Notes on Context Free Grammars

Geoffrey Matthews

Department of Computer Science
Western Washington University

November 6, 2018

Readings

- http://www.cs.rochester.edu/~nelson/courses/csc_173/grammars/cfg.html
- http://en.wikipedia.org/wiki/Context-free_grammar
- http://en.wikipedia.org/wiki/Context-free_language
- http://en.wikipedia.org/wiki/Parsing
- http://en.wikipedia.org/wiki/Pushdown_automata
- http://en.wikipedia.org/wiki/LR_parser
- https://parasol.tamu.edu/~rwerger/Courses/434/lec12-sum.pdf
- http://www.cs.sunysb.edu/~cse350/slides/cfg3.pdf

String Substitution

- Start with the string $ABBA$
- If we make the substitutions $A \to a$ and $B \to b$
- $ABBA \Rightarrow aBBA \Rightarrow abBA \Rightarrow abbA \Rightarrow abba$
- $ABBA \Rightarrow^* abba$
- If we make the substitutions $A \to ab$ and $B \to ba$
- $ABBA \Rightarrow abBBA \Rightarrow abbaBA \Rightarrow abbabaA \Rightarrow abbabaab$
- $ABBA \Rightarrow^* abbabaab$
- If we make the substitutions $A \to bab$ and $B \to bbb$
- $ABBA \Rightarrow babBBA \Rightarrow babbbbBA \Rightarrow babbbbbbbA \Rightarrow babbbbbbbbab$
- $ABBA \Rightarrow^* babbbbbbbbab$

Formal Grammars

- A set of **terminals**, e.g. $\{$the,cat,sat,on,mat$\}$
- A set of **nonterminals**, or **variables**, e.g. $\{S, N\}$
- A special nonterminal, the **start symbol**, e.g. S
- A set of **production rules**:

$$\begin{aligned} S &\rightarrow \text{the } N \text{ sat on the } N \\ N &\rightarrow \text{cat} \\ N &\rightarrow \text{mat} \end{aligned}$$

- A **derivation** is any string we get by starting with the start symbol and repeatedly making a single substitution until we only have terminals.
- $S \Rightarrow$ the N sat on the $N \Rightarrow$ the cat sat on the $N \Rightarrow$ the cat sat on the mat
- $S \Rightarrow$ the N sat on the $N \Rightarrow$ the mat sat on the $N \Rightarrow$ the mat sat on the mat

Vertical bar means "or"

This grammar:

$$\begin{aligned} S &\rightarrow \text{the } N \text{ sat on the } N \\ N &\rightarrow \text{cat} \\ N &\rightarrow \text{mat} \end{aligned}$$

is equivalent to this grammar:

$$\begin{aligned} S &\rightarrow \text{the } N \text{ sat on the } N \\ N &\rightarrow \text{cat} \mid \text{mat} \end{aligned}$$

Rules can be recursive

$$
\begin{aligned}
S &\rightarrow S \text{ and } S \\
S &\rightarrow \text{the } N \text{ sat on the } N \\
N &\rightarrow \text{cat} \mid \text{mat}
\end{aligned}
$$

Context Free Grammar

A context free grammar is a grammar where all the rules are the following form:

$$S \to w$$

where S is a single nonterminal and w is a string of terminals and nonterminals.

Context Free Grammar Example 1

$$S \quad \rightarrow \quad aS \mid \epsilon$$

$$S \Rightarrow \epsilon$$

$$S \Rightarrow aS \Rightarrow aaS \Rightarrow aaaS \Rightarrow aaa$$

$$S \Rightarrow aS \Rightarrow aaS \Rightarrow aaaS \Rightarrow aaaaS \Rightarrow aaaaaS \Rightarrow aaaaa$$

Context Free Grammar Example 2

$$S \rightarrow Sa \mid \epsilon$$

$$S \Rightarrow \epsilon$$

$$S \Rightarrow Sa \Rightarrow Saa \Rightarrow Saaa \Rightarrow aaa$$

$$S \Rightarrow Sa \Rightarrow Saa \Rightarrow Saaa \Rightarrow Saaaa \Rightarrow aaaa$$

Context Free Grammar Example 3

$$
\begin{aligned}
S &\rightarrow ABC \\
A &\rightarrow a \\
B &\rightarrow b \\
C &\rightarrow c
\end{aligned}
$$

$$S \Rightarrow ABC \Rightarrow aBC \Rightarrow abC \Rightarrow abc$$

$$S \Rightarrow ABC \Rightarrow AbC \Rightarrow abC \Rightarrow abc$$

$$S \Rightarrow ABC \Rightarrow aBC \Rightarrow aBc \Rightarrow abc$$

$$S \Rightarrow ABC \Rightarrow ABc \Rightarrow Abc \Rightarrow abc$$

Context Free Grammar Example 4

$$
\begin{aligned}
S &\rightarrow AB \mid A \\
A &\rightarrow aA \mid a \\
B &\rightarrow Bb \mid b
\end{aligned}
$$

$$S \Rightarrow A \Rightarrow aA \Rightarrow aaA \Rightarrow aaaA \Rightarrow aaaa$$

$$S \Rightarrow AB \Rightarrow aAB \Rightarrow aaB \Rightarrow aaBb \Rightarrow aaBbb \Rightarrow aabbb$$

$$S \Rightarrow AB \Rightarrow aAB \Rightarrow aaAB \Rightarrow aaaB \Rightarrow aaaBb \Rightarrow aaaBbb \Rightarrow aaabbb$$

Context Free Grammar for Arithmetic Expressions

$$
\begin{array}{rcl}
E & \rightarrow & T \\
E & \rightarrow & E + E \\
E & \rightarrow & E * E \\
E & \rightarrow & (E) \\
T & \rightarrow & a \\
T & \rightarrow & b \\
T & \rightarrow & T0 \\
T & \rightarrow & T1
\end{array}
$$

$E \Rightarrow E + E \Rightarrow E * E + E \Rightarrow T * E + E \Rightarrow a * E + E \Rightarrow$
$a * T + E \Rightarrow a * b + E \Rightarrow a * b + T \Rightarrow a * b + a$
Note that T could have been reprsented by a regular language.

Context Free Grammar for Programming Language

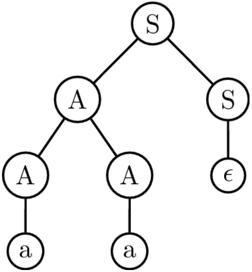
$$
\begin{array}{rcl}
S & \rightarrow & \textbf{while } E \textbf{ do } S \;\mid\; \textbf{if } E \textbf{ then } S \textbf{ else } S \;\mid\; I \textbf{ := } E \\
S & \rightarrow & \{\ SL\ \} \\
L & \rightarrow & SL\ \textbf{;}\ \mid \epsilon \\
E & \rightarrow & \ldots \\
I & \rightarrow & \ldots
\end{array}
$$

- Reference manuals for programming languages usually give the syntax of the language as a CFG.
- Note that keywords, punctuation, *etc.* can be represented by a regular language.

Derivations

- Start with S
- Find a rule for a nonterminal.
- Replace nonterminal with RHS.
- Until no more nonterminals.

$$
\begin{array}{rcl}
S & \rightarrow & AS \mid \epsilon \\
A & \rightarrow & AA \mid a
\end{array}
$$



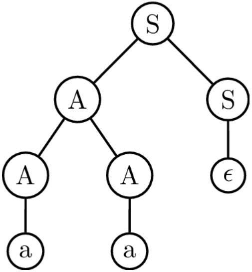
Parse Tree

$$S \Rightarrow AS \Rightarrow AAS \Rightarrow AA \Rightarrow Aa \Rightarrow aa$$

Derivations

- Start with S
- Find a rule for a nonterminal.
- Replace nonterminal with RHS.
- Until no more nonterminals.

$$
\begin{array}{rcl}
S & \rightarrow & AS \mid \epsilon \\
A & \rightarrow & AA \mid a
\end{array}
$$



Parse Tree

$$S \Rightarrow AS \Rightarrow AAS \Rightarrow AA \Rightarrow Aa \Rightarrow aa$$

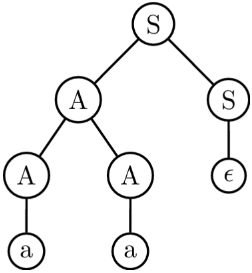
The **language of a grammar** is the set of all sentences for which there exists a derivation.

Derivations

- If there is more than one possible tree for some sentence, the grammar is **ambiguous**.
- There are usually many possible derivations, but only one tree.
- Important derivations are **leftmost** and **rightmost**.
- Leftmost and rightmost derivations also unique in unambiguous languages.

$$\begin{array}{lcl} S & \rightarrow & AS \mid \epsilon \\ A & \rightarrow & AA \mid a \end{array}$$

Leftmost:
$\underline{S} \Rightarrow \underline{A}S \Rightarrow \underline{A}AS \Rightarrow a\underline{A}S \Rightarrow aa\underline{S} \Rightarrow aa$
Proof that this grammar is ambiguous:
Rightmost:
$\underline{S} \Rightarrow A\underline{S} \Rightarrow AA\underline{S} \Rightarrow A\underline{A} \Rightarrow \underline{A}a \Rightarrow aa$
$\underline{S} \Rightarrow A\underline{S} \Rightarrow \underline{A} \Rightarrow A\underline{A} \Rightarrow \underline{A}a \Rightarrow aa$

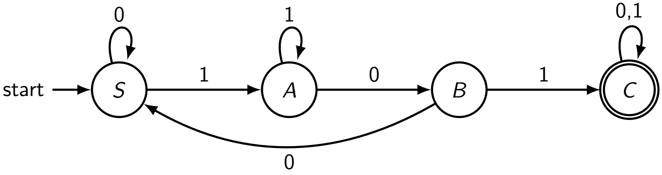


Parse Tree

CFG problems

- Any regular language.
- $a^n b^n$
- $a^n b^{2n}$
- $a^n b^{3n}$
- $a^{4n+5} b^{3n+2}$
- $a^n b^m a^n$
- Even length palindromes
- Odd length palindromes
- All palindromes
- All strings with the same number of a's and b's
- $\{a^i b^j c^k \mid i \neq j \text{ or } j \neq k\}$

All regular languages are context free languages



$$
\begin{aligned}
S &\rightarrow 0S \mid 1A \\
A &\rightarrow 0B \mid 1A \\
B &\rightarrow 0S \mid 1C \\
C &\rightarrow 0C \mid 1C \mid \epsilon
\end{aligned}
$$

Generate All Possible Sentences from a CF grammar

Length 1 derivations:

$$
\begin{aligned}
S &\rightarrow abS \mid bAc \mid d \\
A &\rightarrow aA \mid \epsilon
\end{aligned}
$$

$$
\begin{aligned}
S &\Rightarrow abS \\
S &\Rightarrow bAc \\
S &\Rightarrow d
\end{aligned}
$$

1. d

Generate All Possible Sentences from a CF grammar

Length 2 derivations:

$$\begin{aligned} S &\rightarrow abS \mid bAc \mid d \\ A &\rightarrow aA \mid \epsilon \end{aligned}$$

$$\begin{aligned} S &\Rightarrow abS \Rightarrow ababS \\ S &\Rightarrow abS \Rightarrow abbAc \\ S &\Rightarrow abS \Rightarrow abd \\ S &\Rightarrow bAc \Rightarrow baAc \\ S &\Rightarrow bAc \Rightarrow bc \end{aligned}$$

1. d
2. abd
3. bc

Generate All Possible Sentences from a CF grammar

$$
\begin{aligned}
S &\rightarrow abS \mid bAc \mid d \\
A &\rightarrow aA \mid \epsilon
\end{aligned}
$$

Length 3 derivations:

$$
\begin{aligned}
S &\Rightarrow abS \Rightarrow ababS \Rightarrow abababS \\
S &\Rightarrow abS \Rightarrow ababS \Rightarrow ababbAc \\
S &\Rightarrow abS \Rightarrow ababS \Rightarrow ababd \\
S &\Rightarrow abS \Rightarrow abbAc \Rightarrow abbaAc \\
S &\Rightarrow abS \Rightarrow abbAc \Rightarrow abbc \\
S &\Rightarrow bAc \Rightarrow baAc \Rightarrow baaAc \\
S &\Rightarrow bAc \Rightarrow baAc \Rightarrow bac
\end{aligned}
$$

1. d
2. abd
3. bc
4. $ababd$
5. $abbc$
6. bac
7. $\ldots$

Generate All Possible Sentences from a CF grammar

$$S \rightarrow abS \mid bAc \mid d$$
$$A \rightarrow aA \mid \epsilon$$

Length 3 derivations:

$$\begin{aligned}
S &\Rightarrow abS \Rightarrow ababS \Rightarrow abababS \\
S &\Rightarrow abS \Rightarrow ababS \Rightarrow ababbAc \\
S &\Rightarrow abS \Rightarrow ababS \Rightarrow ababd \\
S &\Rightarrow abS \Rightarrow abbAc \Rightarrow abbaAc \\
S &\Rightarrow abS \Rightarrow abbAc \Rightarrow abbc \\
S &\Rightarrow bAc \Rightarrow baAc \Rightarrow baaAc \\
S &\Rightarrow bAc \Rightarrow baAc \Rightarrow bac
\end{aligned}$$

1. d
2. abd
3. bc
4. $ababd$
5. $abbc$
6. bac
7. $\ldots$

Given a grammar G, is there an algorithm to find out if $s \in L(G)$?

Generate All Possible Sentences from a CF grammar

Length 3 derivations:

$$\begin{aligned} S &\rightarrow abS \mid bAc \mid d \\ A &\rightarrow aA \mid \epsilon \end{aligned}$$

$$\begin{aligned}
S &\Rightarrow abS \Rightarrow ababS \Rightarrow abababS \\
S &\Rightarrow abS \Rightarrow ababS \Rightarrow ababbAc \\
S &\Rightarrow abS \Rightarrow ababS \Rightarrow ababd \\
S &\Rightarrow abS \Rightarrow abbAc \Rightarrow abbaAc \\
S &\Rightarrow abS \Rightarrow abbAc \Rightarrow abbc \\
S &\Rightarrow bAc \Rightarrow baAc \Rightarrow baaAc \\
S &\Rightarrow bAc \Rightarrow baAc \Rightarrow bac
\end{aligned}$$

1. d
2. abd
3. bc
4. $ababd$
5. $abbc$
6. bac
7. $\ldots$

Given a grammar G, is there an algorithm to find out if $s \in L(G)$?

Almost, but not quite.

Generate All Possible Sentences from a CF grammar

$$
\begin{aligned}
S &\rightarrow abS \mid bAc \mid d \\
A &\rightarrow aA \mid \epsilon
\end{aligned}
$$

Length 3 derivations:

$$
\begin{aligned}
S &\Rightarrow abS \Rightarrow ababS \Rightarrow abababS \\
S &\Rightarrow abS \Rightarrow ababS \Rightarrow ababbAc \\
S &\Rightarrow abS \Rightarrow ababS \Rightarrow ababd \\
S &\Rightarrow abS \Rightarrow abbAc \Rightarrow abbaAc \\
S &\Rightarrow abS \Rightarrow abbAc \Rightarrow abbc \\
S &\Rightarrow bAc \Rightarrow baAc \Rightarrow baaAc \\
S &\Rightarrow bAc \Rightarrow baAc \Rightarrow bac
\end{aligned}
$$

1. d
2. abd
3. bc
4. $ababd$
5. $abbc$
6. bac
7. ...

Given a grammar G, is there an algorithm to find out if $s \in L(G)$?

Almost, but not quite.

We can tell if $s \in L(G)$ but what if $s \notin L(G)$?

Removing ϵ from grammars

$$\begin{array}{rcl} S & \rightarrow & aDaE \\ D & \rightarrow & bD \mid E \\ E & \rightarrow & cE \mid \epsilon \end{array}$$

- Find all nonterminals N such that $N \Rightarrow^* \epsilon$.
- Make new rules from old by removing one or more of the null nonterminals.
- Remove all null productions $N \rightarrow \epsilon$.
- May have to keep $S \rightarrow \epsilon$, but only if $\epsilon \in L(S)$.

Removing ϵ from grammars

$$
\begin{aligned}
S &\rightarrow aDaE \\
D &\rightarrow bD \mid E \\
E &\rightarrow cE \mid \epsilon
\end{aligned}
$$

- Null nonterminals: D and E

Original Production	New Productions
$S \rightarrow aDaE$	$S \rightarrow aaE \mid aDa \mid aa$
$D \rightarrow bD$	$D \rightarrow b$
$D \rightarrow E$	$D \rightarrow \epsilon$
$E \rightarrow cE$	$E \rightarrow c$
$E \rightarrow \epsilon$	none

Final grammar:

$$
\begin{aligned}
S &\rightarrow aDaE \mid aaE \mid aDa \mid aa \\
D &\rightarrow bD \mid b \mid E \\
E &\rightarrow cE \mid c
\end{aligned}
$$

Chomsky Normal Form

- All rules must be in one of these forms:

$$
\begin{aligned}
A &\rightarrow BC \\
A &\rightarrow a \\
S &\rightarrow \epsilon
\end{aligned}
$$

- A, B and C are nonterminals, a is a single terminal, and S is the start symbol.
- The last rule is necessary only if the language contains ϵ.

Converting to Chomsky Normal Form

1. Add S_0, a new start symbol, and the rule $S_0 \to S$.
   - Only necessary if S is recursive
2. Eliminate ϵ rules.
   - Except possibly $S_0 \to \epsilon$
3. Eliminate **unit** rules, $A \to B$:
   - Find all rules $B \to W$, where W is a string longer than one.
   - Add $A \to W$ for all of them.
4. Fix longer rules:
   - Replace $A \to UVWXYZ$ with
   - $A \to UA_1$, $A_1 \to VA_2$, $A_2 \to WA_3$, $A_3 \to XA_4$, $A_4 \to YZ$.
5. For each terminal x, add a rule $X \to x$ and replace all terminals in long (length two) strings with the corresponding nonterminals.

Example converting to Chomsky Normal Form

$$\begin{aligned} S &\rightarrow aSb \mid T \\ T &\rightarrow cT \mid \epsilon \end{aligned}$$

Step 1:
$$\begin{aligned} S_0 &\rightarrow S \\ S &\rightarrow aSb \mid T \\ T &\rightarrow cT \mid \epsilon \end{aligned}$$

Step 2:
$$\begin{aligned} S_0 &\rightarrow S \mid \epsilon \\ S &\rightarrow aSb \mid ab \mid T \\ T &\rightarrow cT \mid c \end{aligned}$$

Step 3:
$$\begin{aligned} S_0 &\rightarrow aSb \mid ab \mid cT \mid c \mid \epsilon \\ S &\rightarrow aSb \mid ab \mid cT \mid c \\ T &\rightarrow cT \mid c \end{aligned}$$

Step 4:
$$\begin{aligned} S_0 &\rightarrow aD \mid ab \mid cT \mid c \mid \epsilon \\ S &\rightarrow aD \mid ab \mid cT \mid c \\ D &\rightarrow Sb \\ T &\rightarrow cT \mid c \end{aligned}$$

Step 5:
$$\begin{aligned} S_0 &\rightarrow AD \mid AB \mid CT \mid c \mid \epsilon \\ S &\rightarrow AD \mid AB \mid CT \mid c \\ D &\rightarrow SB \\ T &\rightarrow CT \mid c \\ A &\rightarrow a \\ B &\rightarrow b \\ C &\rightarrow c \end{aligned}$$

Note on Eliminating Unit Rules

If we have two mutually recursive unit rules, for example in the following grammar:

$$\begin{array}{rcl} A & \rightarrow & B \mid XY \mid UVW \\ B & \rightarrow & A \mid DE \mid FGH \end{array}$$

It doesn't hurt to eliminate both of them, so long as the remainder of the clauses is added to each:

$$\begin{array}{rcl} A & \rightarrow & XY \mid UVW \mid DE \mid FGH \\ B & \rightarrow & DE \mid FGH \mid XY \mid UVW \end{array}$$

Any derivation that used the two unit rules, such as

$$A \Rightarrow B \Rightarrow A \Rightarrow B \Rightarrow FGH$$

Can be replaced by a short-circuited version

$$A \Rightarrow FGH$$

Consequences of Chomsky Normal Form

- What do all trees look like?

Consequences of Chomsky Normal Form

- What do all trees look like?
  - Binary trees

Consequences of Chomsky Normal Form

- What do all trees look like?
  - Binary trees
- What do all derivations look like?

Consequences of Chomsky Normal Form

- What do all trees look like?
  - Binary trees
- What do all derivations look like?
  - $S \Rightarrow AB \Rightarrow ABC \Rightarrow ABCD \Rightarrow aBCD \Rightarrow abCD \Rightarrow abcD \Rightarrow abcd$

Consequences of Chomsky Normal Form

- What do all trees look like?
  - Binary trees
- What do all derivations look like?
  - $S \Rightarrow AB \Rightarrow ABC \Rightarrow ABCD \Rightarrow aBCD \Rightarrow abCD \Rightarrow abcD \Rightarrow abcd$
- How long are the derivations?

Consequences of Chomsky Normal Form

- What do all trees look like?
  - Binary trees
- What do all derivations look like?
  - $S \Rightarrow AB \Rightarrow ABC \Rightarrow ABCD \Rightarrow aBCD \Rightarrow abCD \Rightarrow abcD \Rightarrow abcd$
- How long are the derivations?
  - $2n - 1$

Consequences of Chomsky Normal Form

- What do all trees look like?
  - Binary trees
- What do all derivations look like?
  - $S \Rightarrow AB \Rightarrow ABC \Rightarrow ABCD \Rightarrow aBCD \Rightarrow abCD \Rightarrow abcD \Rightarrow abcd$
- How long are the derivations?
  - $2n - 1$
- Any derivation of a string of length n must take $2n - 1$ steps.
- To find a parse we can search all derivations of this length.
- Guaranteed to say "yes" or "no" in a finite amount of time.

Greibach Normal Form

- All rules must be in one of these forms:

$$\begin{aligned} A &\rightarrow aB_1B_2 \ldots B_n \\ S &\rightarrow \epsilon \end{aligned}$$

- $B_1B_2 \ldots B_n$ is a (possibly empty) string of nonterminals.
- The last rule is needed only if the language contains ϵ.
- There can be no left recursion.
- What do the trees look like?
- How long are the derivations?

Converting to Greibach Normal Form

1. Add S_0, a new start symbol, and the rule $S_0 \rightarrow S$.
2. Eliminate **unit** rules, $A \rightarrow B$.
3. Remove left recursion.
4. Eliminate ϵ rules.
5. Make substitutions as needed.
   - Can be very difficult and explode to many rules.

Consequences of Greibach Normal Form

- All derivations of a string of length n have n steps.
- There are only finitely many derivations of length n or less.
- To find a parse we can exhaustively search all derivations of length n or less.
- There exists an **effective procedure** to find a parse for a CFL.

Pumping Lemma for CF Languages

- If a tree has maximum branching factor b and is of height h, what is the maximum number of leaves?

Pumping Lemma for CF Languages

- If a tree has maximum branching factor b and is of height h, what is the maximum number of leaves?

$$b^h$$

Pumping Lemma for CF Languages

- If a tree has maximum branching factor b and is of height h, what is the maximum number of leaves?
$$b^h$$
- If there are n symbols in a grammar, and the maximum length of a rule's RHS is b, and a parsed string is longer than b^n, what does that say about the height of the parse tree?

Pumping Lemma for CF Languages

- If a tree has maximum branching factor b and is of height h, what is the maximum number of leaves?

$$b^h$$

- If there are n symbols in a grammar, and the maximum length of a rule's RHS is b, and a parsed string is longer than b^n, what does that say about the height of the parse tree?

$$h > n$$

Pumping Lemma for CF Languages

- If a tree has maximum branching factor b and is of height h, what is the maximum number of leaves?
$$b^h$$
- If there are n symbols in a grammar, and the maximum length of a rule's RHS is b, and a parsed string is longer than b^n, what does that say about the height of the parse tree?
$$h > n$$
- What does the pigeonhole principle say about a tree where each node in the tree is labeled with a symbol from the grammar, but the height of the tree is greater than the number of symbols?

Pumping Lemma for CF Languages

- If a tree has maximum branching factor b and is of height h, what is the maximum number of leaves?

$$b^h$$

- If there are n symbols in a grammar, and the maximum length of a rule's RHS is b, and a parsed string is longer than b^n, what does that say about the height of the parse tree?

$$h > n$$

- What does the pigeonhole principle say about a tree where each node in the tree is labeled with a symbol from the grammar, but the height of the tree is greater than the number of symbols?

  There must be a path with a repeated symbol.

Pumping Lemma for CF Languages

$A_j = A_k$

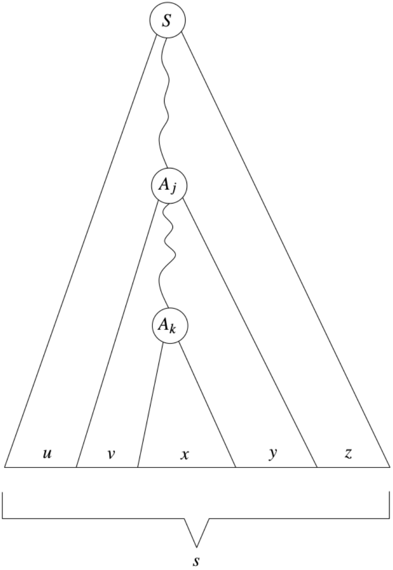
uxz

$uvxyz$

$uvvxyyz$

$uvvvxyyyz$

uv^ixy^iz



Pumping Lemma for CF Languages

- If a CFL is infinite, it must have recursion in it somewhere:

$$\begin{array}{rcl} S & \rightarrow & uNy \\ N & \rightarrow & vNx \mid w \end{array}$$

- This means derivations like this are possible:

$S \Rightarrow uNy \Rightarrow uvNxy \Rightarrow uvvNxxy \Rightarrow \ldots \Rightarrow uv^4Nx^4y \Rightarrow uv^4wx^4y$

Pumping Lemma for CF Languages

- If a CFL is infinite, it must have recursion in it somewhere:

$$\begin{array}{rcl} S & \rightarrow & uNy \\ N & \rightarrow & vNx \mid w \end{array}$$

- This means derivations like this are possible:

$$S \Rightarrow uNy \Rightarrow uvNxy \Rightarrow uvvNxxy \Rightarrow \ldots \Rightarrow uv^4Nx^4y \Rightarrow uv^4wx^4y$$

- Similar arguments can be made for indirect recursion.

Pumping Lemma for CF Languages

Theorem

If a language L is context-free, then there exists some $p \geq 1$ such that any string s in L with $|s| \geq p$ can be written as

$$s = abcde$$

such that

1. $|bd| \geq 1$
2. $|bcd| \leq p$
3. $ab^n cd^n e$ *is in L for all* $n \geq 0$

Proof that $L = 0^n 1^n 2^n$ is not CF

- Suppose L is CF, then p exists as in the theorem.
- $0^p 1^p 2^p \in L$ by definition.
- From theorem, $0^p 1^p 2^p = abcde$ and $ab^n cd^n e \in L$ for all n, but we don't know which parts are where.
- Case 1: b or d contains two different digits.
  - Then ab^2cd^2e must have digitss out of numerical order.
  - Then $ab^2cd^2e \notin L$, contradiction.
- Case 2: b and d each contain only one kind of digit.
  - Then ab^2cd^2e contains more of 1 or 2 kinds of digit.
  - Then there can't be the same number for all 3 kinds of digits.
  - Then $ab^2cd^2e \notin L$, contradiction.

Pumping Lemma exercises (some are hard)

Show that each of the following languages is not CF.

- 1^n where n is prime
- 1^m where $m = n^2$
- $0^\ell 1^m 2^n$ where $\ell < m < n$
- $0^n 1^n 2^i$ where $i \leq n$
- ww where $w \in (0+1)^*$
- $0^n 1^n 2^n$

CF Languages are closed under union, product, and closure

- Let S_1 and S_2 be the start symbols for L_1 and L_2.
- A grammar for $L_1 \cup L_2$ can be constructed starting with

$$S \to S_1 \mid S_2$$

- A grammar for $L_1 L_2$ can be constructed starting with

$$S \to S_1 S_2$$

- A grammar for L_1^* can be constructed starting with

$$S \to S_1 S \mid \epsilon$$

CF languages are NOT closed under complement

- $L = a^\ell b^m c^n$ where either $\ell \neq m$ or $m \neq n$ is CF
  - (exercise)
- The complement of L is $a^n b^n c^n$, which is not CF
  - (see previous)

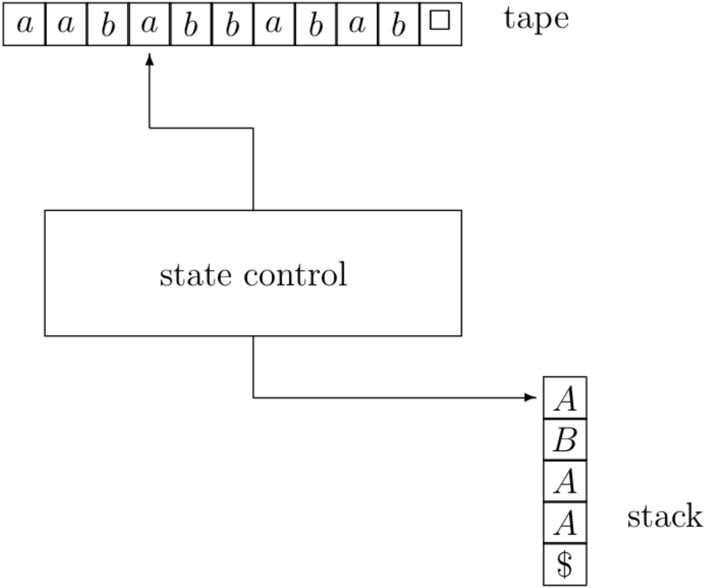
CF languages are NOT closed under intersection

- $L_1 = a^m b^m c^n$ is CF
  - (exercise)
- $L_2 = a^m b^n c^n$ is CF
  - (exercise)
- $L_1 \cap L_2 = a^n b^n c^n$, which is not CF
  - (see previous)

The intersection of a CF language and a regular language is CF

- Given a PDA for one and a DFSA for the other:
- Create a new PDA with states that are the cross product of the states of the two machines.
- As input is processed, run both machines in parallel.
- Accept if both accept.

Pushdown automata



Tape drives

Deterministic pushdown automata

Definition 3.5.1 A *deterministic pushdown automaton* is a 5-tuple $M = (\Sigma, \Gamma, Q, \delta, q)$ where

1. Σ is a finite set, called the *tape alphabet*; the blank symbol $\Box$ is not in Σ.
2. Γ is a finite set, called the *stack alphabet*; this alphabet contains the special symbol \$.
3. Q is a finite set, whose elements are called *states*.
4. q is an element of Q, called the *start state*.
5. δ is the *transition function*, which is a function

$$\delta : Q \times (\Sigma \cup \{\Box\}) \times \Gamma \to Q \times \{N, R\} \times \Gamma^*$$

Given a state, a tape letter, and the top letter on the stack, (r, a, A), δ determines the next state, whether or not to move the tape head, and the replacement for A on the stack:

$$\delta(r, a, A) = (r', \sigma, w)$$

Computation and Termination

Start: The automaton is in start state q, the tape head is at the leftmost symbol of the input string $a_1 a_2 ... a_n$, and the stack contains only the special symbol \$.

Computation: The state, input and the stack are transformed, step by step, by following the transition function δ.

Termination: The PDA *terminates* when the stack becomes empty.

Acceptance: The PDA *accepts* $a_1 a_2 ... a_n$ if

1. the automaton terminates on this input, and
2. at the time of termination, the tape head is immediately to the right of a_n.

Properly nested parentheses

$qa\$ \to qR\$S$	because of the a, S is pushed onto the stack
$qaS \to qRSS$	because of the a, S is pushed onto the stack
$qbS \to qR\epsilon$	because of the b, the top element is popped from the stack
$qb\$ \to qN\epsilon$	the number of bs read is larger than the number of as read; the stack is made empty (hence, the computation terminates before the entire string has been read), and the input string is rejected
$q\square\$ \to qN\epsilon$	the entire input string has been read; the stack is made empty, and the input string is accepted
$q\square S \to qNS$	the entire input string has been read, it contains more as than bs; no changes are made (thus, the automaton does not terminate), and the input string is rejected

0^n1^n

$q_0 0\$ \to q_0 R\$S$	push S onto the stack
$q_0 0S \to q_0 RSS$	push S onto the stack
$q_0 1\$ \to q_0 N\$$	first symbol in the input is 1; loop forever
$q_0 1S \to q_1 R\epsilon$	first 1 is encountered
$q_0 \Box\$ \to q_0 N\epsilon$	input string is empty; accept
$q_0 \Box S \to q_0 NS$	input only consists of 0s; loop forever
$q_1 0\$ \to q_1 N\$$	0 to the right of 1; loop forever
$q_1 0S \to q_1 NS$	0 to the right of 1; loop forever
$q_1 1\$ \to q_1 N\$$	too many 1s; loop forever
$q_1 1S \to q_1 R\epsilon$	pop top symbol from the stack
$q_1 \Box\$ \to q_1 N\epsilon$	accept
$q_1 \Box S \to q_1 NS$	too many 0s; loop forever

b in the middle

$qa\$ \to qR\$S$	push S onto the stack
$qaS \to qRSS$	push S onto the stack
$qb\$ \to q'R\$$	reached the middle
$qb\$ \to qR\$S$	did not reach the middle; push S onto the stack
$qbS \to q'RS$	reached the middle
$qbS \to qRSS$	did not reach the middle; push S onto the stack
$q\Box\$ \to qN\$$	loop forever
$q\Box S \to qNS$	loop forever
$q'a\$ \to q'N\epsilon$	stack is empty; terminate, but reject, because the entire input string has not been read
$q'aS \to q'R\epsilon$	pop top symbol from stack
$q'b\$ \to q'N\epsilon$	stack is empty; terminate, but reject, because the entire input string has not been read
$q'bS \to q'R\epsilon$	pop top symbol from stack
$q'\Box\$ \to q'N\epsilon$	accept
$q'\Box S \to q'NS$	loop forever