Formal Language and Automata Theory (CS21004)

Soumyajit Dey
CSE, IIT Kharagpur

Announcements

- The slide is just a short summary
- Follow the discussion and the boardwork
- Solve problems (apart from those we dish out in class)

Table of Contents

$$
\begin{array}{rcl}
\langle \text{stmt} \rangle & \models & \langle \text{if-stmt} \rangle \mid \langle \text{while-stmt} \rangle \mid \langle \text{begin-stmt} \rangle \\
\langle \ldots \rangle & \models & \mid \langle \text{assg-stmt} \rangle \\
\langle \text{if-stmt} \rangle & \models & \textit{if} \ \langle \text{bool-expr} \rangle \ \textit{then} \ \langle \text{stmt} \rangle \ \textit{else} \ \langle \text{stmt} \rangle \\
\langle \text{while-stmt} \rangle & \models & \textit{while} \ \langle \text{bool-expr} \rangle \ \textit{do} \ \langle \text{stmt} \rangle \\
\langle \text{begin-stmt} \rangle & \models & \textit{begin} \ \langle \text{stmt-list} \rangle \ \textit{end} \\
\langle \text{stmt-list} \rangle & \models & \langle \text{stmt} \rangle \mid \langle \text{stmt} \rangle \ ; \langle \text{stmt-list} \rangle \\
\langle \text{assg-stmt} \rangle & \models & \langle \text{var} \rangle \\
\langle \text{var} \rangle & \models & \langle \text{arith-expr} \rangle \\
\langle \text{bool-expr} \rangle & \models & \langle \text{arith-expr} \rangle \langle \text{compare-op} \rangle \langle \text{arith-expr} \rangle \\
\langle \text{compare-op} \rangle & \models & < \mid > \mid \leq \mid \geq = \mid \neq \\
\langle \text{arith-expr} \rangle & \models & \langle \text{var} \rangle \mid \langle \text{const} \rangle \mid \langle \text{arith-expr} \rangle \langle \text{arith-op} \rangle \langle \text{arith-expr} \rangle \\
\langle \text{arith-op} \rangle & \models & + \mid - \mid * \mid / \\
\langle \text{const} \rangle & \models & 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9 \\
\langle \text{const} \rangle & \models & a \mid b \mid c \mid \cdots \mid x \mid y \mid z
\end{array}
$$

For the string
while $x \leq y$ do begin $x = x + 1; y = y - 1$ end
The first few sentential forms in its derivation are
$\langle \text{stmt} \rangle$
$\langle \text{while-stmt} \rangle$
while $\langle \text{bool-stmt} \rangle$ do $\langle \text{stmt} \rangle$
while $\langle \text{arith-expr} \rangle \langle \text{compare-op} \rangle \langle \text{arith-expr} \rangle$ do $\langle \text{stmt} \rangle$
while $\langle \text{var} \rangle \langle \text{compare-op} \rangle \langle \text{arith-expr} \rangle$ do $\langle \text{stmt} \rangle$
while $\langle \text{var} \rangle \leq \langle \text{arith-expr} \rangle$ do $\langle \text{stmt} \rangle$
while $\langle \text{var} \rangle \leq \langle \text{var} \rangle$ do $\langle \text{stmt} \rangle$
while $\langle x \rangle \leq \langle \text{var} \rangle$ do $\langle \text{stmt} \rangle$
while $\langle x \rangle \leq \langle y \rangle$ do $\langle \text{stmt} \rangle$
while $\langle x \rangle \leq \langle y \rangle$ do $\langle \text{begin-stmt} \rangle$

Formally, a *context-free grammar* (CFG) is a quadruple
$G = (N, \Sigma, P, S)$
where

- N is a finite set (the *non-terminal symbols*),
- Σ is a finite set (the *terminal symbols*) disjoint from N,
- P is a finite subset of $N \times (N \cup \Sigma)^*$ (the *productions*), and
- $S \in N$ (the *start symbol*)

- $\{a^n b^n \mid n \geq 0\}$ is a CFL with CFG $S \rightarrow aSb \mid \epsilon$
- Palindromes over $\{a, b\}$ is a CFL with CFG $S \rightarrow aSa \mid bSb \mid a \mid b \mid \epsilon$
- Balanced parentheses over $\{(,)\}$ is a CFL with CFG $S \rightarrow (S) \mid SS \mid \epsilon$

How do you prove that each grammar generates the corresponding language ??

Table of Contents

It helps to restrict formal grammars to 'standard forms'. You can write algorithms that work on grammars assuming this standard

- A CFG is in *Chomsky Normal Form* (CNF) if all productions are of the form
  1. $A \to BC$ or
  2. $A \to a$

  where $A, B, C \in N$ and $a \in \Sigma$. Note that CNF form grammars cannot generate ϵ
- For any CFG G, there is a CFG G' in Chomsky Normal Form such that

$$L(G') = L(G) - \{\epsilon\}$$

Create grammar for ϵ free language

- For any CFG $G = (N, \Sigma, P, S)$, there is a CFG G' with no ϵ or unit productions $(A \rightarrow B)$ such that

$$L(G') = L(G) - \{\epsilon\}$$

.

- Proof: Let $\hat{P}$ be the smallest set of productions containing P and closed under the rules
  1. if $A \rightarrow \alpha B \beta$ and $B \rightarrow \epsilon$ are in $\hat{P}$; then $A \rightarrow \alpha\beta$ is in $\hat{P}$
  2. if $A \rightarrow B$ and $B \rightarrow \gamma$ are in $\hat{P}$, then $A \rightarrow \gamma$ is in $\hat{P}$
- Point to note : $\alpha, \beta, \gamma \in (N \cup \Sigma)^*$

Create grammar for ϵ free language

Note that for the language $\hat{G} = (N, \Sigma, \hat{P}, S)$, $L(G) = L(\hat{G})$

- $L(G) \subseteq L(\hat{G})$ since $P \subseteq \hat{P}$
- $L(G) = L(\hat{G})$: $\hat{P}$ only contains those rules as extra over P which are simulatable in two steps by existing rules of P

Create grammar for ϵ free language

Can prove that, for any $x \neq \epsilon$, $\exists$ a derivation $S \underset{\hat{G}}{\Rightarrow} x$ without ϵ and *unit* productions

- Remove ϵ and *unit* productions from $\hat{G}$

Create $G' = (N, \Sigma, P', S)$ where P' is same as $\hat{P}$ but w/o ϵ and *unit* productions

G' to CNF

- For each terminal $a \in \Sigma$ introduce a new nonterminal A_a and production $A_a \to a$ and replace all occurrences of a on the right hand side of old productions (except productions of the form $B \to a$) with A_a
- Now all productions are of the form
  1. $A \to a$ or
  2. $A \to B_1 B_2 \cdots B_k, k \geq 2$

  where the B_i are nonterminals.

G' to CNF

- For any production

$$A \rightarrow B_1 B_2 \cdots B_k$$

with $k \geq 3$, introduce a new nonterminal C and replace this production with the two productions

1. $A \rightarrow B_1 C$ and
2. $C \rightarrow B_2 B_3 \cdots B_k$

until all right hand sides are of length at most 2.

- Derive a CNF grammar for

$$\{a^n b^n | n \geq 0\} - \{\epsilon\} = \{a^n b^n | n \geq 1\}$$

- Starting with the grammar

$$S \rightarrow aSb|\epsilon$$

1. Remove ϵ productions to get $S \rightarrow aSB|ab$
2. Add non terminals A,B and replace the productions with $S \rightarrow ASB|AB, A \rightarrow a, B \rightarrow b$.
3. Add a nonterminal C and replace $S \rightarrow ASB$ with $S \rightarrow AC$ and $C \rightarrow SB$.

- The final grammar in CNF is $S \rightarrow AB|AC, C \rightarrow SB, A \rightarrow a, B \rightarrow b$.

- Derive a CNF grammar for the set of non-null strings of balanced parenthesis [ ]
- Starting with the grammar

$$S \rightarrow [S] | SS | \epsilon$$

1. Remove ϵ productions to get $S \rightarrow [S] | SS | [\,]$
2. Add non terminals A,B and replace the productions with $S \rightarrow SS | ASB | AB, A \rightarrow [, B \rightarrow]$.
3. Add a nonterminal C and replace $S \rightarrow ASB$ with $S \rightarrow AC$ and $C \rightarrow SB$.
4. The final grammar in CNF is $S \rightarrow SS | AB | AC, C \rightarrow SB, A \rightarrow [, B \rightarrow]$.

Table of Contents

- Consider the grammar $G = (\{S, A, B, C\}, \{a, b, c\}, S, P)$ where $P = \{S \rightarrow ABC, A \rightarrow aA, A \rightarrow \epsilon, B \rightarrow bB, B \rightarrow \epsilon, C \rightarrow cC, C \rightarrow \epsilon\}$.
- With this grammar, there is a choice of variables to expand. If we always expanded the leftmost variable first, we would have a *leftmost derivation*: $S \rightarrow ABC \rightarrow aABC \rightarrow aBC \rightarrow abBC \rightarrow abbBC \rightarrow abbC \rightarrow abbcC \rightarrow abbc$
- Conversely, if we always expanded the rightmost variable first, we would have a *rightmost derivation*: $S \rightarrow ABC \rightarrow ABcC \rightarrow ABc \rightarrow AbBc \rightarrow AbbBc \rightarrow Abbc \rightarrow aAbbc \rightarrow abbc$

- A grammar G is ambiguous if $\exists w \in L(G)$ for which
  - there are two or more distinct derivation/parse trees, or
  - there are two or more distinct leftmost derivations, or
  - there are two or more distinct rightmost derivations.
- Ambiguity is a property of a grammar, and it is usually (but not always) possible to find an equivalent unambiguous grammar.
- An *inherently ambiguous language* is a language for which no unambiguous grammar exists.

- Consider $S \rightarrow AS|\epsilon, A \rightarrow A1|0A1|01$. The string 00111 has the following two leftmost derivations from S:
  1. $S \rightarrow AS \rightarrow 0A1S \rightarrow 0A11S \rightarrow 00111S \rightarrow 00111$
  2. $S \rightarrow AS \rightarrow A1S \rightarrow 0A11S \rightarrow 00111S \rightarrow 00111$
- Intuitively, we can use $A \rightarrow A1$ first or second to generate the extra 1. The language of our example grammar is not inherently ambiguous, even though the grammar is ambiguous.
- Change the grammar to force the extra 1's to be generated last. $S \rightarrow AS|\epsilon, A \rightarrow 0A1|B, B \rightarrow B1|01$
- $\{a^i b^j c^k | i = j \text{ or } j = k\}$ is an inherently ambiguous language

Table of Contents

For every CFL A, $\exists k \geq 0$ such that $\forall z \in A$ with $|z| \geq k$ $\exists u, v, w, x, y$ such that

- $z = uvwxy$
- $vx \neq \epsilon$
- $|vwx| \leq k$
- $\forall i \geq 0,\ uv^iwx^iy \in A$

Intuitive (informal) idea :

- The set of a regular language grows in *one direction* per production : standard form is strictly right/left linear
- A Context Free language grows in *two directions* per production : Consider as standard form the CNF representation
- For sufficiently long strings, non terminals will repeat
- The subtree under the higher nonterminal instance can be copied for the lower instance
- This creates two separate possibilities of pumping

CNF grammar for $\{a^n b^n | n \geq 1\}$:
$S \rightarrow AB|AC, C \rightarrow SB, A \rightarrow a, B \rightarrow b.$

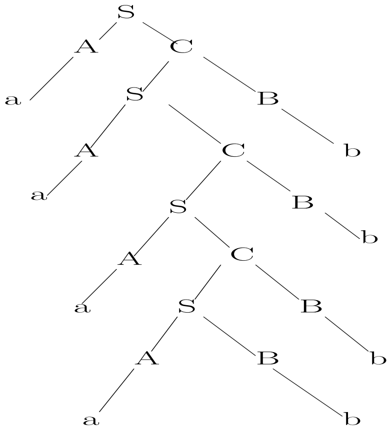


Figure: derivation tree for $a^4 b^4$:

Map the tree to uniform depth in all branches

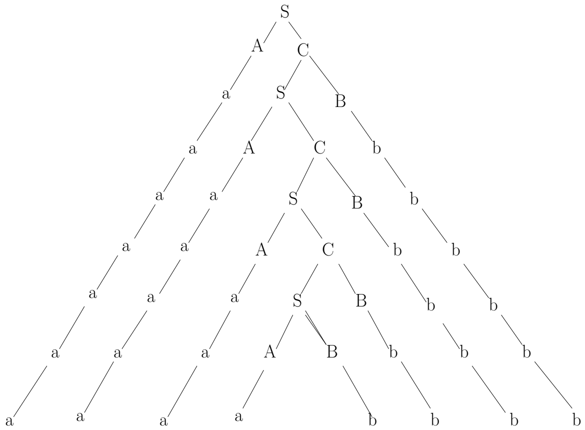
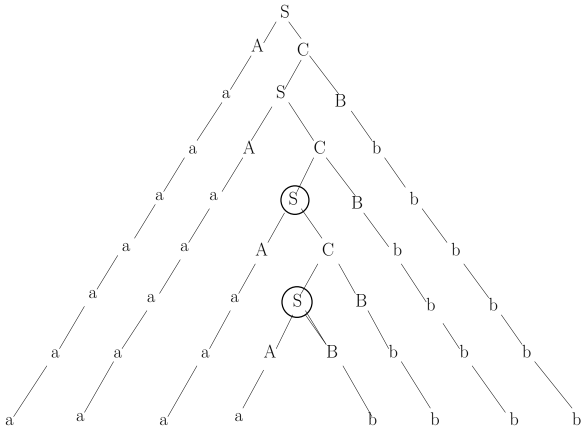
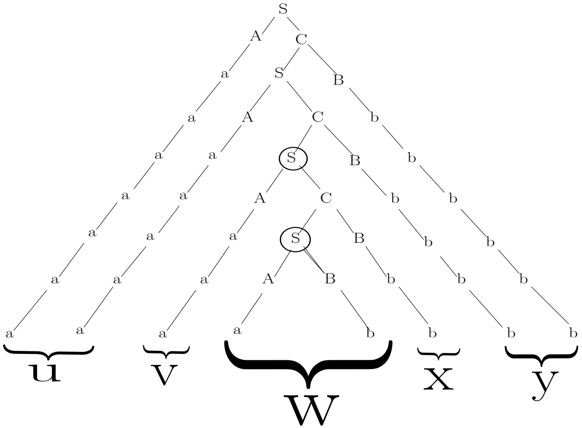


Figure: derivation tree for a^4b^4 :

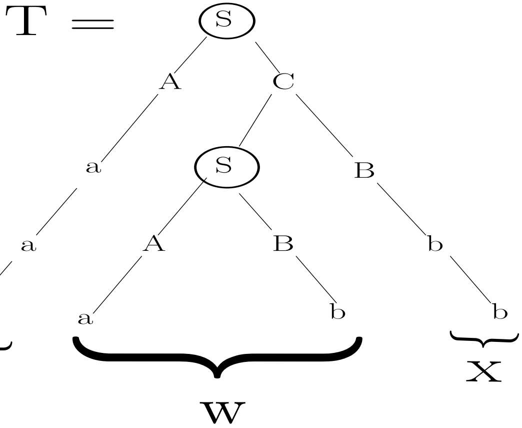
Consider a path with a repeating non-terminal (S here)



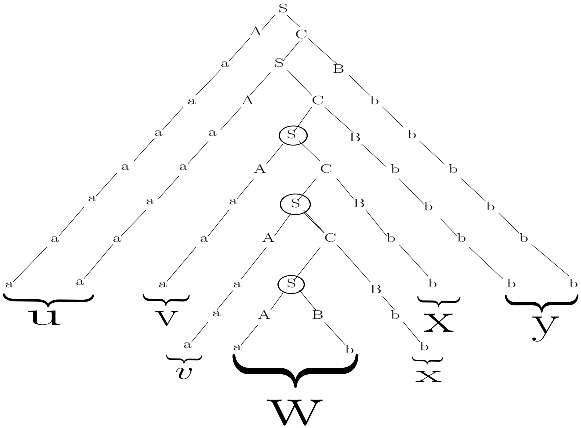
Note the decomposition of the string



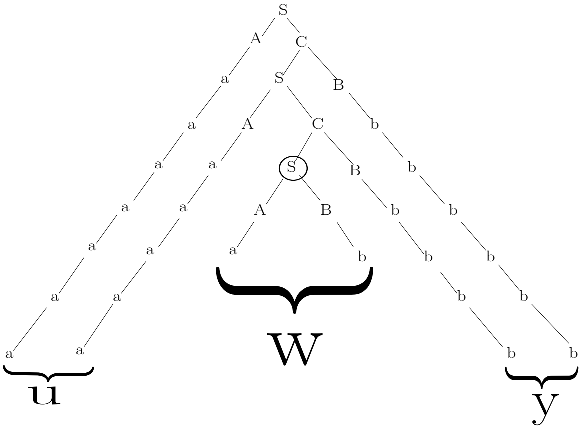
Check the subtree with two occurrences of S



Pumping up



Pumping down



Proof

- Let G be a grammar for A in CNF.
- $k = 2^{n+1}$, where n is the number of nonterminals of G
- Suppose $z \in A$ and $|z| \geq k$
- Any parse tree in G for z must be of depth $n + 1$
- The longest path in the tree is of length at least $n + 1$
- It must contain at least $n + 1$ occurrences of nonterminals.
- By pigeonhole principle, some nonterminal must occur more than once

- Say X is the nonterminal that appears more than once
- Break z up into substrings $uvwxy$ such that
-
  1. w is the string generated by the lower occurrence of X
  2. vwx is the string generated by the upper occurrence
- Let T and t be the subtree rooted at the upper and lower ocurrence of X respectively
- Replacing t with T once we get a valid subtree of uv^2wx^2y
- Repeat it to produce $uv^iwx^iy, i \geq 1$
- Replace T with t to get uv^0wx^0y

- Note that $vx \neq \epsilon$; otherwise $T = t$
- Also $|vwx| \leq k$ as we chose the first repeated occurrence of a nonterminal from the bottom.
- In the longest path, the depth of the subtree under the upper occurrence of X is at most $n + 1$
- So it cannot have more than $2^{n+1} = k$ terminals.

$A = \{a^n b^n a^n | n \geq 0\}$ is not context free

- Adversary picks k in step 1
- You pick $z = a^k b^k a^k$ such that $z \in A$ and $|z| = 3k \geq k$
- Adversary picks $z = uvwxy$ such that $vx \neq \epsilon$, and $|vwx| \leq k$
- You pick $i = 2$ and in all cases you can ensure $uv^2wx^2y \notin A$
  1. either v or x contain at least one a and at least one b
  2. v and x contains only a's or only b's
  3. one of v or x contains only a's and the other contains only b's

$A = \{ww | w \in \{a, b\}^*\}$ is not context free

- As CFLs are closed under intersection with regular sets, it suffices to show that

$$A' = A \cap L(a^* b^* a^* b^*) = \{a^n b^m a^n b^m | m, n \geq 0\}$$

  is not context-free
- Adversary picks k in step 1
- You pick $z = a^k b^k a^k b^k$ such that $z \in A'$ and $|z| \geq k$.
- call each of the four substrings of the form a^k or b^k as blocks
- Adversary picks $z = uvwxy$ such that $vx \neq \epsilon$, and $|vwx| \leq k$
- You pick $i = 2$ and in all cases you can win

- If one of v or x contains both a's and b's then $uv^2wx^2y \notin A'$
- If v and x are from the same block, then uv^2wx^2y has one block longer than the other three
- If v and x are on different blocks, then the blocks must be adjacent; otherwise $|vwx|$ would be greater than k. Thus one of the blocks containing v or x must be a block of a's and the other a block of b's. Then uv^2wx^2y either has two blocks of a's of different size (if vx contains an a) or two blocks of b's of different size (if vx contains a b) or both. Either way $uv^2wx^2y \notin A'$.

Table of Contents

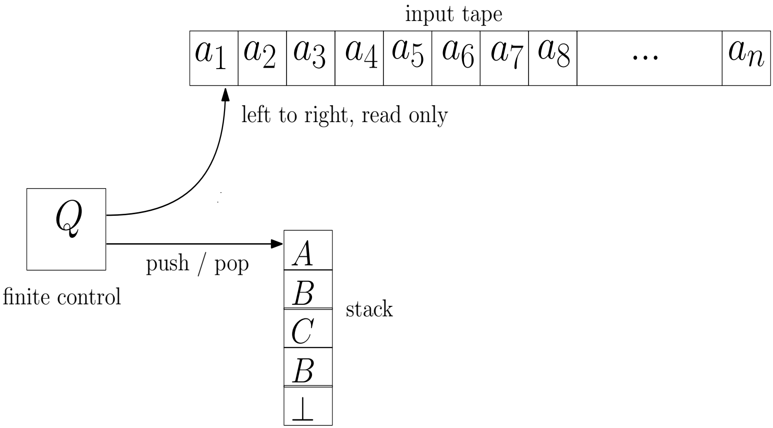


Figure: Non deterministic Pushdown Automata

A non deterministic PDA is a 7 tuple
$M = (Q, \Sigma, \Gamma, \delta, s, \perp, F)$,

- Q is a finite set (the states),
- Σ is a finite set (the input alphabet),
- Γ is a finite set (the stack alphabets),
- $s \in Q$ (the start state),
- $\perp \in \Gamma$ (the initial stack symbol), and
- $F \subseteq Q$(the final or accept states),
- $\delta \subseteq (Q \times (\Sigma \cup \varepsilon) \times \Gamma) \times (Q \times \Gamma^*), \delta$: NPDA has ϵ transitions (can move w/o input)

$((p, a, A), (q, B_1 \cdots B_k)) \in \delta$: from a state p while reading some input symbol a with some stack top element A, the PDA moves to another state q, pops A, pushes $B_1 \cdots B_k$ (B_k first)

- $((p, a, A), (q, B_1 \cdots B_k)) \in \delta$: from a state p while reading some input symbol a with some stack top element A, the PDA moves to another state q, pops A, pushes $B_1 \cdots B_k$ (B_k first)
- $((p, \epsilon, A), (q, B_1 \cdots B_k)) \in \delta$: from a state p with some stack top element A, the PDA moves to another state q, pops A, pushes $B_1 \cdots B_k$ (B_k first) w/o reading any input symbol

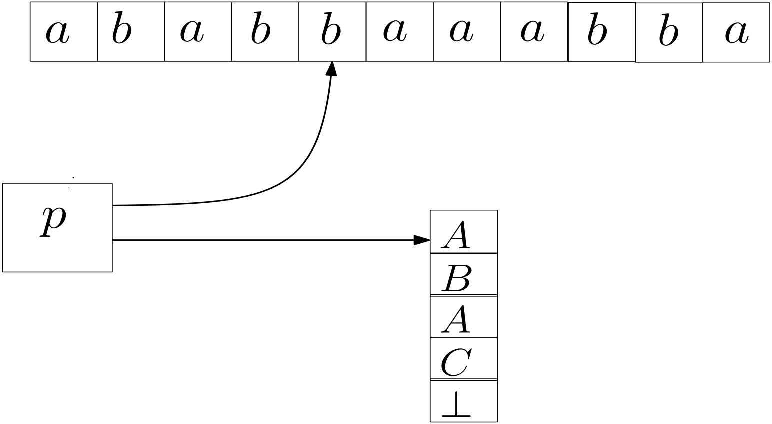


Figure: Example of Non deterministic Pushdown Automata

Config : $(p, baaabba, ABAC\bot)$: (state, unread tape, stack content)

- The start configuration on input x is $(s, x, \perp)$. That is, the machine always starts in its start state s with its read head pointing to the leftmost input symbol and the stack containing only the symbol $\perp$.
- The next configuration relation $\xrightarrow[M]{1}$ describes how the machine can move from one configuration to another in one step.
- If $((p, a, A), (q, \gamma)) \in \delta$, then for any $y \in \Sigma^*$ and $\beta \in \Gamma^*$, $(p, ay, A\beta) \xrightarrow[M]{1} (q, y, \gamma\beta)$;
- and if $((p, \varepsilon, A), (q, \gamma)) \in \delta$,then for any $y \in \Sigma^*$ and $\beta \in \Gamma^*$, $(p, y, A\beta) \xrightarrow[M]{1} (q, y, \gamma\beta)$

Acceptance

- by **final state** : $(s, x, \perp) \xrightarrow[M]{*} (q, \epsilon, \gamma)$: get to some $q \in F$ when string exhausted
- by **empty stack** : $(s, x, \perp) \xrightarrow[M]{*} (q, \epsilon, \epsilon)$: pop stack bottom element when string exhausted

Both kinds of M/Cs can be converted to an equivalent NPDA M that has a single final state t and M can empty its stack after it enters state t.

PDA for balanced parenthesis

- $Q = \{q\}$,
- $\Sigma = \{[,]\}$,
- $\Gamma = \{\perp, [\}$,
- start state $= q$
- initial stack symbol $= \perp$,
- and let δ consist of the following transitions:
  1. $((q, [, \perp), (q, [\perp))$;
  2. ((q,[,[), (q,[[));
  3. $((q,], [), (q, \varepsilon))$;
  4. $((q, \varepsilon, \perp), (q, \varepsilon))$.

	Configuration		Transition
	(q,	[[[ ]] [ ]] [ ], $\perp$)	start configuration
$\rightarrow$	(q,	[[ ]] [ ]] [ ], [$\perp$)	transition (1)
$\rightarrow$	(q,	[ ]] [ ]] [ ], [[$\perp$)	transition (2)
$\rightarrow$	(q,	]] [ ]] [ ], [[[$\perp$)	transition (2)
$\rightarrow$	(q,	] [ ]] [ ], [[$\perp$)	transition (3)
$\rightarrow$	(q,	[ ]] [ ], [$\perp$)	transition (3)
$\rightarrow$	(q,	]] [ ], [[$\perp$)	transition (2)
$\rightarrow$	(q,	] [ ], [$\perp$)	transition (3)
$\rightarrow$	(q,	[ ], $\perp$)	transition (3)
$\rightarrow$	(q,	], [$\perp$)	transition (1)
$\rightarrow$	(q,	ϵ, $\perp$)	transition (2)
$\rightarrow$	(q,	ϵ, ϵ)	transition (4)

Design an NPDA for $\{a, b\}^* - \{ww \mid w \in \{a, b\}^*\}$

CFG to NPDA

All productions of $G = (\Sigma, N, P, S)$ are of the form: $A \rightarrow cB_1B_2.....B_k$, Where $c \in \Sigma \cup \{\epsilon\}$ and $k \geq 0$. An equivalent NPDA M with only one state that accepts by empty stack is $M = (\{q\}, \Sigma, N, \delta, q, S, \emptyset)$ where

- q is the only state.
- Σ is the input alphabet of M
- N is the set of stack alphabet of M
- δ is the transition relation
- q is the start state
- S, is the start symbol of G, is the initial stack symbol of M
- $\emptyset$, the null set, is the set of final states

Conversion Rule

- For each production $A \rightarrow cB_1B_2.....B_k$ in P
- The transition relation δ is defined as $((q, c, A), (q, B_1B_2....B_k))$
  1. δ has one transition of each production of G
  2. When in state q scanning input symbol c with A on top of the stack, pop A off the stack, push $B_1B_2.....B_k$ onto the stack (B_k first) and enter in q state
  3. For $c = \epsilon$, when in state q with A on top of the stack, without scanning an input symbol, pop A off the stack, push $B_1B_2.....B_k$ onto the stack (B_k first) and enter in q state

Example Conversion

Consider the set of nonnull balanced strings of parentheses [ ]. The list of production rules of the grammar and corresponding transition of NPDA for this set are as follows:

1. $S \rightarrow [BS$ $((q, [, S), (q, BS))$
2. $S \rightarrow [B$ $((q, [, S), (q, B))$
3. $S \rightarrow [SB$ $((q, [, S), (q, SB))$
4. $S \rightarrow [SBS$ $((q, [, S), (q, SBS))$
5. $B \rightarrow]$ $((q,], B), (q, \epsilon))$

Example Derivation for the i/p 'x = [[[ ]] [ ]]'

Rule applied	*Sentential forms in a leftmost* derivation of x in G	*Configuration of M in an accepting computation of M on input x*
	S	(q, [[[ ]] [ ]], S)
(3)	[SB	(q, [[ ]] [ ]], SB)
(4)	[[SBSB	(q, [ ]] [ ]], SBSB)
(2)	[[[BBSB	(q,]] [ ]], BBSB)
(5)	[[[ ]BSB	(q,] [ ]], BSB)
(5)	[[[ ]]SB	(q, [ ]], SB)
(2)	[[[ ]] [BB	(q,]], BB)
(5)	[[[ ]] [ ]B	(q,], B)
(5)	[[[ ]] [ ]]	(q, ϵ, ϵ)

Lemma: For any $z, y \in \Sigma^*, \gamma \in N^*$, and $A \in N, A \xrightarrow[G]{n} z\gamma$ via a leftmost derivation iff $(q, zy, A) \xrightarrow[M]{n} (q, y, \gamma)$

- For example, in the fourth row of the table above, we would have z = [[[, y =]] [ ]], γ = BBSB, A = S and n=3
- *Proof:* Try Yourself

CFG G converted to NPDA M

Language Equivalence : $L(M) = L(G)$
Proof:

$$x \in L(G)$$

$\Leftrightarrow S \xrightarrow[G]{*} x$ by a leftmost derivation. [definition of L(G)]

$\Leftrightarrow (q, x, S) \xrightarrow[M]{*} (q, \epsilon, \epsilon)$ using the last Lemma

$\Leftrightarrow x \in L(M)$ definition of L(M)

NPDA to CFG

- Every NPDA can be simulated by an NPDA with one state.
- Every NPDA with one state has an equivalent CFG

Conclusion : With NPDA $\rightarrow$ CFG and CFG $\rightarrow$ NPDA we have established that 'NPDAs and CFGs are equivalent in expressive power'

Single state NPDA to CFG

- Consider a NPDA M with one state that accepts by empty stack
  $M = (\{q\}, \Sigma, \Gamma, \delta, q, \perp, \emptyset)$
- The equivalent CFG be $G = (\Gamma, \Sigma, P, \perp)$, where P contains a production $A \rightarrow cB_1B_2...B_k$ for every transition $((q, c, A), (q, B_1B_2...B_k)) \in \delta$

(the conversion we discussed is invertible).

Arbitrary NPDA to single state NPDA

Consider a NPDA $M = (Q, \Sigma, \Gamma, \delta, s, \perp, \{t\})$

- M has a single final state t and M can empty its stack after it enters state t.
- Let $\Gamma' \stackrel{def}{=} Q \times \Gamma \times Q$. Elements of Γ' are written $\langle pAq \rangle$, where $p, q \in Q$ and $A \in \Gamma$

Equivalent NPDA $M' = (\{*\}, \Sigma, \Gamma', \delta', *, \langle s \perp t \rangle, \emptyset)$

- M' has one state * and accepts by empty stack.
- $(*, x, pAq) \xrightarrow[M']{*} (*, \epsilon, \epsilon)$ iff $(p, x, A) \xrightarrow[M]{*} (q, \epsilon, \epsilon)$

For each transition $((p, c, A), (q_0, B_1 B_2 ... B_k)) \in \delta$, where $c \in \Sigma \cup \{\epsilon\}$

- $((*, c, \langle pAq_k \rangle), (*, \langle q_0 B_1 q_1 \rangle \langle q_1 B_2 q_2 \rangle .... \langle q_{k-1} B_k q_k \rangle)) \in \delta'$. For all possible choice of $q_1, q_2, ..., q_k$.
- For k=0, $((*, c, \langle pAq_0 \rangle), (*, \epsilon)) \in \delta'$ iff $((p, c, A), (q_0, \epsilon)) \in \delta$

M' nondeterministically guesses a computation sequence, saves the guessed sequence in stack
verifies later in case of an actual run.

NPDA to CFG - another option

Consider a PDA $P = (Q, \Sigma, \Gamma, \delta, q_0, \perp, \{q_F\})$ with the restriction that every transition either pushes a symbol or pops a symbol from the stack, i.e. $\delta(q, a, X)$ contains either $(q', \gamma X)$, $\gamma \in \Gamma$ or (q', ϵ). Consider the equivalent grammar $G_P = (V, T, P, S)$ such that $V = \{A_{p,q} : p, q \in Q\}$, $T = \Sigma$, $S = A_{q_0,q_F}$ and P has transitions of the following form:

- $\forall q \in Q \quad (A_{q,q} \rightarrow \epsilon) \in P$
- $\forall p, q, r \in Q \quad (A_{p,q} \rightarrow A_{p,r} A_{r,q}) \in P$
- $(A_{p,q} \rightarrow a A_{r,s} b) \in P$ if $\delta(p, a, \epsilon)$ contains (r, X) and $\delta(s, b, X)$ contains (q, ϵ)

NPDA to CFG

Theorem: *If $A_{p,q} \stackrel{*}{\Rightarrow} x$ then x can bring the PDA P from state p on empty stack to state q on empty stack.*

Proof:
We can prove this theorem by induction on the number of steps in the derivation of x from $A_{p,q}$.

NPDA to CFG

<u>*Base case*</u>: If $A_{p,q} \stackrel{*}{\Rightarrow} x$ in one step, then the only rule to generate a terminal string in one step is $A_{p,p} \rightarrow \epsilon$

<u>*Inductive step:*</u> If $A_{p,q} \stackrel{*}{\Rightarrow} x$ in n + 1 steps. The first step in the derivation must be $A_{p,q} \rightarrow A_{p,r}A_{r,q}$ or $A_{p,q} \rightarrow aA_{r,s}b$.

- If it is $A_{p,q} \rightarrow A_{p,r}A_{r,q}$, then the string x can be broken into two parts x_1x_2 such that $A_{p,r} \stackrel{*}{\Rightarrow} x_1$ and $A_{r,q} \stackrel{*}{\Rightarrow} x_2$ in at most n steps. The theorem easily follows in this case.
- If it is $A_{p,q} \rightarrow aA_{r,s}b$, then the string x can be broken as ayb such that $A_{r,s} \stackrel{*}{\Rightarrow} y$ in n steps. Notice that from state p, on reading a the PDA pushes a symbol X to stack, while it pops X in state s on reading b and goes to state q.

Table of Contents

The following grammar generates the well-parenthesized propositional expressions

$$
\begin{aligned}
E &\implies (EBE)|(UE)|C|V, \\
B &\implies \vee|\wedge|\rightarrow|\leftrightarrow, \\
U &\implies \neg, \\
C &\implies \top|\bot, \\
V &\implies P|Q|R|\cdots .
\end{aligned}
$$

The words well-parenthesized means that there must be parentheses around any compound expression. E.g.

$$(((P \vee Q) \wedge R) \vee (Q \wedge (\neg P)))$$

A parser

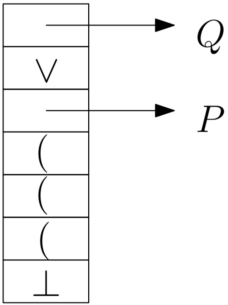
Start with the initial stack symbol $\perp$ and scan the expression from left to right.

1. If the symbol is a (, push it in the stack.
2. If the symbol is an operator, push it in the stack.
3. If the symbol is a constant, push it in the stack.
4. If the symbol is a variable, push it in the stack.
5. If the symbol is a), do a reduce
    1. Create a new node.
    2. Pop the top. It should be a constant, variable or another node.
    3. Pop the top. It should be an operand.
    4. Pop the top. It should be a constant, variable or another node.
    5. Pop the top. It should be a (.
    6. Push the node in the stack.

Parsing Example

$$(((P \vee Q) \wedge R) \vee (Q \wedge (\neg P)))$$

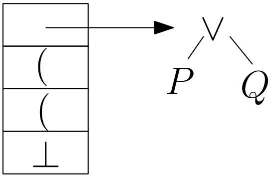
- Push the 3 ('s in the stack.(i)
- Push P in the stack.(iv)
- Push $\vee$ in the stack.(ii)
- Push Q in the stack.(iv)



Parsing Example contd.

$$(((P \vee Q) \wedge R) \vee (Q \wedge (\neg P)))$$

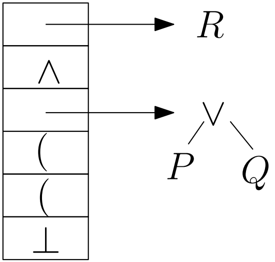
- Scan) and reduce.
- Pop the stack until (and create a new node.



Parsing Example contd.

$$(((P \lor Q) \land R) \lor (Q \land (\neg P)))$$

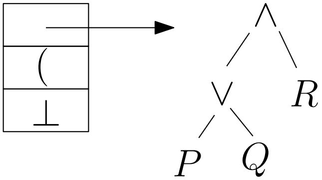
- Push $\land$ and R in the stack



Parsing Example contd.

$$(((P \vee Q) \wedge R) \vee (Q \wedge (\neg P)))$$

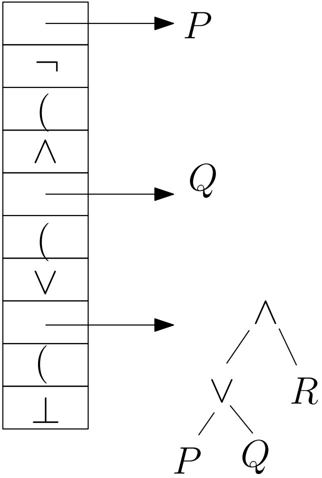
- Scan) and reduce.



Parsing Example contd.

$$(((P \vee Q) \wedge R) \vee (Q \wedge (\neg P)))$$

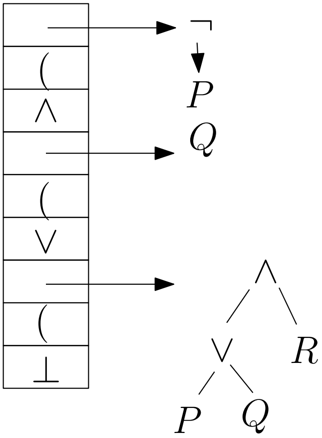
- Push everything until the next).



Parsing Example contd.

$$(((P \vee Q) \wedge R) \vee (Q \wedge (\neg P)))$$

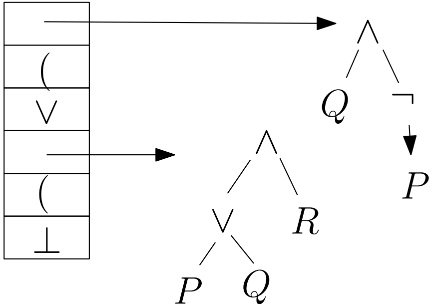
- Scan the first of the final 3) and reduce.



Parsing Example contd.

$$(((P \vee Q) \wedge R) \vee (Q \wedge (\neg P)))$$

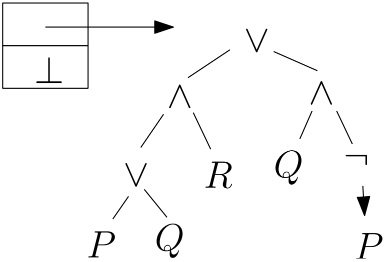
- Scan the next) and reduce.



Parsing Example contd.

$$(((P \vee Q) \wedge R) \vee (Q \wedge (\neg P)))$$

- Scan the final 3) and reduce.



Operator Precedence

Consider the grammar for arithmetic expressions.

$$E \to E + E|E - E|E \cdot E|E/E| - E|C|V|(E),$$
$$C \to 0|1,$$
$$V \to a|b|c.$$

This grammar is ambiguous. The equivalent unambiguous grammar is

$$E \to E + F|E - F|F,$$
$$F \to F \cdot G|F/G|G,$$
$$G \to -G|H,$$
$$H \to C|V|(E),$$
$$C \to 0|1,$$
$$V \to a|b|c.$$

Parsing Example

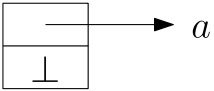
Given a precedence relation,on the operators, we modify the parsing rules as follows. When we scan a binary operator B

1. Check if top of the stack is an operand
2. Look at the stack symbol A immediately below
    - If A has lower precedence than B push B
    - Otherwise, reduce

Consider the following example

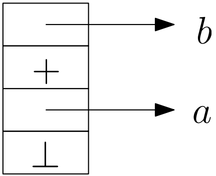
$$a + b.c + d$$

- Start with the stack containing $\perp$
- Scan and push a in the stack



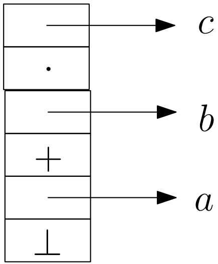
Parsing Example contd.

- Scan $+$. Check that $\perp$ has lower precedence so push $+$.
- Scan and push b



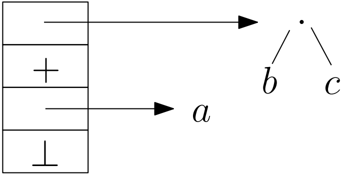
Parsing Example contd.

- Scan $\cdot$. Check that $+$ has lower precedence so push $\cdot$.
- Scan and push c

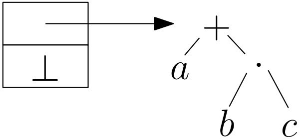


Parsing Example contd.

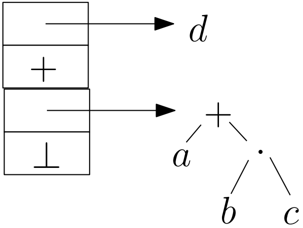
- Scan the second $+$. Check that $\cdot$ has higher precedence so reduce.



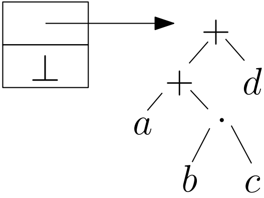
- Check that $+$ has equal precedence so reduce.



Parsing Example contd.



- Check that $\perp$ has lower precedence so push $+$.
- Scan and push d



- End of expression so reduce.

Parsing Example contd.

Why reduce in case of equal precedence ?

- Eventually you have to reduce everything
- No point keeping things in stack when it can be reduced
- Unnecessary Push n Pop operations in that case

Table of Contents

Closure Properties of CFL

Context-free languages are closed under the following operations:

1. Union
2. Concatenation
3. Kleene closure
4. Homomorphism
5. Substitution
6. Inverse-homomorphism
7. Reverse

Closure Properties of CFLs

- CFL are closed under intersection with regular sets.
- CFL are not closed under intersection.

Closure Properties of CFL

Context-free languages are **not closed** *under intersection and complementation.*

Proof:

Consider the languages

$L_1 = \{0^n 1^n 2^m : n, m \geq 0\}$ and

$L_2 = \{0^m 1^n 2^n : n, m \geq 0\}$

Both languages are CFLs.

What is $L_1 \bigcap L_2$?

Closure Properties of CFL

Proof: contd...

$$L = L_1 \bigcap L_2 = \{0^n 1^n 2^n : n \geq 0\} \text{ and it is not a CFL.}$$

Hence CFLs are not closed under **intersection**.

Use Demorgans law to prove non-closure under **complementation**.

Linear Grammars n languages

A linear grammar is a CFG that has at most one nonterminal in the R.H.S. of each of its productions.

- Consider $\{a^i b^i \mid i \geq 0\}$
- $P = \{S \to aSb, S \to \epsilon\}$

We call such languages as linear languages. Recall, the special cases

- **left-linear grammars** : a type of linear grammar with R.H.S. nonterminals strictly at the left ends;
- **right-linear grammars** : a type of linear grammar with R.H.S. nonterminals strictly at the right ends;

Linear Grammars n languages

- All linear languages are context-free
- There are CFLs that are non-linear : balanced parenthesis (the famous "Dyck language")
  $S \rightarrow (S)|SS|\epsilon$

Thus

- regular languages are a proper subset of linear languages
- linear languages are a proper subset of CFLs

Linear Grammars n languages

- Linear languages that are regular are deterministic (DPDA acceptable naturally)
- there exist linear languages that are nondeterministic

Ex : the language of even-length palindromes on $\{0, 1\}$ has the linear grammar $S \rightarrow 0S0|1S1|\epsilon$. Require NPDA

- linear languages are closed under intersection with regular sets
- linear languages are closed under homomorphism and inverse homomorphism.

Table of Contents

Deterministic Pushdown Automata

A PDA $P = (Q, \Sigma, \Gamma, \delta, q_0, \perp, F)$ is deterministic if

- $\delta(q, a, X)$ has at most one member for every $q \in Q$, $a \in \Sigma$ or $a = \epsilon$, and $X \in \Gamma$.
- If $\delta(q, a, X)$ is nonempty for some $a \in \Sigma$ then $\delta(q, \epsilon, X)$ must be empty.

Example: $L = \{0^n 1^n : n \geq 1\}$.

Deterministic Pushdown Automata

- Theorem:

  Every regular language can be accepted by a deterministic pushdown automata that accepts by final states.

- Theorem ($DPDA \neq PDA$):
  There are some CFLs, for instance $\{ww^R\}$ *that can not be accepted by a DPDA.*

Closure Properties of DCFLs

A deterministic context free language is a language accepted by a deterministic PDA(DPDA). Every DCFL is a CFL but not vice versa.

- DCFL are not closed under union
- DCFL are not closed under reversal
- DCFL are closed under complementation

Table of Contents

CYK Algorithm

Cubic-time Algorithm to check if a string x belongs to a grammar G in CNF.
Consider the following grammar for the set of all non-null strings with equal number of a's and b's.

$$
\begin{aligned}
S &\to AB|BA|SS|AC|BD, \\
A &\to a, \\
B &\to b, \\
C &\to SB, \\
D &\to SA.
\end{aligned}
$$

CYK Algorithm contd.

We'll run the algorithm on the string

	a		a		b		b		a		b	
0		1		2		3		4		5		6

For $0 \leq i < j \leq n$, x_{ij} denote the substring of x between lines i and j. Build a table T with $\binom{n}{2}$ entries, one for each pair i, j.

0						
–	1					
–	–	2				
–	–	–	3			
–	–	–	–	4		
–	–	–	–	–	5	
–	–	–	–	–	–	6

$T_{i,j}$ refers to substring $x_{i,j}$.

We will fill each entry $T_{i,j} \in T$ with the set of nonterminals of G that produce substring $x_{i,j}$. We start with substring of length 1. For each substring $c = x_{i,i+1}$, if there is a production $X \to c \in G$, we write X in $T_{i,j}$.

0						
A	1					
–	A	2				
–	–	B	3			
–	–	–	B	4		
–	–	–	–	A	5	
–	–	–	–	–	B	6

For each substring $c = x_{i,i+2}$, we break the substring into two non-null substrings $x_{i,i+1}$ and $x_{i+1,i+2}$ and check the table entries

0						
A	1					
ϕ	A	2				
–	S	B	3			
–	–	ϕ	B	4		
–	–	–	S	A	5	
–	–	–	–	S	B	6

Now proceed to strings of length 3. For each substring $c = x_{i,i+3}$, we have two ways to proceed: we break the substring into two non-null substrings $x_{i,i+1}$ and $x_{i+1,i+3}$ or $x_{i,i+2}$ and $x_{i+2,i+3}$. For example,$x_{0,3} = x_{0,1}x_{1,3} = x_{0,2}x_{2,3}$. For the first, $A \in T_{0,1}$ and $S \in T_{1,3}$ but AS is not present in the right side of any production. Similarly nothing produces $T_{0,2}$. So $T_{0,3}$ is ϕ

0						
A	1					
ϕ	A	2				
ϕ	S	B	3			
–	–	ϕ	B	4		
–	–	–	S	A	5	
–	–	–	–	S	B	6

For $x_{1,4} = x_{1,2}x_{2,4} = x_{1,3}x_{3,4}$, $A \in T_{1,2}$ and $\phi \in T_{2,4}$. But $S \in T_{1,3}$ and $B \in T_{3,4}$ and $C \to SB \in G$. So $T_{1,4}$ is labeled with C.

0						
A	1					
ϕ	A	2				
ϕ	S	B	3			
–	C	ϕ	B	4		
–	–	–	S	A	5	
–	–	–	–	S	B	6

Continue in this fashion for stings of length three,four etc. For strings of length four there are 3 ways to break them up and every one must be checked. the final result is

0						
A	1					
ϕ	A	2				
ϕ	S	B	3			
S	C	ϕ	B	4		
D	S	ϕ	S	A	5	
S	C	ϕ	C	S	B	6

We see that $T_{0,6}$ is labeled with S so we conclude that x is generated by G.

Algorithm 1: CYK Algorithm

```
for i := 0 to n − 1 do    /* 1 length strings first */
    T_{i,i+1} := Φ                    /* Initialize to Φ */
    for A → a ∈ G do
        if a = x_{i,i+1} then
            T_{i,i+1} := T_{i,i+1} ∪ {A}
for m := 2 to n do        /* for each length m ≥ 2 */
    for i := 0 to n − m do    /* for each substring */
        T_{i,i+m} := Φ                /* of length m */
        for j := i + 1 to i + m − 1 do  /* for all ways
         to breakup the string */
            for A → BC ∈ G do
                if B ∈ T_{i,j} ∧ C ∈ T_{j,i+m} then
                    T_{i,i+m} := T_{i,i+m} ∪ {A}
```

Table of Contents

- An unrestricted grammar is a 4-tuple $G = (V, \Sigma, S, P)$, where V and Σ are disjoint sets of variables and terminals, respectively. S is an element of V called the start symbol, and P is a set of productions of the form

$$\alpha \to \beta$$

  where $\alpha, \beta \in (V \cup \Sigma)^*$ and α contains at least one variable.
- A context-sensitive grammar (CSG) is an unrestricted grammar in which no production is length-decreasing. In other words, every production is of the form $\alpha \to \beta$, where $|\beta| \geq |\alpha|$.
- A language is a context-sensitive language (CSL) if it can be generated by a context-sensitive grammar.

A CSG Generating $L = \{a^n b^n c^n | n \geq 1\}$

$$
\begin{aligned}
S &\rightarrow SABC|\mathcal{A}BC \\
BA &\rightarrow AB \\
CA &\rightarrow AC \\
CB &\rightarrow BC \\
\mathcal{A} &\rightarrow a \\
aA &\rightarrow aa \\
aB &\rightarrow ab \\
bB &\rightarrow bb \\
bC &\rightarrow bc \\
cC &\rightarrow cc
\end{aligned}
$$

The derivation for the string *aabbcc* is

$$
\begin{aligned}
S &\rightarrow aSBC \\
&\rightarrow aaBCBC \\
&\rightarrow aabCBC \\
&\rightarrow aabBCC \\
&\rightarrow aabbCC \\
&\rightarrow aabbcC \\
&\rightarrow aabbcc
\end{aligned}
$$