SYMBOLIC COMPUTING

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 ***rewrite rules,*** also called productions.

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 ***rewrite rules,*** also called productions.

- A $\boxed{\textit{rewrite-rule}}$ is of the form $z \rightarrow y$ where z, y are strings.

- z is the $\boxed{\textit{source}}$ of the production, and y its $\boxed{\textit{target.}}$

- A finite set of rewrite rules is a $\boxed{\textit{rewrite system.}}$

A familiar example of rewriting

$$\mathtt{0} \wedge \mathtt{0} \;\longrightarrow\; \mathtt{0}$$

$$\mathtt{0} \wedge \mathtt{1} \;\longrightarrow\; \mathtt{0}$$

$$\mathtt{1} \wedge \mathtt{0} \;\longrightarrow\; \mathtt{0}$$

$$\mathtt{1} \wedge \mathtt{1} \;\longrightarrow\; \mathtt{1}$$

A familiar example of rewriting

$0 \wedge 0 \to 0$	$0 \vee 0 \to 0$
$0 \wedge 1 \to 0$	$0 \vee 1 \to 1$
$1 \wedge 0 \to 0$	$1 \vee 0 \to 1$
$1 \wedge 1 \to 1$	$1 \vee 1 \to 1$

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

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

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 $u \rightarrow u'$ 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 $\boxed{\textit{reduces to}}$ 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 $\boxed{\textit{derivation}}$ 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.

Example

A derivation in our boolean rewrite-system:

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

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 ***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: $\mathbf{S}$.

Grammars

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

  These are of the form $w \to t$

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

Examples

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

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

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

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 $S \to \mathtt{a}$ and $S \to \mathtt{b}$.
  $\mathcal{L}(G) = \{\mathtt{a}, \mathtt{b}\}$.

Examples

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

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

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

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 $S \rightarrow \mathrm{a}S$, $S \rightarrow S\mathrm{b}$ and $S \rightarrow \varepsilon$.
- $\mathcal{L}(G) = \mathcal{L}(\mathrm{a}^*\mathrm{b}^*)$
- $S \rightarrow \mathrm{a}S\mathrm{b}$ and $S \rightarrow \varepsilon$.
- 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 $\boxed{\textbf{\textit{context-free grammar (CFG)}}}$ 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 $\boxed{\textbf{\textit{context-free language (CFL).}}}$
- Context-free grammars are also called $\boxed{\textbf{\textit{inductive grammars.}}}$
- 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

$$\begin{array}{lcl} P & \to & L\,P\,L \quad | \quad L \,|\, \varepsilon \\ L & \to & \mathtt{a} \mid \mathtt{b} \mid \cdots \mid \mathtt{z} \end{array}$$

  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:
  $P \quad \to \quad \sigma\, P\sigma \mid \sigma \mid \varepsilon \quad (\sigma \in \Sigma)$

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

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

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

The Context-Freedom Theorem

- Intuitively clear: context-free productions guarantee a separation between descendents of one occurrence of a variable and descendents of another.

- This is captured more formally by the

  Context-Freedom 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

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

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

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

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

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

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

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

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

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

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!

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 CFLs

Using the strictly-regular definition

- We show that every regular language us CF.

Using the strictly-regular definition

- We show that every regular language us CF.
- We use the generative definition of the strincly regular languages. Their definition as strictly-regular languages is simplifying this.

Using the strictly-regular definition

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

1. The rivial languages $\emptyset, \ \{\boldsymbol{\varepsilon}\}, \ \{\boldsymbol{\sigma}\} \ \ (\sigma \in grS)$ are strictly-regular.
2. The union, concatenation, and star of strictly-regular languaes are strictly regular.

Using the strictly-regular definition

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

1. The rivial languages $\emptyset, \ \{\boldsymbol{\varepsilon}\}, \ \{\boldsymbol{\sigma}\} \ (\sigma \in grS)$ are strictly-regular.
2. The union, concatenation, and star of strictly-regular languaes are strictly regular.

- We show that all such languages are CF by induction on this generative definition.

The trivial languages are CF

- $\emptyset$:

The trivial languages are CF

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

The trivial 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 variable
  and R is $R_0 \cup R_1$ augmented with the production $S \to S_0 \mid S_1$.

Closure under union

- $L_0 \cup L_1$ is generated by $(\Sigma,\ V \cup V' + S,\ S,\ R)$
  where S is a fresh variable
  and R is $R_0 \cup R_1$ augmented with the production $S \to S_0 \mid S_1$.
- 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 variable
  and R is $R_0 \cup R_1$ augmented with the production $S \rightarrow S_0 \mid S_1$.
- G generates each $w \in L_0 \cup L_1$.
- Conversely, a derivation D in G for $S \Rightarrow_G w$
  must start with $S \rightarrow S_0$ or $S \rightarrow S_1$ 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

- $L_0 \cdot L_1$ is generated by $(\Sigma,\ V \cup V' + S,\ S,\ R)$
  where S is a fresh variable
  and R is $R_0 \cup R_1$ augmented with the production $S \to S_0\, S_1$.

Closure under concatenation

- $L_0 \cdot L_1$ is generated by $(\Sigma,\ V \cup V' + S,\ S,\ R)$
  where S is a fresh variable
  and R is $R_0 \cup R_1$ augmented with the production $S \rightarrow S_0 S_1$.
- G generates each $w \in L_0 \cdot L_1$.

Closure under concatenation

- $L_0 \cdot L_1$ is generated by $(\Sigma,\ V \cup V' + S,\ S,\ R)$
  where S is a fresh variable
  and R is $R_0 \cup R_1$ augmented with the production $S \rightarrow S_0\, S_1$.
- G generates each $w \in L_0 \cdot L_1$.
- Conversely, a derivation D in G for $S \Rightarrow_G w$
  must start with $S \rightarrow S_0 \cdot S_1$, and by the Context-freedom Theorem we have $w = w_0 \cdot w_1$ with D a merge of a derivation of w_0 from S_0 and a derivation of w_1 from S_1 .

Closure under star

- L_0^* is generated by $(\Sigma, V_0 + S,\ S,\ R)$
  where S is a fresh variable
  and R is R_0 augmented with the production $S \to S_0 S \mid \varepsilon$.

Closure under star

- L_0^* is generated by $(\Sigma, V_0 + S,\ S,\ R)$
  where S is a fresh variable
  and R is R_0 augmented with the production $S \to S_0 S \mid \varepsilon$.
- G generates each $w \in L_0^*$.
  By induction on k each $w = w_1 \cdot w_k$ $(w_i \in L_0)$ is derived:

For $k = 0$ the string $w = \varepsilon$ is derived outright.
And $S \Rightarrow w_1 \cdot \cdots \cdot w_k$ for each $w_1, \ldots w_k \in L_0$
then $S \Rightarrow w_1 \cdot \cdots \cdot w_k \cdot w_{k+1}$ is derived by reducing S to $S_0 \Rightarrow S$ and combining a deriation in G for $S \Rightarrow w_1 \cdot \cdots \cdot w_k$ with a derivation in G_0 of w_{k+1}.

Closure under star

- L_0^* is generated by $(\Sigma, V_0 + S,\ S,\ R)$
  where S is a fresh variable
  and R is R_0 augmented with the production $S \to S_0 S \mid \varepsilon$.
- G generates each $w \in L_0^*$.
  By induction on k each $w = w_1 \cdot w_k$ $(w_i \in L_0)$ is derived:
  For $k = 0$ the string $w = \varepsilon$ is derived outright.
  And $S \Rightarrow w_1 \cdot \cdots \cdot w_k$ for each $w_1, \ldots w_k \in L_0$
  then $S \Rightarrow w_1 \cdot \cdots \cdot w_k \cdot w_{k+1}$ is derived by reducing S to $S_0 \Rightarrow S$ and combining a deriation in G for $S \Rightarrow w_1 \cdot \cdots \cdot w_k$ with a derivation in G_0 of w_{k+1}.
- For the converse use induction on derivation length,
  If D is a derivation in G for $S \Rightarrow w$ then it must start with $S \to S_0 S$,
  By the Context-Freedom Theorem $w = u \cdot v$ where $S_0 \Rightarrow u$ and $S \to v$.
  We have $u \in L_0$ and by IH $v \in L_0^*$. SO $w \in L_0^*$.

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: $\{\mathrm{a}^n\mathrm{b}^n \mid n \geqslant 0\}$ is CF.

Parsing

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 prcess.

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

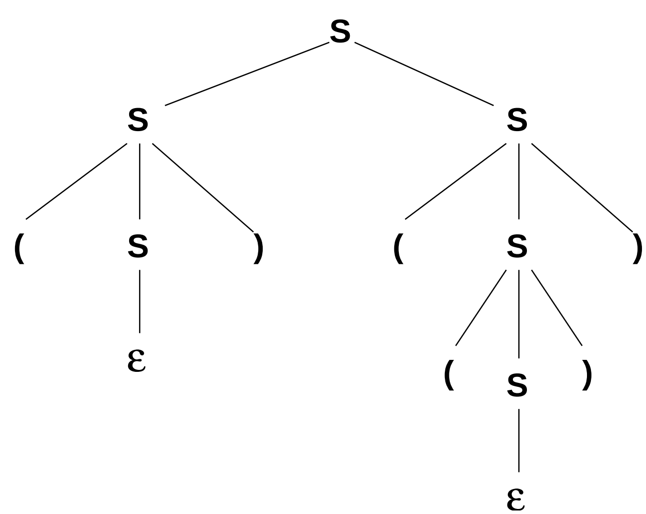
- Recall CFG for balanced parentheses: $S \to \varepsilon \mid SS \mid (S)$
- A derivation for the string $()(())$:

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

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

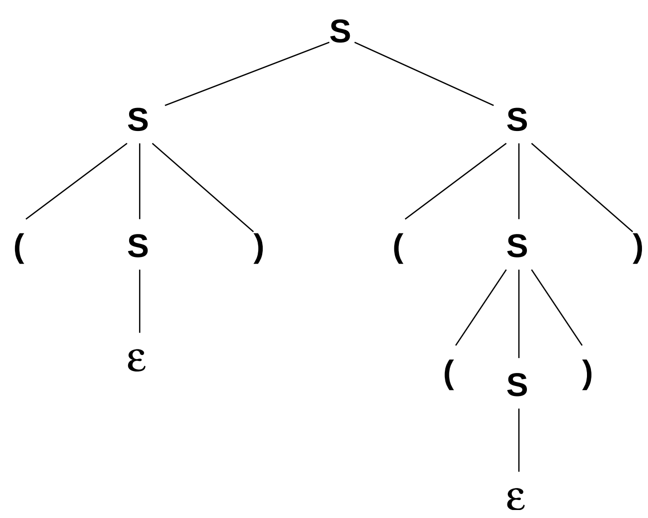
$$S \Rightarrow SS \Rightarrow S(S) \Rightarrow (S)(S) \Rightarrow ()(S) \Rightarrow ()((S)) \Rightarrow ()(())$$

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



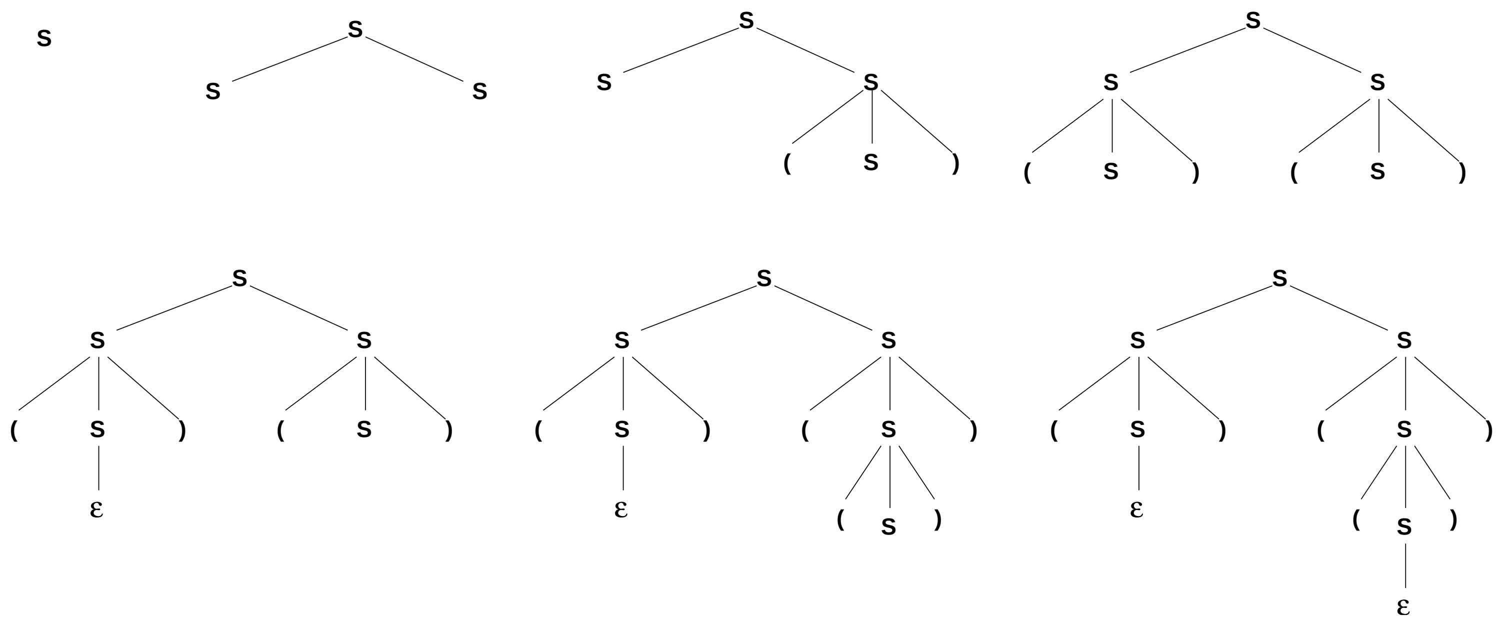
- Recall CFG for balanced parentheses: $S \to \varepsilon \mid SS \mid (S)$
- A derivation for the string $()(())$:

  $S \Rightarrow SS \Rightarrow S(S) \Rightarrow (S)(S) \Rightarrow ()(S) \Rightarrow ()((S)) \Rightarrow ()(())$
- Represented as a tree with terminals for leaves
  and variables for internal nodes:

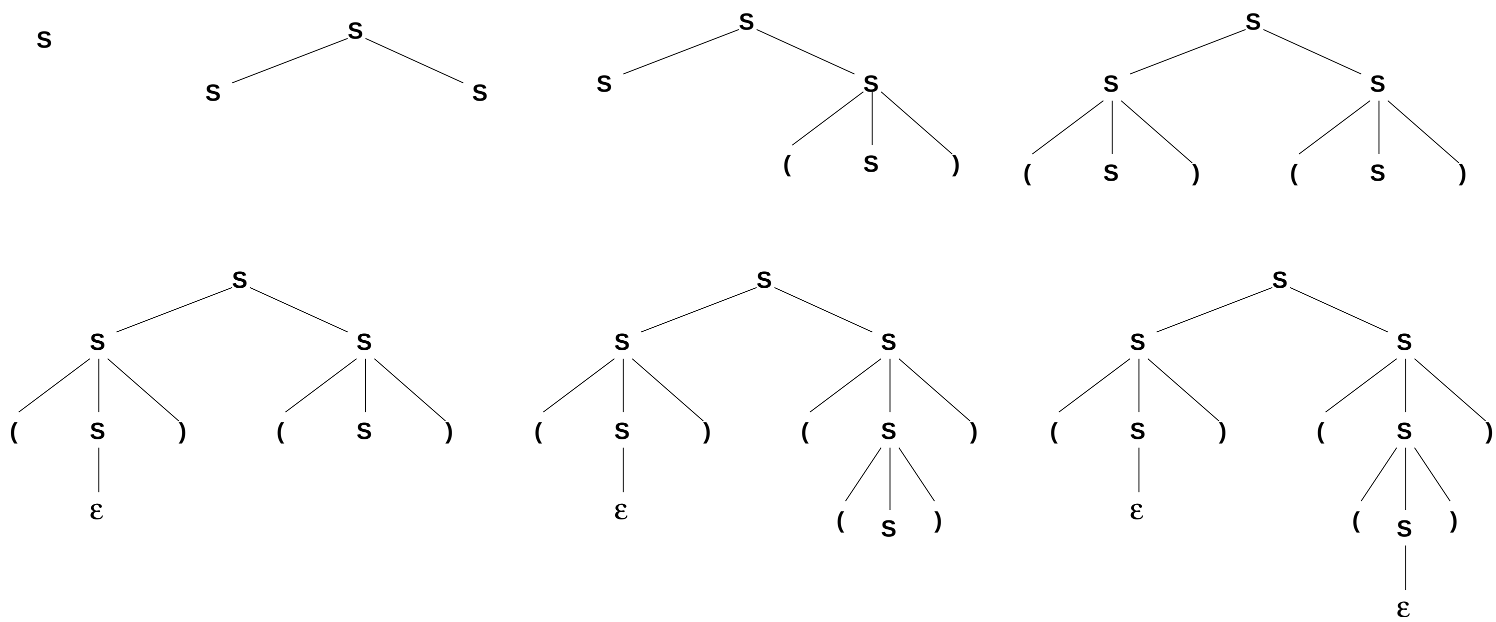


- This is a ***derivation-tree,*** or ***pars-tree***
  (of the grammar G for the string w).

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



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



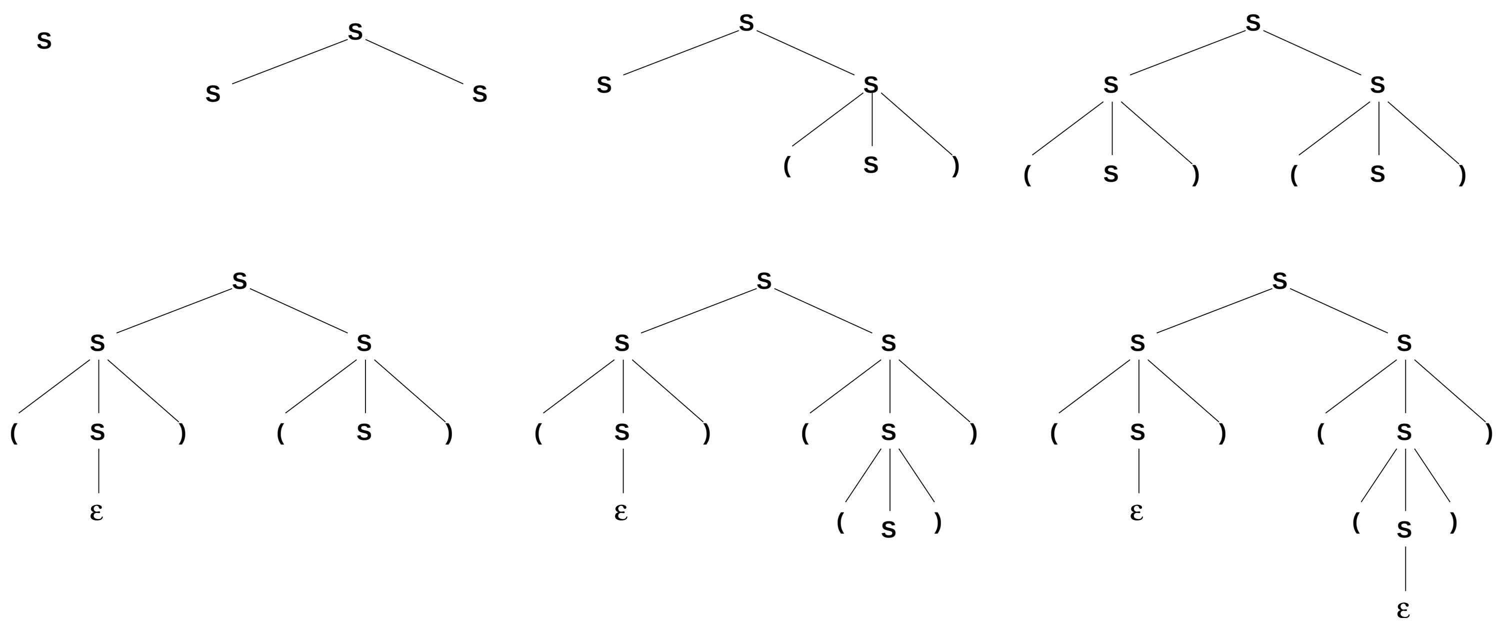
- The parse-tree is more important than the derivation.

  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 is more important than the derivation.

  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 ***leftmost-derivation*** for the tree,

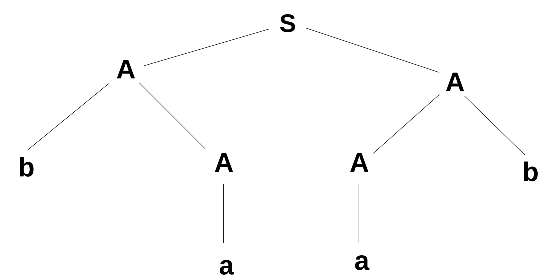
  obtained by repeatedly expanding the leftmost variable.

Another example

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

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

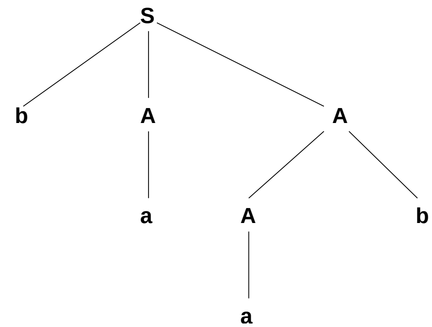
- The corresponding derivation tree:



- 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:

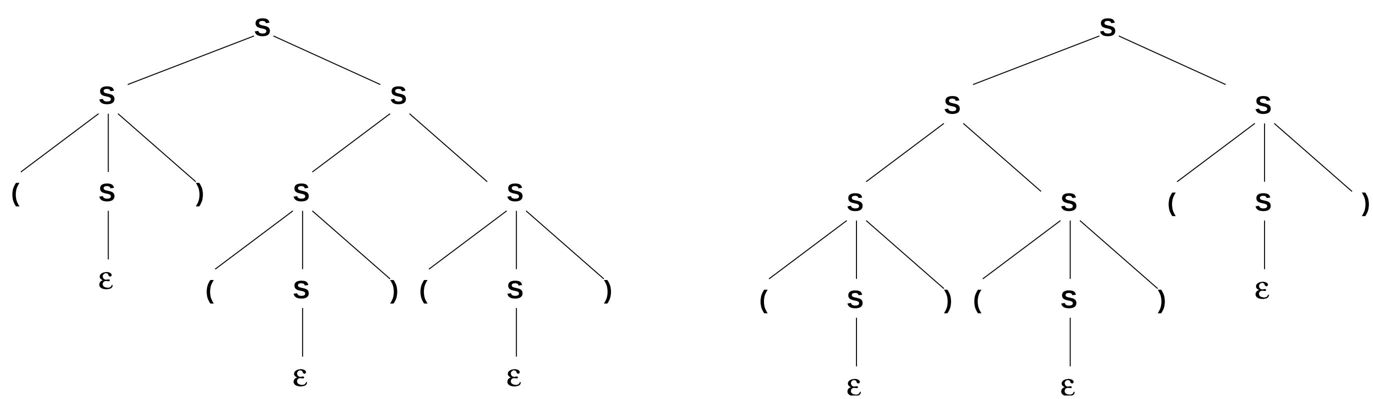


The leftmost derivation for this parse-tree:

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

Ambiguous grammars

- A derivation-tree usually represents several derivations.
  Can a grammar have different derivation-*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

Familiar example: Arith w/o parentheses

- Alphabet $\{\mathtt{a}, \mathtt{b}, \mathtt{+}, \times\}$,
  Grammar G with production rules:

$$S \to S\mathtt{+}S \mid S\times S \mid \mathtt{a} \mid \mathtt{b}$$

- Two different derivations of G for the string $\mathtt{a+b}\times\mathtt{a+b}$.

$$\begin{aligned} S &\Rightarrow S+S \\ &\Rightarrow \mathtt{a}+S \\ &\Rightarrow \mathtt{a}+S\times S \\ &\Rightarrow \mathtt{a}+\mathtt{b}\times S \\ &\Rightarrow \mathtt{a}+\mathtt{b}\times S+S \\ &\Rightarrow \mathtt{a}+\mathtt{b}\times \mathtt{a}+S \\ &\Rightarrow \mathtt{a}+\mathtt{b}\times \mathtt{a}+\mathtt{b} \end{aligned} \qquad \begin{aligned} S &\Rightarrow S\times S \\ &\Rightarrow S+S\times S \\ &\Rightarrow \mathtt{a}+S\times S \\ &\Rightarrow \mathtt{a}+\mathtt{b}\times S \\ &\Rightarrow \mathtt{a}+\mathtt{b}\times S+S \\ &\Rightarrow \mathtt{a}+\mathtt{b}\times \mathtt{a}+S \\ &\Rightarrow \mathtt{a}+\mathtt{b}\times \mathtt{a}+\mathtt{b} \end{aligned}$$

Dual-clipping in CFLs

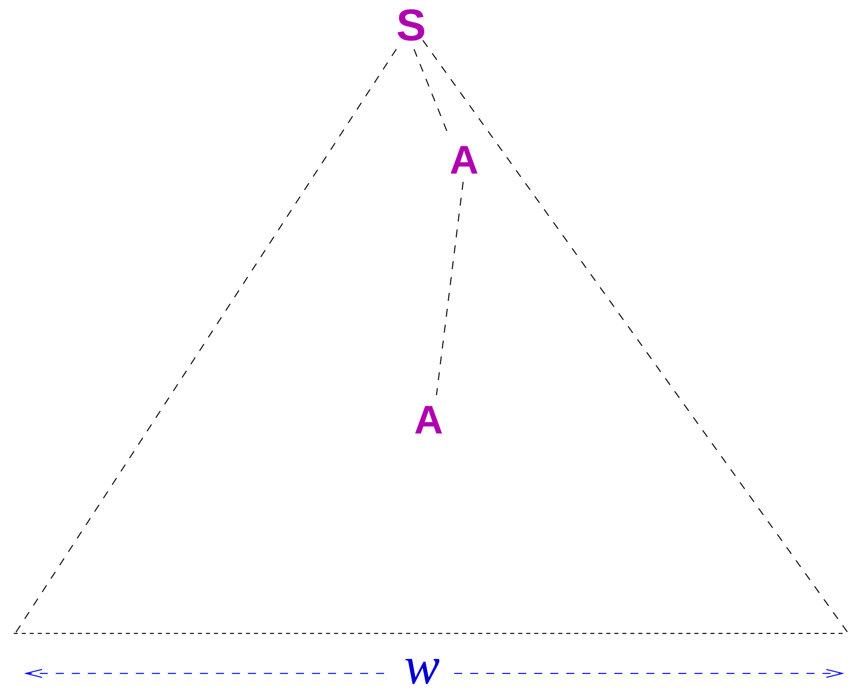
- The Clipping Theorem is based on the observation that if M is a k-state DFA then any trace of M of length $\geqslant k$ has some state q repeating.
- And a substring y leading from one occurence of q to another may be short-circuited, yielding the acceptance of a clipped string.

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.
- And a substring y leading from one occurence of q to another may be short-circuited, yielding the acceptance of a clipped string.
- This does not work as stated for for CFLs. But why?

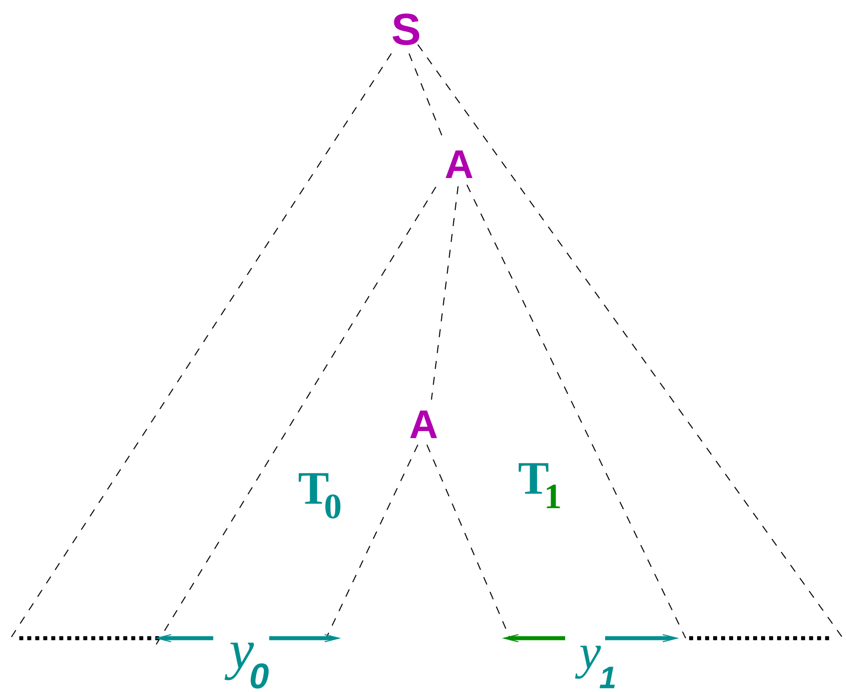
Dual-clipping in CFLs

- 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.
- And a substring y leading from one occurence of q to another may be short-circuited, yielding the acceptance of a clipped string.
- This does not work as stated for for CFLs. But why?
- Whereas a DFA accepts a string w by a "horizobntal" scan, 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

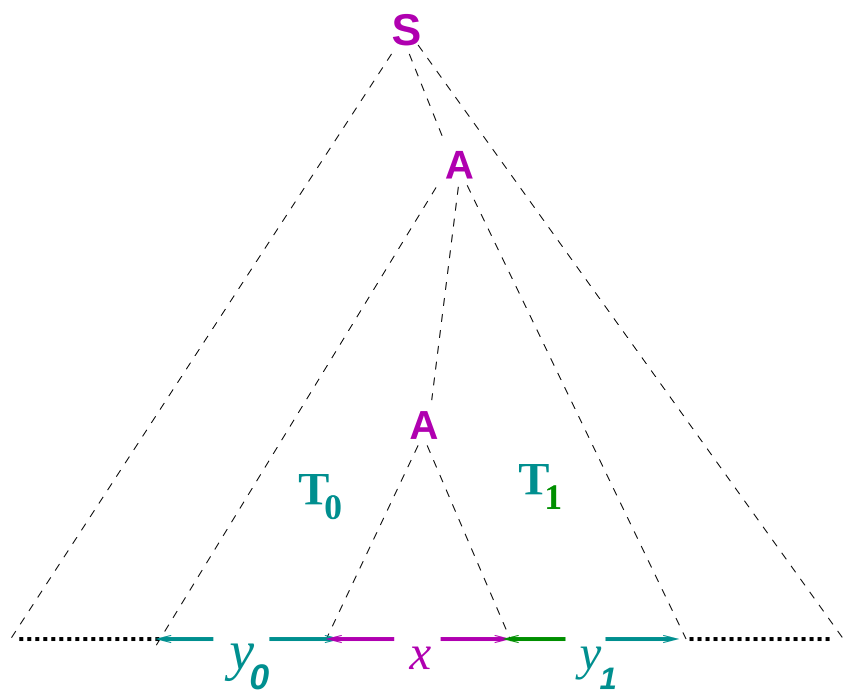
Dual-Clipping for CFLs

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



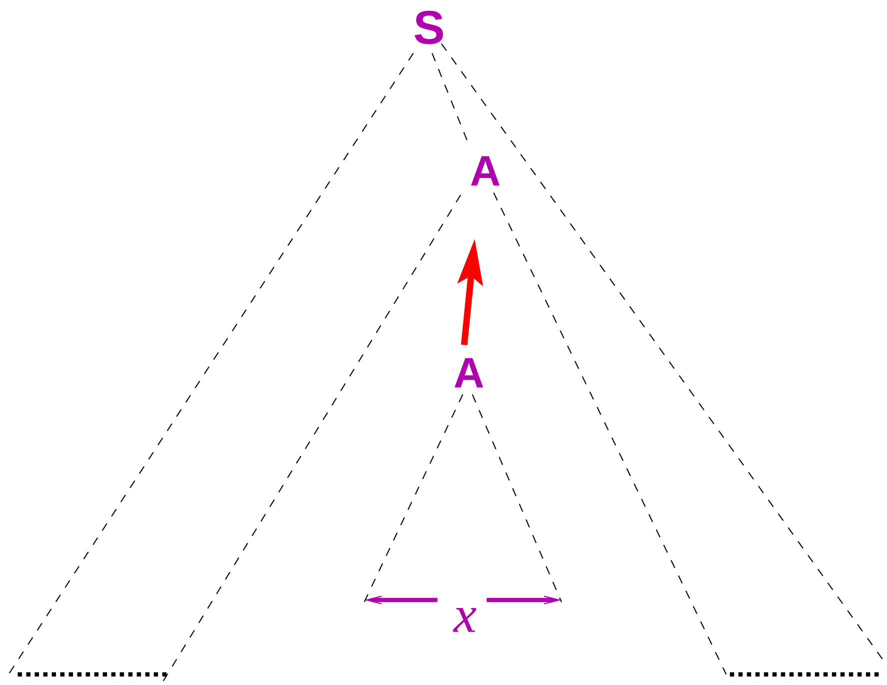
Dual-Clipping for CFLs

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



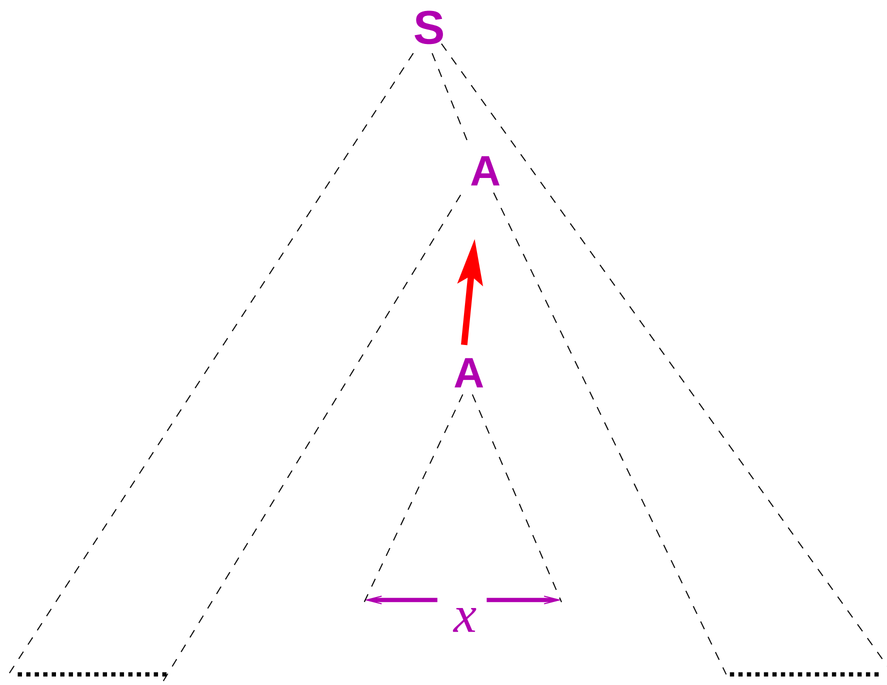
Dual-Clipping for CFLs

- 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 $w' \in L$.

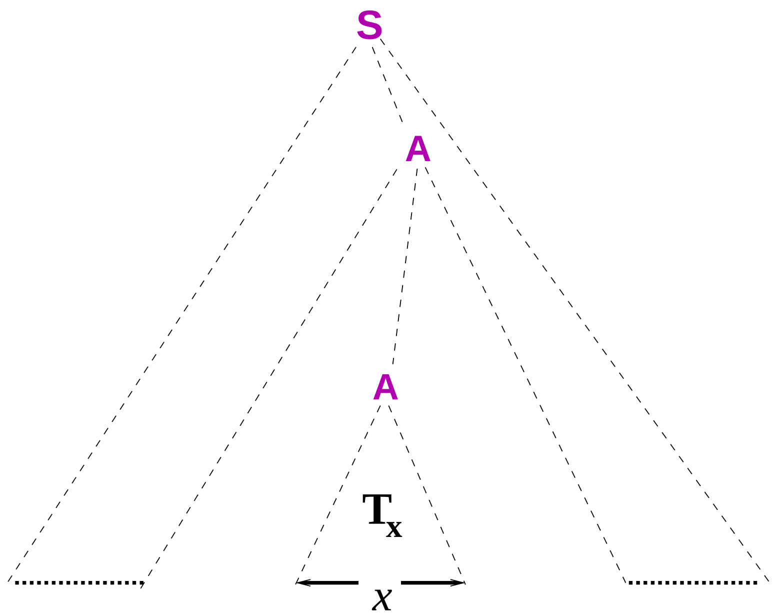
- Core idea: variable repeating on a branch.

- We'll also need to

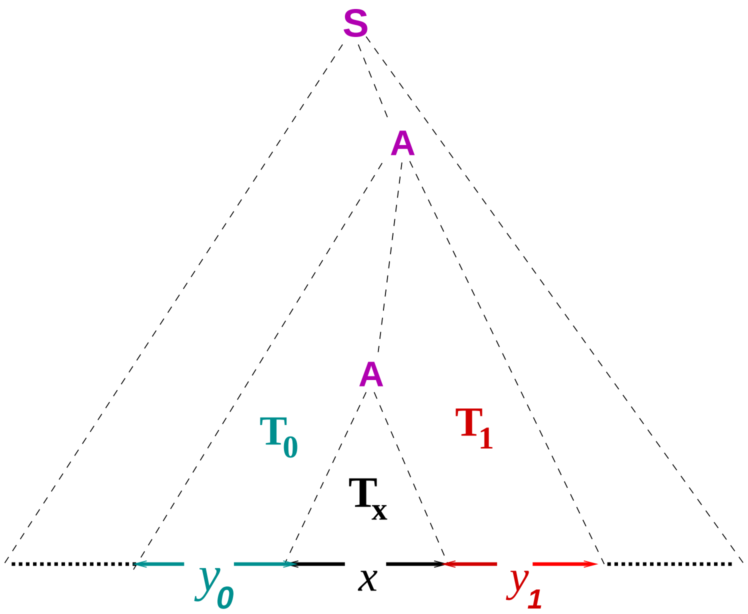
  1. Give conditions that guarantee such a repetition
  2. Ensure that the clipping obtained is non-empty
  3. Obtain two clipped substrings that are "not too far apart".

A repeated variable on a branch

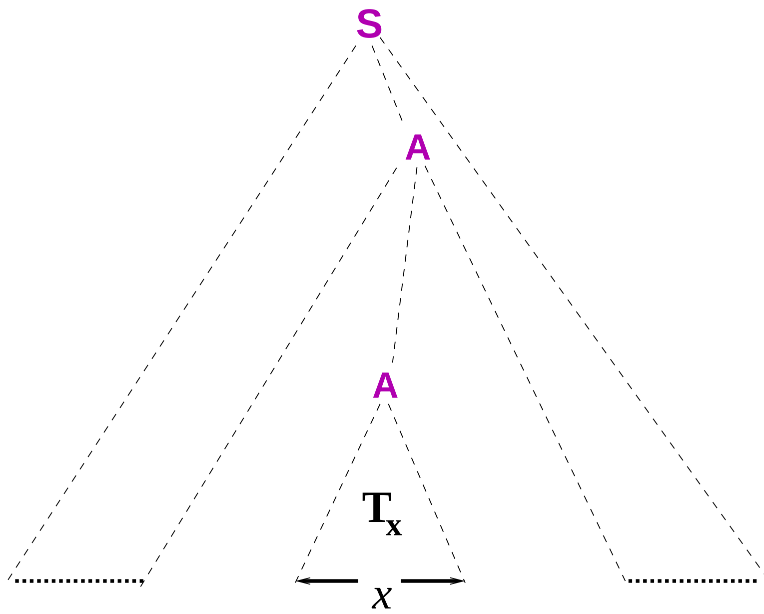
- Suppose T is a parse-tree of a CFG G for w with variable A repeating on a branch.



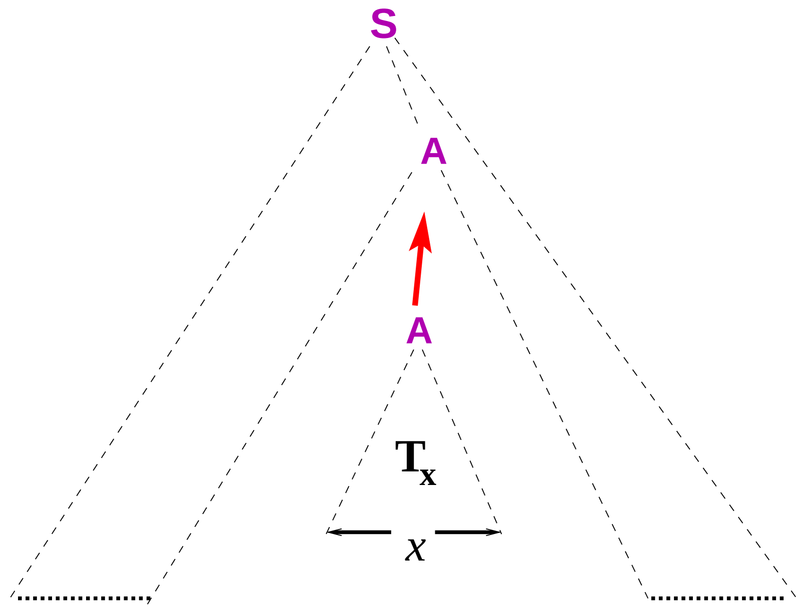
- The lower occurrence of A generates a substring x.



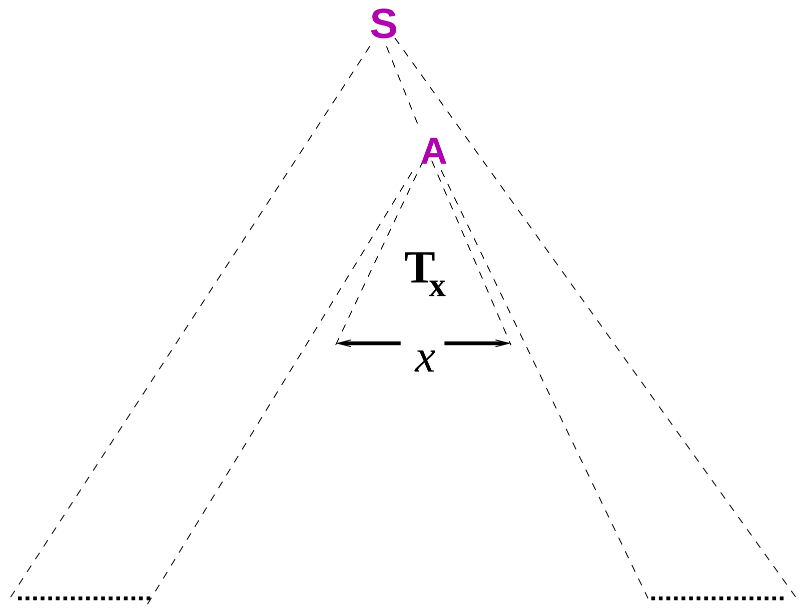
- The upper occurrence of A generates a substring $y_0\, x\, y_1$.



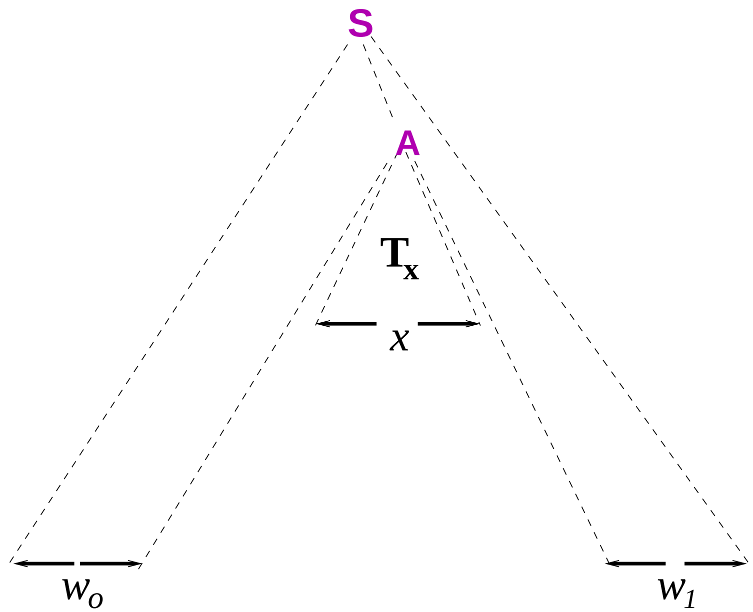
- Eliminating y_0 and y_1 yields a parse-tree except for the branch-segment between the two occurrences of A .



- 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

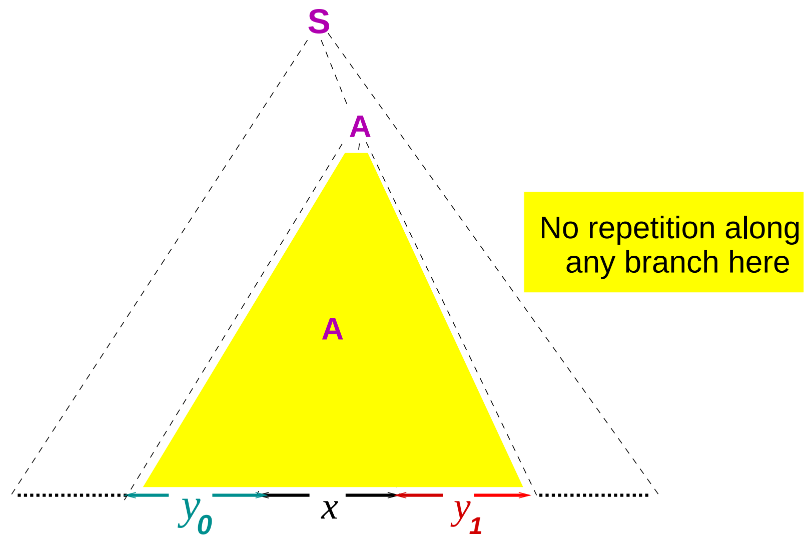
- Say that a production $X \to \sigma_1 \cdots \sigma_\ell$ has ***length*** ℓ and that the $\boxed{\textit{degree}}$ of a grammar is the maximal length of its productions.
- A binary tree of height h has $\leqslant 2^h$ leaves.
  Generally, a tree of degree d has $\leqslant d^h$ leaves.
- For a grammar 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 lowermost pair of a variable repeating:
  there can be then no repetition on a branch under the upper occurrence.



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

The Dual-clipping Theorem

Dual-clipping Theorem for CFLs (Formal statement)

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

The Dual-clipping Theorem

Dual-clipping Theorem for CFLs (Formal statement)

- **Theorem.** Let G be a CFG over Σ with m variables and of degree d (all productions are of length $\leqslant d$.
  - If $w \in \mathcal{L}(G)$ has length $\geqslant k = d^m$
  - then w has a substring p of length $\leqslant k$, with disjoint substrings y_0, y_1 not both empty, such that the string w' obtained from w by removing y_0 and y_1 is also in L.
- Stated formally: w can be factored as $w = w_0 \cdot y_0 \cdot x \cdot y_1 \cdot w_1$, where y_0, y_1 are not both empty and $|y_0 \cdot x \cdot y_1| \leqslant k$, so that $w_0 \cdot x \cdot w_1 \in L$.

The Dual-clipping Theorem

Dual-clipping Theorem for CFLs (Formal statement)

- **Theorem.** Let G be a CFG over Σ with m variables and of degree d (all productions are of length $\leqslant d$.
  - If $w \in \mathcal{L}(G)$ has length $\geqslant k = d^m$
  - then w has a substring p of length $\leqslant k$, with disjoint substrings y_0, y_1 not both empty, such that the string w' obtained from w by removing y_0 and y_1 is also in L.
- Stated formally: w can be factored as $w = w_0 \cdot y_0 \cdot x \cdot y_1 \cdot w_1$, where y_0, y_1 are not both empty and $|y_0 \cdot x \cdot y_1| \leqslant k$, so that $w_0 \cdot x \cdot w_1 \in L$.
- We refer to $k = d^m$ as G's ***clipping constant,*** and to p as the ***critical substring.***

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 ***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 then 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 *we can find*
    a $w \in L$ *of length* $\geqslant k$ *so that*
    for every substring $p = y_0 \cdot x \cdot h_1$ *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 $L = \{\mathtt{a}^n\mathtt{b}^n\mathtt{c}^n \mid n \geqslant 0\}$.
  We show that L is not CF.

Example: an-bn-cn

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

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.
- Suppose $L = \mathcal{L}(G)$, where G is a CFG with clipping constant k.
- Take $w = \mathrm{a}^k\mathrm{b}^k\mathrm{c}^k \in L$.
  By the Dual-Clipping Theorem we can clip off some y_0, y_1
  within a k-long substring p of w yielding a string $w' \in L$.
- But this is impossible:
  since $|p| \leqslant k$ it has at most two of the three letters,
  and w' must have fewer occurrences of a removed letter than of a non-removed one.
- Conclusion: 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.
- Suppose $L = \mathcal{L}(G)$, where G is a CFG with clipping constant k.
- Take $w = \mathrm{a}^k\mathrm{b}^k\mathrm{c}^k \in L$.
  By the Dual-Clipping Theorem we can clip off some y_0, y_1
  within a k-long substring p of w yielding a string $w' \in L$.
- But this is impossible:
  since $|p| \leqslant k$ it has at most two of the three letters,
  and w' must have fewer occurrences of a removed letter than of a non-removed one.
- 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. The substring p and its factorization $p = y_0 \cdot x \cdot y_1$ are all 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. The substring p and its factorization $p = y_0 \cdot x \cdot y_1$ are all unknown, i.e. ***given to us.***
4. We must show that ***whatever they are,*** subject to the constraints, the clipped string w' is out of L.

Same proof ariticulated as failures

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

Same proof ariticulated 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 ariticulated 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 ariticulated 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 ariticulated 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

The intersection of CFL ***need not be CF!!***

•

$$\begin{aligned} L_{ab} &= \{\mathtt{a}^n\mathtt{b}^n\mathtt{c}^k \mid n,k \geqslant 0\} \quad \text{is CF} \\ L_{bc} &= \{\mathtt{a}^k\mathtt{b}^n\mathtt{c}^n \mid n,k \geqslant 0\} \quad \text{is CF} \end{aligned}$$

• But their interscetion

$$L_{ab} \cap L_{bc} = \{\mathtt{a}^n\mathtt{b}^n\mathtt{c}^n \mid n \geqslant 0\}$$

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 Languae 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

- $\{a^i b^j c^i \mid i, j \geqslant 0\}$ 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 $w = a^k b^k c^k d^k \in L$. $w \in L,\ |w| \geqslant k$.

Example: Alternating equals

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

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.

NONDETERMINISTIC STACK ACCEPTORS (PDAs)

A missing computation model

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

DFA = Deterministic Finite Acceptor

A missing computation model

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

NFA = Non-deterministic Finite Acceptor

A missing computation model

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

A missing computation model

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

NSA = Non-deterministic Stack Acceptor

A missing computation model

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

PDA = Push-Down Automata

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 varienties 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 $\{a^n b^n \mid n > 0\}$

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

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

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

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

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

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

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

- 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 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 $\mathtt{b}$ is followed by $\mathtt{a}$
  then computation aborts: no production for p reading $\mathtt{a}$.

PDA semantics: configurations and yield

- A $\boxed{\textit{configuration}}$ of a PDA is a triplet (q, w, α)
  where $q \in Q,\ w \in \Sigma^*$ and $\alpha \in \Gamma^*$.

- The intent:

  q is the current state

  w is the remaining portion of the input (from cursor on)

  α is a string representing the stack, from top to bottom.

PDA semantics: configurations and yield

- A $\boxed{\textit{configuration}}$ of a PDA is a triplet (q, w, α) where $q \in Q,\ w \in \Sigma^*$ and $\alpha \in \Gamma^*$.
- The intent:
  q is the current state
  w is the remaining portion of the input (from cursor on)
  α is a string representing the stack, from top to bottom.
- The transition rules generate
  a yield relation $\Rightarrow$ between configurations:
  If $q \xrightarrow{\sigma\,(\alpha \to \beta)} p$
  then $(q,\ \sigma x,\ \alpha \cdot \gamma) \quad \Rightarrow \quad (p,\ x,\ \beta \cdot \gamma)$
  (for all $x \in \Sigma^*$ and $\gamma \in \Gamma^*$).

PDA semantics: recognized languages

- The $\boxed{\textit{initial configuration}}$ for input w is (s, w, ε)

PDA semantics: recognized languages

- The $\boxed{\textit{initial configuration}}$ for input w is (s, w, ε)
- An input string $w \in \Sigma^*$ is $\boxed{\textit{accepted}}$ if
  $(s, w, \varepsilon) \;\Rightarrow^*\; (a, \varepsilon, \gamma)$
  for some accepting state $a \in A$ and some $\gamma \in \Gamma^*$.

PDA semantics: recognized languages

- The $\boxed{\textit{initial configuration}}$ for input w is (s, w, ε)
- An input string $w \in \Sigma^*$ is $\boxed{\textit{accepted}}$ if
  $(s, w, \varepsilon) \ \Rightarrow^* \ (a, \varepsilon, \gamma)$
  for some accepting state $a \in A$ and some $\gamma \in \Gamma^*$.
- A cfg $c = (q, w, \gamma)$ is $\boxed{\textit{terminal}}$ if
  there is no cfg c' where $c \Rightarrow_M c'$.
  if in addition $q \in A \quad w = \varepsilon$ then it is $\boxed{\textit{accepting.}}$

Examples of traces

Recall the transitions

$$
\begin{aligned}
s &\xrightarrow{a\,(\epsilon \to a\$)} q &\qquad p &\xrightarrow{b\,(a \to \epsilon)} p \\
q &\xrightarrow{a\,(\epsilon \to a)} q &\qquad p &\xrightarrow{\epsilon\,(\$ \to \epsilon)} f \\
q &\xrightarrow{b\,(a \to \epsilon)} p & &
\end{aligned}
$$

Examples of traces

Recall the transitions

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

A trace for **aabb**:

$$\begin{array}{rcl} (s, \mathtt{\textbf{aabb}}, \varepsilon) & \Rightarrow & (q, \mathtt{\textbf{abb}}, \mathtt{a}\$) \\ & \Rightarrow & (q, \mathtt{\textbf{bb}}, \mathtt{aa}\$) \\ & \Rightarrow & (p, \mathtt{\textbf{b}}, \mathtt{a}\$) \\ & \Rightarrow & (p, \varepsilon, \$) \\ & \Rightarrow & (f, \varepsilon, \varepsilon) \end{array}$$

Examples of traces

Recall the transitions

$$
\begin{array}{lclclcl}
s & \xrightarrow{a\,(\epsilon \to a\$)} & q & \qquad & p & \xrightarrow{b\,(a \to \epsilon)} & p \\
q & \xrightarrow{a\,(\epsilon \to a)} & q & \qquad & p & \xrightarrow{\epsilon\,(\$ \to \epsilon)} & f \\
q & \xrightarrow{b\,(a \to \epsilon)} & p & & & &
\end{array}
$$

Non-accepting traces:

$$
\begin{aligned}
(s, \mathbf{aab}, \varepsilon) &\Rightarrow (q, \mathbf{ab}, \mathtt{a}\$) \\
&\Rightarrow (q, \mathbf{b}, \mathtt{aa}\$) \\
&\Rightarrow (p, \varepsilon, \mathtt{a}\$)
\end{aligned}
\qquad
\begin{aligned}
(s, \mathbf{abb}, \varepsilon) &\Rightarrow (q, \mathbf{bb}, \mathtt{a}\$) \\
&\Rightarrow (q, \mathbf{b}, \$)
\end{aligned}
$$

Example: Palindromes around $\texttt{c}$

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

Example: Palindromes around $\mathtt{c}$

- 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 $\mathtt{c}$ switch to a new state,
  match subsequent input symbols with the top of the stack,
  popping the top.

Example: Palindromes around $\texttt{c}$

- Construct a PDA to recognize $\{w \cdot \texttt{c} \cdot w^R \mid w \in \{\mathtt{a}, \mathtt{b}\}^*\}$
- **Algorithm:** Push successive input symbols.
  When reading $\texttt{c}$ 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)

$q \xrightarrow{\texttt{c}\,(\epsilon \to \epsilon)} p$ (if $\texttt{c}$, 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 $q \xrightarrow{c\,(\epsilon \to \epsilon)} p$ above by
  by $q \xrightarrow{\epsilon\,(\epsilon \to \epsilon)} p$
- The resulting PDA:

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

Repeated use of nondeterminism

- Consider $\{\mathtt{a}^n\,\mathtt{b}^m \in \Sigma^* \mid m \leqslant n \leqslant 2m\}$
- 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:

$$
\begin{array}{llll}
s \xrightarrow{\epsilon\,(\epsilon\to\$)} q & \qquad & p \xrightarrow{b\,(a\to\epsilon)} p \\
q \xrightarrow{a\,(\epsilon\to a)} q & & p \xrightarrow{b\,(aa\to\epsilon)} p \\
q \xrightarrow{\epsilon\,(\epsilon\to\epsilon)} p & & p \xrightarrow{\epsilon\,(\$\to\epsilon)} f
\end{array}
$$

- 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 $\mathcal{L}(G) = \mathcal{L}(M)$.
- Motivating example:
  G has rules $\mathtt{S} \to \mathtt{aSb}$ and $\mathtt{S} \to \varepsilon$.
- 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 the rules of G deep in 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	$\mathtt{a}S\mathtt{b}\$$	
compare			
	abb	$S\mathtt{b}\$$	
			generate
	abb	$\mathtt{a}S\mathtt{bb}\$$	
compare			
	bb	$S\mathtt{bb}\$$	
			generate
	bb	$\mathtt{bb}\$$	
compare			
	b	$\mathtt{b}\$$	
compare			
	ε	$\$$	

PDAs recognize all CFLs

- Let $G = (R, N, S)$ be a CFG over Σ.
  Define a PDA M to recognize $\mathcal{L}(G)$.

PDAs recognize all CFLs

- Let $G = (R, N, S)$ be a CFG over Σ.
  Define a PDA M to recognize $\mathcal{L}(G)$.
- Three states: s, q and f.
  s initial, f accepting.

  Auxiliary symbols: variables of G and $\$$.

PDAs recognize all CFLs

- Let $G = (R, N, S)$ be a CFG over Σ.
  Define a PDA M to recognize $\mathcal{L}(G)$.
- Three states: s, q and f.
  s initial, f accepting.
  Auxiliary symbols: variables of G and $\$$.
- Transition rules:
  - Initializing the stack: $s \xrightarrow{\epsilon(\epsilon \to S\$)} q$

PDAs recognize all CFLs

- Let $G = (R, N, S)$ be a CFG over Σ.
  Define a PDA M to recognize $\mathcal{L}(G)$.
- Three states: s, q and f.
  s initial, f accepting.
  Auxiliary symbols: variables of G and $\$$.
- Transition rules:
  - Initializing the stack: $s \xrightarrow{\epsilon(\epsilon \to S\$)} q$
  - For each production $A \to \alpha$: $q \xrightarrow{\epsilon(A \to \alpha)} q$
    I.e., if stack-top is variable A, apply a production of G.

PDAs recognize all CFLs

- Let $G = (R, N, S)$ be a CFG over Σ.
  Define a PDA M to recognize $\mathcal{L}(G)$.
- Three states: s, q and f.
  s initial, f accepting.
  Auxiliary symbols: variables of G and $\$$.
- Transition rules:
  - Initializing the stack: $s \xrightarrow{\epsilon(\epsilon \to S\$)} q$
  - For each production $A \to \alpha$: $q \xrightarrow{\epsilon(A \to \alpha)} q$
    I.e., if stack-top is variable A, apply a production of G.
  - For each $\sigma \in \Sigma$: $q \xrightarrow{\sigma(\sigma \to \epsilon)} q$
    I.e., if stack-top is a terminal σ *matching current input symbol,*
    then σ is read off input, and popped off the stack.

– Acceptance: $q \xrightarrow{\epsilon(\$ \rightarrow \epsilon)} f$

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 and the corresponding trace of M:

	$(s, \texttt{aabb}, \varepsilon)$	
S	$(q, \texttt{aabb}, S\$)$	
$\texttt{aSb}$	$(q, \texttt{aabb}, \mathrm{a}S\mathrm{b}\$)$	generate
	$(q, \texttt{abb}, S\mathrm{b}\$)$	
$\texttt{aaSbb}$	$(q, \texttt{abb}, \texttt{aSbb}\$)$	generate
	$(q, \mathrm{bb}, \texttt{Sbb}\$)$	
$\texttt{aabb}$	$(q, \mathrm{bb}, \mathrm{bb}\$)$	generate
	$(q, \mathrm{b}, \mathrm{b}\$)$	
	$(q, \varepsilon, \$)$	
	$(f, \varepsilon, \varepsilon)$	

Converting PDAs to CFGs

- We already had a conversion from NFAs to regular expressions.
- For pairs (q, p) of states we assigned the language of strings leading from q to p via deleted states.
- A pre-processor guaranteed that the language assigned to the pair (s, a) (i.e. *start* to *accept* is the language recognized by the given NFA.
- For pairs (q, p) of states let L_{qp} consist of the strings w leading from q with an empty stack to p with an empty stack:
  $$L_{qp} = \{w \in \Sigma^* \mid (q, w, \varepsilon) \Rightarrow^* (p, \varepsilon, \varepsilon)\}$$
- Note that if $(q, w, \varepsilon) \Rightarrow (p, \varepsilon, \varepsilon)$ then $(q, w, \alpha) \Rightarrow (p, w, \alpha)$ for all stack α.

A pre-processor

- Converting NFA to equivalent RegExp we pre-processed.
- Here convert given PDA M to one that

  1. has all stack operations broken push and pop of one symbol;
  2. accepts a string only when the stack is empty.

- (1) helps us restrict attention to basic changes in the stack.

  (2) enables focusing on traces that start and end with empty stack.

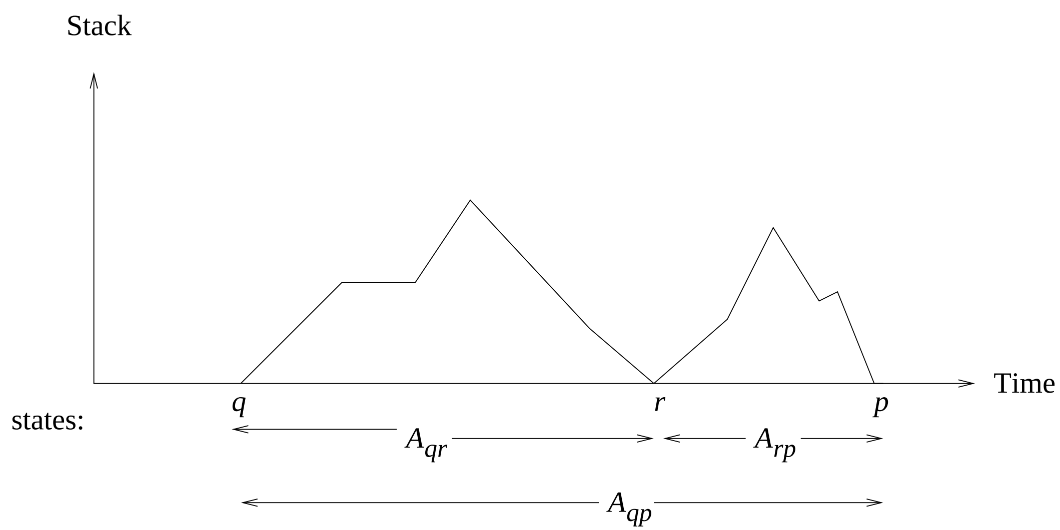
- A PDA M can be converted into an equivalent one satisfying (1) by breaking compound $u_0 \to v_0$ into single-letter push and pop.
- (2) is obtained by adding transitions that empty the stack when M accepts.

Generating simultanuously the languages L_{qp}

- We use productions to code a generative definition of the languages L_{qp} .
- Right off we have, for each state q, $(q, \varepsilon) \xrightarrow{\epsilon} (q, \varepsilon)$.
  I.e. $\varepsilon \in L_{qq}$.
- So we include in our grammar, for each state q,
  the production $A_{qq} \to \varepsilon$.

Concatenation

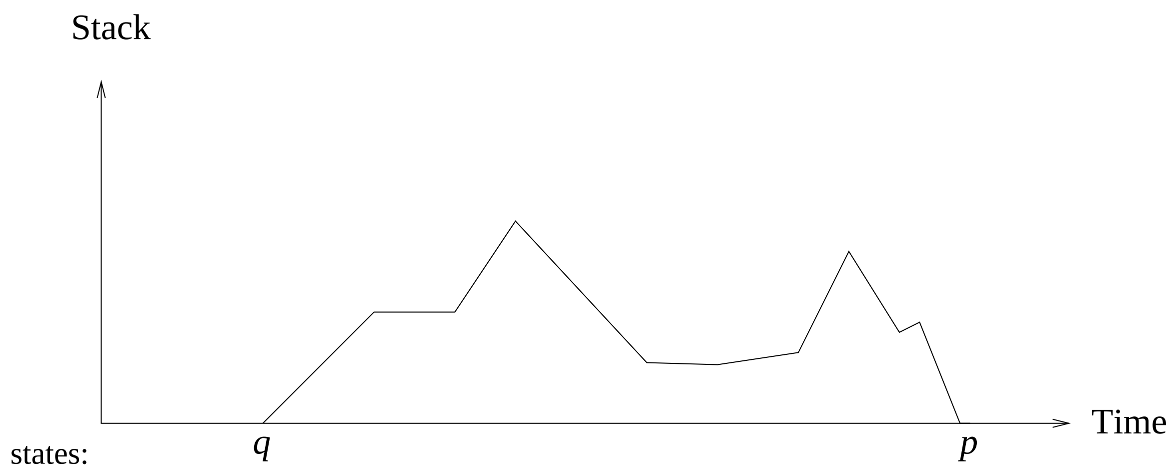
- If $(q, \varepsilon) \xrightarrow{u} (r, \varepsilon) \xrightarrow{v} (p, \varepsilon)$ then $(q, \varepsilon) \xrightarrow{u \cdot v} (p, \varepsilon)$.
- In other words, if we already have that
  $A_{qr} \Rightarrow^* u$ and $A_{rp} \Rightarrow^* v$,
  then we should have $A_{qp} \Rightarrow^* u \cdot v$.
- This is achieved by including the production $A_{qp} \to A_{qr}\, A_{rp}$



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

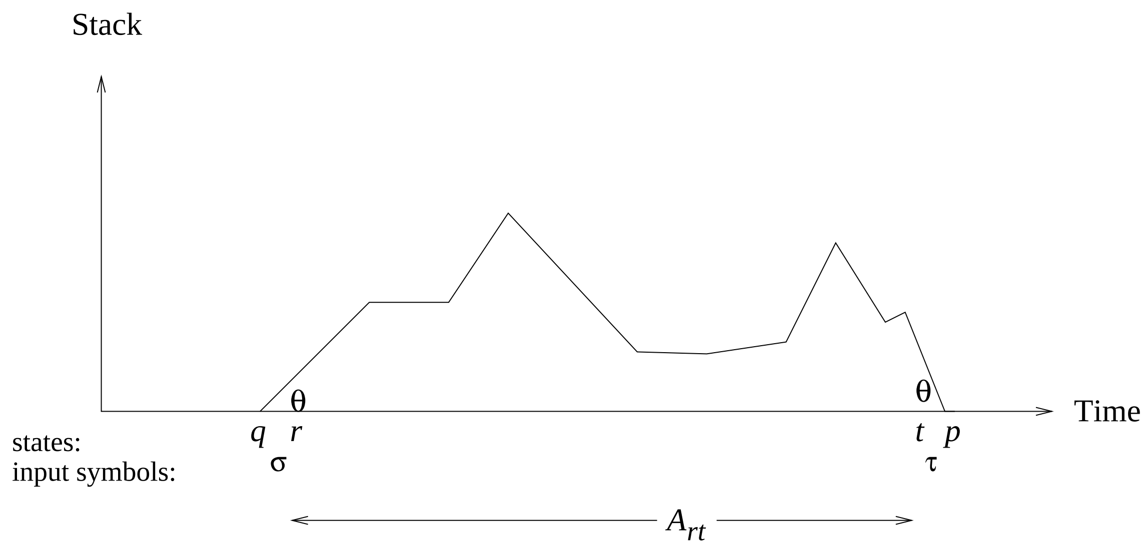
Productions for stack operations

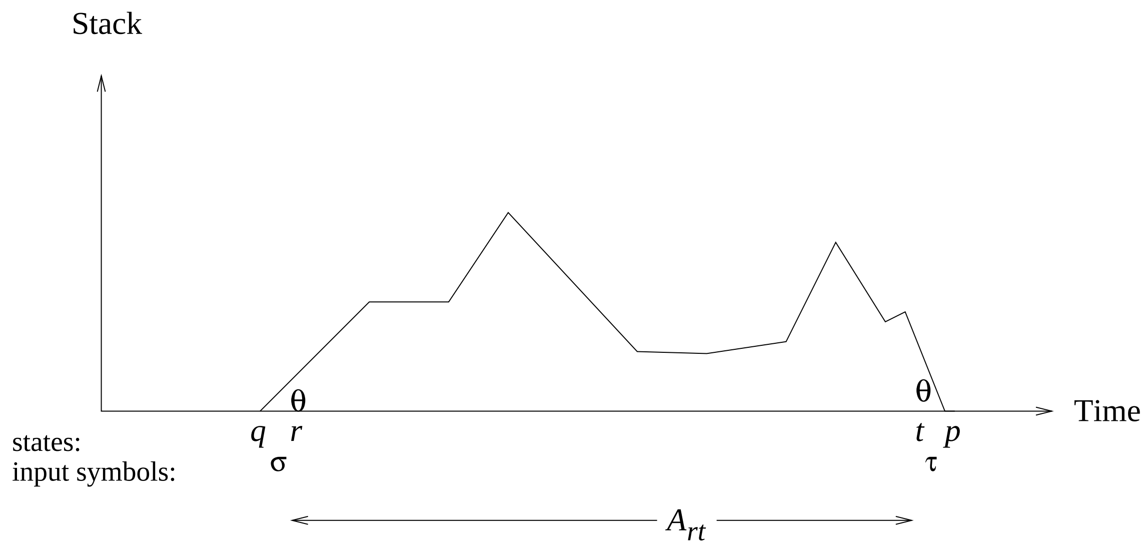
- So far we have looked at productions that apply to any PDA.
- Suppose $(q, w, \varepsilon) \Rightarrow^* (p, \varepsilon, \varepsilon)$.
  If the computation trace has an empty stack along the way,
  i.e. a configuration (r, v, ε) with $w = u \cdot v$,
  then the concatenation production will yield w.
- 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.
- The last move must then read some symbol $\tau \in \Sigma_\epsilon$ which causes M to pop that θ (which is undisturbed through the trace). That is, for some states r, t:

$$\begin{aligned} (q, \sigma v, \varepsilon) &\Rightarrow (r, v, \theta) \\ (t, \tau, \theta) &\Rightarrow (p, \varepsilon, \varepsilon) \end{aligned}$$





- This is conveyed by the production $A_{qp} \to \sigma A_{rt} \tau$.
- In general, whenever M has 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 has the production $A_{qp} \to \sigma A_{rt} \tau$.

Proof concluded

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

$$A_{qp} \Rightarrow_G^* w \quad \text{IFF} \quad (q, w, \varepsilon) \rightarrow_M^* (p, \varepsilon, \varepsilon)$$

- When q, p are the initial and accepting states s, f
  $A_{sf} \Rightarrow_G^* w$ (G generates w) iff
  $(s, w, \varepsilon) \rightarrow_M^* (f, \varepsilon, \varepsilon)$ (M accepts w),

Example

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

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

- The construction above yields the following grammar

$A_{tt} \to \varepsilon$ (all states t)

$A_{tu} \to A_{tv}\, A_{vu}$ (all states t, u, v)

(with initial variable A_{sf}) $A_{qr} \to \mathtt{a}\, A_{qr}\, \mathtt{b}$ (pushing and popping $\mathtt{a}$, rules 2 and

$A_{qr} \to \mathtt{c}\, A_{pp}\, \varepsilon$ (pushing and popping $\mathtt{b}$, rules 3 and

$A_{sf} \to \varepsilon\, A_{qr}\, \varepsilon$ (pushing and popping $\$$, rules 1 and

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.

   (Why is this example a bit silly?)

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. Suppose $L \subseteq \Gamma^*$ is recognized by a PDA.

   If $\Sigma \subset \Gamma$, what about the set of Σ-strings in L?

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 $\{\mathtt{a}^n\mathtt{b}^n\mathtt{c}^n \mid n \geqslant 0\} = L \cap \mathcal{L}(\mathtt{a}^* \cdot \mathtt{b}^* \cdot \mathtt{c}^*)$

   So L cannot be CF.

2. Suppose $L \subseteq \Gamma^*$ is recognized by a PDA.

   If $\Sigma \subset \Gamma$, what about the set of Σ-strings in L?

   It is $L \cap \Sigma^*$, and therefore CF.

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	stack
ACCESS:	on-line	on-line + stack

Revisiting our non-CF grammar

$$S \rightarrow \varepsilon \mid SABC$$

$$AB \rightarrow BA \qquad BA \rightarrow AB$$

$$AC \rightarrow CA \qquad CA \rightarrow AC$$

$$BC \rightarrow CB \qquad CB \rightarrow BC$$

$$A \rightarrow \mathtt{a}$$

$$B \rightarrow \mathtt{b}$$

$$C \rightarrow \mathtt{c}$$

- $\mathcal{L}(G) = \{w \in \{a,b,c\}^* \mid \#_a(w) = \#_b(w) = \#_c(w)\}$ is not context free.

The context-sensitive languages

- A grammar is $\boxed{\textbf{\textit{context sensitive}}}$ (a CSG) if all its productions are of the form $uAv \rightarrow uxv$.
- This is just like a CFG, except that rules $A \rightarrow x$ may be restricted to a *context* $u \cdots v$, where u, v are strings of gterminals.
- These are the $\boxed{\textbf{\textit{context-sensitive languages (CSL's)}}}$.
- **Theorem.**
  A language is context-sensitive iff it is recognized by an LBA.

A larger table

LANGUAGE CLASS:	**Regular**	**CFL**	**CSL**
GRAMMARS:	regular	CF	CS
MACHINES:	DFA=NFA	NFA + stack	LBA
MEMORY:	internal	stack	on-site
ACCESS:	on-line	on-line + stack	two-way

LANGUAGE CLASS:	**Regular**	**Context-free**	**Context-sensitive**
GRAMMARS:	regular grammars	CF grammars	CS grammars
MACHINES:	DFA=NFA	NFA + stack	LBA
MEMORY:	internal	stack	on-site
ACCESS:	on-line	on-line + stack	two-way
SMTH NEW:		$\mathrm{a}^n\mathrm{b}^n$	$\mathrm{a}^n\mathrm{b}^n\mathrm{c}^n$

- This is a *strict hierarchy:*
  every level contains the previous plus more.