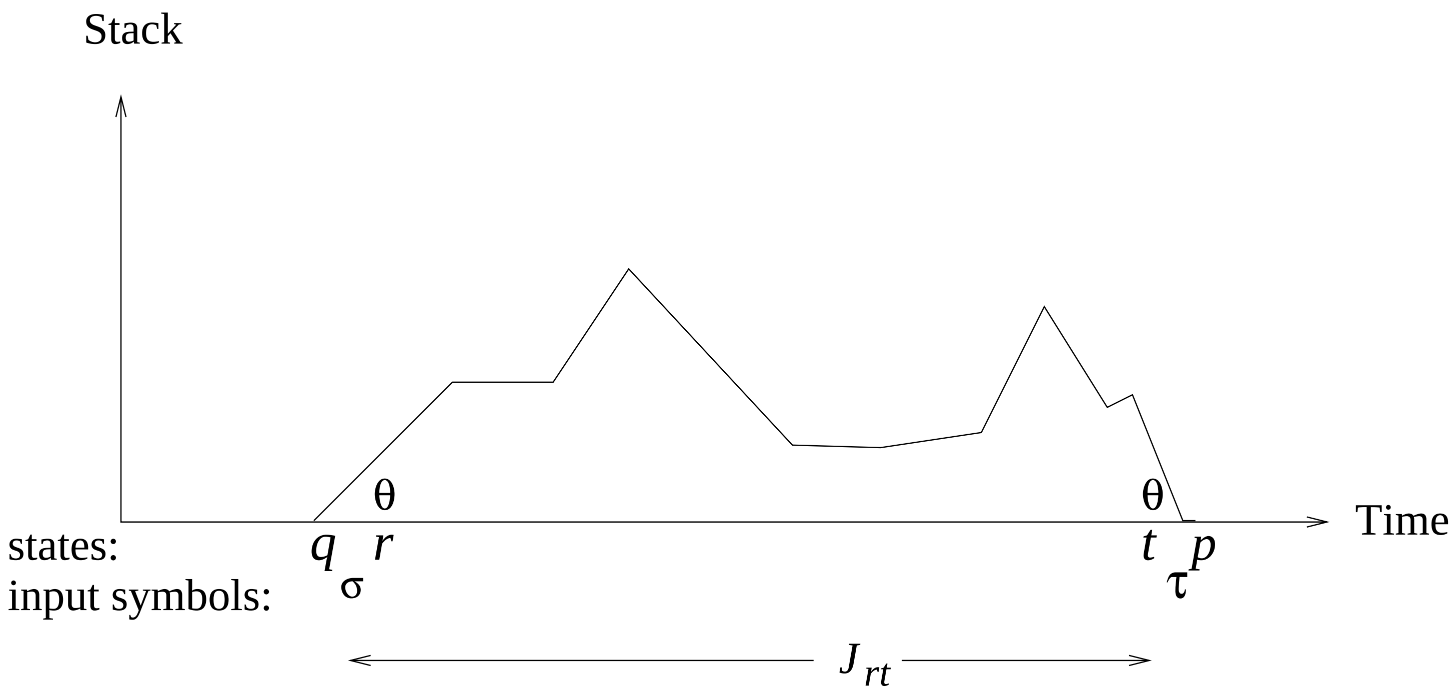
- Our productions so far are unrelated to the transitions of P.
- Suppose $(q, \varepsilon) \xrightarrow{w} (p, \varepsilon)$.
  If the trace has an empty stack along the way,
  i.e. $w = u \cdot v$ with $(q, \varepsilon) \xrightarrow{u} (r, \varepsilon) \xrightarrow{v} (p, \varepsilon)$ then we already have the production $J_{qp} \to J_{qr}\, J_{rp}$.
- If not, then we have



- The first move in this trace must read a symbol $\sigma \in \Sigma_\epsilon$, and push some symbol θ on the stack.
- Last move reads some $\tau \in \Sigma_\epsilon$ causing P to pop that θ (undisturbed throughout: the stack does not empty).

- That is, for some states r, t:

$$(q, \varepsilon) \xrightarrow{\sigma} (r, \theta) \quad \text{and} \quad (t, \theta) \xrightarrow{\tau} (p, \varepsilon)$$



- This is conveyed in G by the production $J_{qp} \rightarrow \sigma J_{rt} \tau$.

- In general, whenever P has transition-rules

$$q \xrightarrow{\sigma\,(\epsilon\to\theta} r \quad \text{and} \quad t \xrightarrow{\tau\,(\theta\to\epsilon)} p$$

with the same θ in both,
the grammar G includes the production $J_{qp} \to \sigma J_{rt} \tau$.

Proof concluded

- By induction on trace-length in M
  we obtain that, for all $q, p \in Q$,

$$J_{qp} \Rightarrow^*_G \sigma x \tau \quad \text{IFF} \quad (q, \varepsilon) \xrightarrow{\sigma x \tau} (p, \varepsilon)$$

- When q, p are the initial and accepting states s, f

$$J_{sf} \Rightarrow^* w \qquad (G \text{ generates } w)$$

IFF

$$(s, \varepsilon) \xrightarrow{w} (f, \varepsilon) \qquad (P \text{ accepts } w)$$

Example

- Let M over $\{\mathtt{a}, \mathtt{b}, \mathtt{c}\}$ have the following transition rules.

$$
\begin{array}{llll}
1. & s \xrightarrow{\epsilon\,(\epsilon \to \$)} q & 4. & p \xrightarrow{\epsilon\,(b \to \epsilon)} r \\
2. & q \xrightarrow{a\,(\epsilon \to a)} q & 5. & \xrightarrow{b\,(a \to \epsilon)} r \\
3. & q \xrightarrow{c\,(\epsilon \to b)} p & 6. & ? \xrightarrow{\epsilon\,(\$ \to \epsilon)} f
\end{array}
$$

- The construction above yields the following grammar (with initial nonterminal A_{sf})

$A_{tt} \to \varepsilon$	(all states t)
$A_{tu} \to A_{tv}\, A_{vu}$	(all states t, u, v)
$A_{qr} \to \mathtt{a}\, A_{qr}\, \mathtt{b}$	(pushing and popping $\mathtt{a}$, rules 2 and 5)
$A_{qr} \to \mathtt{c}\, A_{pp}\, \varepsilon$	(pushing and popping $\mathtt{b}$, rules 3 and 4)
$A_{sf} \to \varepsilon\, A_{qr}\, \varepsilon$	(pushing and popping $\$$, rules 1 and 6)

Little puzzles about PDAs

- Suppose M is a PDA that does not use its stack.
  What does M recognize?
- Suppose M is a PDA that uses its stack only up to depth 1000.
  What sort of language does M recognize?
- Suppose M is a super-PDA, that uses two stacks.
  What sort of language does M recognize?

Little puzzles about PDAs

- For a DFA M recognizing $L \subseteq \Sigma^*$,
  we obtained an automaton $\bar{M}$ recognizing $\bar{L} = \Sigma^* - L$
  by flipping accepting and non-accepting states.

  For PDAs we can't, since the complement of a CFL need not be CF.

  What's wrong with the same sort of flipping for PDAs?

Little puzzles about PDAs

- For DFAs M, N we constructed a product DFA
  that recognizes $\mathcal{L}(M) \cap \mathcal{L}(N)$.

  Why can't we use the same idea to build,
  for PDAs M, N a PDA that recognizes $\mathcal{L}(M) \cap \mathcal{L}(N)$?

The intersection of a CFL and a regular language

- But what if N does not use its stack?
- **Theorem**. *The intersection of a CFL and a regular language is CF.*

Examples of intersecting CF with Reg

1. $L = \{w \in \{\mathtt{a}, \mathtt{b}, \mathtt{c}\} \mid \#_a(w) = \#_b(w) = \#_c(w)\}$

   We have $\{\mathrm{a}^n\mathrm{b}^n\mathrm{c}^n \mid n \geqslant 0\} = L \cap \mathcal{L}(\mathrm{a}^* \cdot \mathrm{b}^* \cdot \mathrm{c}^*)$ So L cannot be CF.

Examples of intersecting CF with Reg

1. $L = \{w \in \{\mathtt{a}, \mathtt{b}, \mathtt{c}\} \mid \#_a(w) = \#_b(w) = \#_c(w)\ \}$

   We have $\{\mathrm{a}^n\mathrm{b}^n\mathrm{c}^n \mid n \geqslant 0\} = L \cap \mathcal{L}(\mathrm{a}^* \cdot \mathrm{b}^* \cdot \mathrm{c}^*)$ So L cannot be CF.

2. If L is a CFL, and F is finite,

   then $L - F$ is CF. Why? Where did we already use that?

Examples of intersecting CF with Reg

1. $L = \{w \in \{\mathbf{a}, \mathbf{b}, \mathbf{c}\} \mid \#_a(w) = \#_b(w) = \#_c(w)\ \}$

   We have $\{\mathrm{a}^n\mathrm{b}^n\mathrm{c}^n \mid n \geqslant 0\} = L \cap \mathcal{L}(\mathrm{a}^* \cdot \mathrm{b}^* \cdot \mathrm{c}^*)$ So L cannot be CF.

2. If L is a CFL, and F is finite,
   then $L - F$ is CF. Why? Where did we already use that?

3. Suppose $L \subseteq \Gamma^*$ is recognized by a PDA.
   If $\Sigma \subset \Gamma$, what about the set of Σ-strings in L?

The Chomsky Hierarchy

So far: two classes of languages

LANGUAGE CLASS:	**Regular**	**Context-free**
GRAMMARS:	regular grammars	CF grammars
MACHINES:	DFA=NFA	PDA
MEMORY:	internal no-write	stack
ACCESS:	on-line	on-line + stack

A non-CF grammar

- The following general grammar generates $\mathtt{a}^n\mathtt{b}^n\mathtt{c}^n$, which is not CF.

$$
\begin{array}{lllllllll}
S & \rightarrow & \varepsilon \mid SABC & & & & & & \\
C & \rightarrow & \mathtt{c} & \mathtt{c}A & \rightarrow & A\mathtt{c} & \mathtt{c}B & \rightarrow & B\mathtt{c} \\
B & \rightarrow & \mathtt{b} & \mathtt{b}A & \rightarrow & A\mathtt{b} & & & \\
A & \rightarrow & \mathtt{a} & & & & & &
\end{array}
$$

A non-CF grammar

- The following general grammar generates $\mathtt{a}^n\mathtt{b}^n\mathtt{c}^n$, which is not CF.

$$
\begin{array}{lll}
S \;\rightarrow\; \varepsilon \mid SABC & & \\
C \rightarrow \mathtt{c} & \mathtt{c}A \rightarrow A\mathtt{c} & \mathtt{c}B \rightarrow B\mathtt{c} \\
B \rightarrow \mathtt{b} & \mathtt{b}A \rightarrow A\mathtt{b} & \\
A \rightarrow \mathtt{a} & &
\end{array}
$$

- This grammar has production-sources of length > 1, so is not CF.

A non-CF grammar

- The following general grammar generates $\mathtt{a}^n\mathtt{b}^n\mathtt{c}^n$, which is not CF.

$$S \;\rightarrow\; \varepsilon \mid SABC$$
$$C \rightarrow \mathtt{c} \qquad \mathtt{c}A \rightarrow A\mathtt{c} \qquad \mathtt{c}B \rightarrow B\mathtt{c}$$
$$B \rightarrow \mathtt{b} \qquad \mathtt{b}A \rightarrow A\mathtt{b}$$
$$A \rightarrow \mathtt{a}$$

- Sample derivation:

$$S \Rightarrow SABC \Rightarrow SABCSABC \;\Rightarrow^2\; ABCABC$$
$$\Rightarrow^2\; AB\mathtt{c}AB\mathtt{c} \;\Rightarrow\; ABA\mathtt{c}B\mathtt{c} \;\Rightarrow\; ABAB\mathtt{cc}$$
$$\Rightarrow^2\; A\mathtt{b}A\mathtt{bcc} \;\Rightarrow^2\; AA\mathtt{bbcc}$$
$$\Rightarrow^2\; \mathtt{aabbcc}$$

Context-sensitive grammars

- A production (of a general grammar) is ***context-sensitive***
  if it is of the form $uAv \rightarrow uxv$
  where $x \neq \varepsilon$.
  u and v are any strings of terminals and/or nonterminals.
- Think of such a production as being $A \rightarrow x$
  subject to the "context" of u and v, i.e. where A is preceded by u and succeeded by v. We'll say that A is the ***core-source*** and x the ***core-target***
- A grammar is ***context-sensitive*** if all its productions are context-sensitive.

Context-sensitive languages

- A context-sensitive grammar cannot generate ε ,
  because its core-targets cannot be empty.
  A simple remedy is similar to that for Chomsky grammars:
  A **_context-sensitive language (CSL)_**
  is a language L generated by a CSG, possibly with ε added.
- **Theorem.**
  A language is context-sensitive iff it is recognized by an LBA.

Non-contracting languages

- Identifying CSGs for CSLs is often hard,
  so it is useful to refer to grammars that are
  less restrictive than CSGs yet still generate only CSLs.

Non-contracting languages

- Identifying CSGs for CSLs is often hard,
  so it is useful to refer to grammars that are
  less restrictive than CSGs yet still generate only CSLs.
- A ***non-contracting grammar***
  has no production whose target is shorter than its source.

Non-contracting languages

- Identifying CSGs for CSLs is often hard,
  so it is useful to refer to grammars that are
  less restrictive than CSGs yet still generate only CSLs.
- A ***non-contracting grammar***
  has no production whose target is shorter than its source.
- Our previous grammar generating $\{\, \mathrm{a}^n \mathrm{b}^n \mathrm{c}^n \mid n \geqslant 0 \,\}$
  fails to be non-contracting, since it has $S \to \varepsilon$.

Non-contracting languages

- Identifying CSGs for CSLs is often hard,
  so it is useful to refer to grammars that are
  less restrictive than CSGs yet still generate only CSLs.
- A ***non-contracting grammar***
  has no production whose target is shorter than its source.
- Our previous grammar generating $\{ \mathrm{a}^n\mathrm{b}^n\mathrm{c}^n \mid n \geqslant 0 \}$
  fails to be non-contracting, since it has $S \to \varepsilon$.
- A ***non-contracting language*** is a language generated
  by a non-contrasting grammar, possibly with ε added.

Dealing with ε

- $L = \{\ \mathrm{a}^n\mathrm{b}^n\mathrm{c}^n \ \mid\ n \geqslant 0\ \}$
  cannot be generated by a non-contracting grammar
  but $\{\mathrm{a}^n\mathrm{b}^n\mathrm{c}^n \ \mid\ n > 0\}$ can:

Dealing with ε

- $L = \{ \mathrm{a}^n\mathrm{b}^n\mathrm{c}^n \mid n \geqslant 0 \}$
  cannot be generated by a non-contracting grammar
  but $\{\mathrm{a}^n\mathrm{b}^n\mathrm{c}^n \mid n > 0\}$ can:
- Replace in the grammar for the former
  the production $S \rightarrow \varepsilon$ by $S \rightarrow ABC$!

Dealing with ε

- $L = \{\ \mathrm{a}^n\mathrm{b}^n\mathrm{c}^n \ \mid\ n \geqslant 0\ \}$
  cannot be generated by a non-contracting grammar
  but $\{\mathrm{a}^n\mathrm{b}^n\mathrm{c}^n \ \mid\ n > 0\}$ can:
- Replace in the grammar for the former
  the production $S \to \varepsilon$ by $S \ \to\ ABC$!
- In general, if $L = \mathcal{L}(G) \cup \{\varepsilon\}$ where G is non-contracting
  then $L = \mathcal{L}(G')$ where G' is
  G with a fresh initial nonterminal S_0
  and new productions $S_0 \to S \mid \varepsilon$ (S is the initial of G).

Context-sensitive equivalent to non-contracting

- The productions of a CSG have the form

  $uAv \to uxv$ with $x \neq \varepsilon$.

  So they are non-contracting.

Context-sensitive equivalent to non-contracting

- The productions of a CSG have the form
  $uAv \to uxv$ with $x \neq \varepsilon$.
  So they are non-contracting.
- Conversely, every non-contracting production can be obtained using context-sensitive productions.

- Example: $ABC \rightarrow DEF$ is equivalent to the following set of context-sensitive productions:

$$B \rightarrow \dot{B} \qquad C \rightarrow \dot{C}$$

$$A\dot{B}\dot{C} \rightarrow \dot{D}\dot{B}\dot{C} \qquad \dot{D}\dot{B}\dot{C} \rightarrow \dot{D}\dot{E}\dot{C} \qquad D\dot{E}\dot{C} \rightarrow D\dot{E}F$$

$$\dot{D} \rightarrow D \qquad \dot{E} \rightarrow E$$

- Example: $ABC \to DEF$ is equivalent to
  the following set of context-sensitive productions:

$$B \to \dot{B} \qquad C \to \dot{C}$$

$$A\dot{B}\dot{C} \to \dot{D}\dot{B}\dot{C} \qquad \dot{D}\dot{B}\dot{C} \to \dot{D}\dot{E}\dot{C} \qquad D\dot{E}\dot{C} \to D\dot{E}F$$

$$\dot{D} \to D \qquad \dot{E} \to E$$

- $\dot{B}, \dot{C}, \dot{D}$ and $\dot{E}$ prevent these productions
  from interacting with possible other productions for B, C, D, E.

Language Class:	**Regular**	**Context-free**	**Context-sensitive**
Grammars:	regular	Context-free	Context-sensitive
Machines:	DFA=NFA	NFA + stack	LBA
Memory:	internal	stack	on-site
Access:	on-line	on-line + stack	two-way
New:	a^*	$a^n b^n$	$a^n b^n c^n$