Definition 2.8: A CFG is in Chomsky normal form if every rule is of the form $A \rightarrow BC$ or $A \rightarrow a$. Let $S$ be the start symbol.

- Note that our previous definition did not have $\varepsilon$ but did allow the right-hand side to have any number of terminals and variables (including zero of them).
- CNF is a more restrictive grammar which will make some proofs easier.

Note that our previous definition did not have $\varepsilon$ but did allow the right-hand side to have any number of terminals and variables (including zero of them).

- Note that our previous definition did not have $\varepsilon$ but did allow the right-hand side to have any number of terminals and variables (including zero of them).

Chomsky Normal Form

- Only have to worry about binary branching
- Only have to worry about $\varepsilon$ on start variable

Definition 2.8: A CFG is in Chomsky normal form if every rule is of the form $A \rightarrow a$ or $A \rightarrow BC$. Let $S$ be the start symbol. $S$ can only appear on the left-hand side, we allow the rule $S \rightarrow \varepsilon$.

Overview

- Equivalence
- Definition
- Pushdown Automata
- CNF
Proof

Step 1: add a new start variable $S_o$ and the rule $S_o \rightarrow S$

Start state is just on left-hand-side

Step 2: ǫ productions (other than $S_o \rightarrow ǫ$

- Add extra versions of rules where $A$ is on right-hand-side, without the ǫ
- Remove rules $A \rightarrow ǫ$ where $A$ is not $S_o$

Example: $R \rightarrow uAv$ (u a string of terminals and variables without ǫ)

Example: $R \rightarrow uAvAw$

Example: $R \rightarrow A$ ($R$ is not start symbol)

Other rules $A \rightarrow aB$, $B \rightarrow CD$ 

Unit rules $A \rightarrow B$

ǫ productions on rules other than the start symbol

Start state just on left-hand-side

What are the problems with the CFG?

Theorem 2.9: Any CFL is generated by a CFG in Chomsky normal form.

Equivalence of CFL and CFGs
Step 1: Add new start state: $T$

$\phi \leftarrow B$

$B \leftarrow A$

$A \leftarrow B$ | $\epsilon$

Example 2.10

Proof (continued)

• Step 3: Unit rules:
  - Replace each rule $A \rightarrow B$ where $B$ is a new variable, say $U_B$, and add rule $U_B \rightarrow B$
  - Replace each rule where $B$ is a new variable $U_B$ and $B \rightarrow \nu$, where $\nu$ is a terminal, with binary rules $U_B \rightarrow \nu, U_B \rightarrow \nu$

Example:

$A \rightarrow B$

$B \rightarrow D$

$C \rightarrow A$

$B \rightarrow F$

• Step 4: Other rules:
  - $A \rightarrow \nu_1 \nu_2 \ldots \nu_k$ where $k \geq 3$ and each $\nu_i$ is a variable or terminal
    - Replace with binary rules $A \rightarrow \nu_1 A_1, A_1 \rightarrow \nu_2 A_2, \ldots, A_k \rightarrow \nu_{k-1} \nu_k$
    - Where $A_i$'s are new variables
  - $A \rightarrow \nu_1 \nu_2$ where $\nu_1$ or $\nu_2$ is a terminal
    - Replace terminal $\nu_i$ with a new variable, say $U_i$, and add rule $U_i \rightarrow \nu_i$

Example:

$B \leftarrow A$

$A \leftarrow D$

$B \leftarrow E$

Step 3: Unit Rules:

Proof (continued)
Step 3: Remove unit rules

- If $S$ goes to something, add a rule with $L$ goes to it.
  
  $S \rightarrow L$  •

- If $S$ goes to something, add a rule with $V$ goes to it.
  
  $S \rightarrow V$  •

- If $B$ goes to something, add a rule with $V$ goes to it.
  
  $B \rightarrow V$  •

- Anytime that $S$ goes to $L$, $S$ already goes to it, so just remove it.
  
  Remove unit rule. $S \rightarrow S$  •

Step 2: Remove epsilon rules

- Get rid of $A \rightarrow \epsilon$  •

- Get rid of $B \rightarrow \epsilon$  •
Step 4: Rules with 2 on right side & terminals
- Remove $S \rightarrow aB$
- Remove $T \rightarrow aB$

Step 4: Rules with more than 2 on right side
- Replace $S \rightarrow ASA$
- Replace $T \rightarrow ASA$
Pushdown Automata

- Can push onto the top of the stack.
- Transitions to another state (and read it)
- Can pop off the top of the stack
- Has an input head, which it can read
- Decides what to do based on:
  - Function in the control
  - Transitions beyond the amount
  - Provides extra memory
  - Excess of the stack as well

Overview

- Equivalence
- Definition
- Pushdown Automata $\subseteq$ CNF
Non-determinism

For Finite Automata

- NFA and DFA s have same power: they recognize the same classes of languages.
- $\text{NFA with push-down automata}$
- $\text{DFA with push-down automata}$

We will focus on non-deterministic push-down automata, which are equivalent to CFGs.

Example:

- Pushdown automata can read the 0's and push them onto the stack.
- Can pop off the 0's from the stack as it reads the 1's.

$\text{Example: } 0^n1^n$
Formal Definition

A pushdown automaton is a 6-tuple \((Q, \Sigma, \Gamma, \delta, q_0, F)\) where:

1. \(Q\) is the set of states
2. \(\Sigma\) is the input alphabet
3. \(\Gamma\) is the stack alphabet
4. \(\delta: Q \times \Sigma \epsilon \times \Gamma \epsilon \rightarrow P(Q \times \Gamma \epsilon)\) is the transition function
5. \(q_0 \in Q\) is the start state
6. \(F \subseteq Q\) is the set of accept states

Overview

- CNF
- Pushdown Automata

Equivalence
- Definition
- Pushdown Automata

CNF
A pushdown automaton $M = (Q, \Sigma, \Gamma, \delta, q_0, F)$ accepts $w$ if $w$ can be written as $w = w_1, w_2, ... w_m$, where each $w_i \in \Sigma \epsilon$ and there is a sequence of states $r_0, r_1, ..., r_m \in Q$ and a sequence of stack contents $s_0, s_1, ..., s_m \in \Gamma^\ast$ such that:

1. $r_0 = q_0$ (M starts in start state)
2. $s_0 = \epsilon$ (M starts with an empty stack)
3. For $i = 0, ..., m-1$, we have $(r_i, w_i+1, a) \in \delta(r_i, w_i, a)$, where $s_i = at$ and $s_{i+1} = bt$, for some $a, b \in \Gamma \epsilon$ and $t \in \Gamma^\ast$.
4. $r_m \in F$ - Stack formalization is a bit tricky

Transition Function

$\delta: Q \times (\Sigma \epsilon \times \Gamma^\ast) \rightarrow P(Q \times \Gamma^\ast)$

- Transition function decides what to do based on inputs
  - What state it is in
  - Can look at the input (and consume it)
  - Can look at the stack (in which case it pops if)
  - Can push onto stack or not

Transition Function outcomes:

- New state
- Specified by reading a on the stack
- Can ignore the stack (not pop anything off)
- Does this by specifying what to do on a certain stack value
- Can look at the stack (in which case it pops if)

$\delta(q, \epsilon, a) = \delta(q, \epsilon, b) 
\delta(q, \epsilon, t) = \delta(q, \epsilon, t)$

Transition Function decides what to do based on (inputs)
Example

• Give formal definition

• Make sure that it ends properly

• Give state diagram (with just two states). Don’t worry about how it ends.

• What is PDA for $0^n1^n$?

PDA

Does it know when it is at the end of the input?

• Does it know when the stack is empty?

• Can it pop on an empty stack?

• Can it not look at the stack at all during a transition?

• Can it pop and push in the same transition?

• Must stack be empty at accept?

• How is the nondeterminism captured?
Example

\[ \{0, 1\} = \exists \gamma \ni \alpha \]

Give state diagram

- Formally, the definition of PDA does not allow it to check for empty stack or end of input.
  - Add extra states to test for empty stack at end.
  - Have an initial state push `$` on stack.
  - Can add `$` to stack alphabet.

Running out of Stack or Input

- Not needed as accept state only takes effect when machine is at end of input.
Theorem: A language is context-free if and only if some pushdown automaton recognizes it.

Equivalence with CFG
Example

Give a derivation of $aaccbb$ with

$S \to aSb \mid c \mid \varepsilon$

Trace out stack operations

Example

Let $A$ be a CFL, so there is a CFG $G$ that generates $L$.

Lemma: If language is context-free, some PDA recognizes it.

Context-Free Grammar $\Rightarrow$ PDA

Lemma: If language is context-free, some PDA recognizes it.

Construct PDA $P$ that simulates doing a derivation in $G$.

Top of stack

$S \rightarrow aSb|c|\varepsilon$

Top of stack, and just transition to accept state if $S$ is on

So we can tell when we are at end of derivation, start by pushing a $\$$ on top of the stack and just transition to accept state if $\$$ is on top of the stack.
Rules

- Final transition returns to the main looping state
- Subsequent transitions to the subsequent states push \( y \) and then push \( x \)
- If the top of the stack is \( A \) (and pops it) and \( A \) is on the right-hand side, non-deterministically choose a rule with \( A \) on the left-hand side, and push the right-hand side onto the stack
- For each rule \( A \rightarrow xy \) where \( x, y \) are terminals or symbols
- And push the right-hand side onto the stack
- If the top of the stack is a variable, \( A \), pop it, and
- If the top of the stack is a terminal, pop it, and read that character (if they don't match, then this branch of nondeterminism dies)
- If the top of the stack is a terminal, \( \$ \), and read that character from input (if it matches \( \$ \), on left-hand side, and push the left-hand side onto the stack
- Repeat the following forever
  - Place a non-variable on top of stack
  - Place \( \$ \) on top of stack

So far

- Make a PDA that does the following for CFG G
- Place \( \$ \) on top of stack