Overview

⇒ CNF
• Pushdown Automata
• Definition
• Equivalence

Chomsky Normal Form

**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. $S$ can only appear on the left-hand side, we allow the rule $S \rightarrow \epsilon$.

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

- CNF is a more restrictive than CFG, which will make some proofs easier
  + Only have to worry about $\epsilon$ on start variable
  + Only have to worry about binary branching
Equivalence of CFL and CFGs

Theorem 2.9: A CFL can be generated by CFG in Chomsky Normal Form

• What are the problems with the CFG?
  - Start state just on left-hand-side
  - $\epsilon$ productions on rules other than the start symbol
  - Unit rules: $A \rightarrow B$
  - Other rules $A \rightarrow aB, A \rightarrow BCDE$

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: $\epsilon$ productions (other than $S_o \rightarrow \epsilon$)
  - Remove rules $A \rightarrow \epsilon$, where $A$ is not $S_o$
  - Add extra versions of rules where $A$ is on right-hand-side, without the $A$
    + Example: $R \rightarrow uAv$ (u a string of terminals and variables without $A$)
    + Example: $R \rightarrow uAvAw$
    + Example: $R \rightarrow A$ ($R$ is not start symbol)
Proof (continued)

- **Step 3: Unit rules:**
  - Remove each rule $A \rightarrow B$
  - For each rule $B \rightarrow u$ (where $u$ is a string of terminals and variables)
    add $A \rightarrow u$, unless this is a unit rule that was already removed
  + Example: $A \rightarrow B$
    $B \rightarrow DE$
    $C \rightarrow AD$
    $B \rightarrow F$

- **Step 4: Other rules**
  - $A \rightarrow u_1u_2...u_k$ where $k \geq 3$ where each $u_i$ is a variable or terminal
    + Replace with binary rules $A \rightarrow u_1A_1$, $A_1 \rightarrow u_2A_2$, ..., $A_{k-1} \rightarrow u_{k-1}u_k$
    + where $A_i$’s are new variables
  - $A \rightarrow u_1u_2$ where $u_1$ or $u_2$ is a terminal
    + Replace terminal $u_i$ with a new variable, say $U_i$, and add rule $U_i \rightarrow u_i$

---

**Example 2.10**

$$S \rightarrow ASA|aB$$
$$A \rightarrow B$$
$$B \rightarrow b|\epsilon$$

Step 1: add new start state: $T$
Step 2: Remove epsilon rules

• Get rid of $B \rightarrow \epsilon$

• Get rid of $A \rightarrow \epsilon$

Step 3: Remove unit rules

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

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

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

• Remove $T \rightarrow S$
  - If $S$ goes to something, add a rule with $T$ goes to it.
Step 4: Rules with more than 2 on right side

- Replace $T \to ASA$

- Replace $S \to ASA$

Step 4: Rules with 2 on right side & terminals

- Remove $T \to aB$

- Remove $S \to aB$
Overview

- CNF
  \[\Rightarrow\] Pushdown Automata
- Definition
- Equivalence

Pushdown Automata

- Like a NFA, but has a stack as well
  - Provides extra memory beyond finite amount in the control
- Function
  - Decides what to do based on:
    - Has an input head, which it can read the next character
    - Can pop off the top of the stack (and read it)
    - Transitions to another state
    - Can push onto the top of the stack
Non-determinism

- For Finite Automata
  - NFA and DFA's have same power; they recognize the same classes of languages.
- Not so with Push-down automata.

Example: $0^n 1^n$

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

- CNF
- Pushdown Automata
  - Definition
- Equivalence

Formal Definition

A **pushdown automaton** is a 6-tuple \((Q, \Sigma, \Gamma, \delta, q_0, F)\) where \(Q, \Sigma, \Gamma,\) and \(F\) are all finite sets, and

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

\[ \delta : Q \times \Sigma \epsilon \times \Gamma \epsilon \rightarrow \mathcal{P}(Q \times \Gamma \epsilon) \]

- Transition function decides what to do based on inputs
  - What state it is in
  - Can look at the input (and consume it) or not (\( \epsilon \))
  - Can look at the stack (in which case it pops it)
    + Does this by specifying what to do on a certain stack value
  - Can ignore the stack (not pop anything off)
    + Specified by reading \( \epsilon \) on the stack

- Transition function outcomes:
  - New state
  - Can push onto stack or not (\( \epsilon \))

Computation

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 \( s_0, s_1, ..., s_m \in \Gamma^* \) 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 \( (r_{i+1}, b) \in \delta(r_i, w_{i+1}, a) \), where \( s_i = at \) and \( s_{i+1} = bt \), for some \( a, b \in \Gamma \epsilon \) and \( t \in \Gamma^* \).
4. \( r_m \in F \)

- Stack formalization is a bit tricky
  + \( a \) is what gets popped off the stack with this transition
  + \( b \) is what gets pushed on the stack with this transition
  + \( t \) is part of the stack that won’t change with the transition
  + Transition doesn’t care what \( t \) is, as long as it doesn’t change
PDA

- How is the nondeterminism captured?
- Must stack be empty at accept?
- Can it pop and push in the same transition?
- Can it not look at the stack at all during a transition?
- Can it pop on an empty stack?
- Does it know when the stack is empty?
- Does it know when it is at the end of the input?

Example

- What is PDA for $0^n1^n$
  - Give state diagram (with just two states). Don’t worry about how it ends.
  - Make sure that it ends properly

- Give formal definition
Running out of Stack or Input

- Formal definition of PDA does not have a mechanism to allow a PDA to test for an empty stack
  - Can add ‘$’ to stack alphabet
  - Have an initial state push this on stack
  - Add extra states to test for this
- Formal definition of PDA does not allow it to check for end of input
  - Not needed as accept state only takes effect when machine is at end of input

Example $w^R w$ where $\Sigma = \{0, 1\}$

- Give state diagram
Overview

- CNF
- Pushdown Automata
- Definition
  ⇒ Equivalence

Equivalence with CFG

**Theorem:** A language is context free if and only if some pushdown automaton recognizes it.
Example

- Give a derivation of \( aacbb \) with
  \[ S \rightarrow aSb | c | \epsilon \]
- Trace out stack operations
So far

- Make a PDA that does the following for CFG $G$
  - Place ‘$’ on top of stack
  - Place start variable on top of stack
  - Repeat the following forever
    + If top of the stack is a variable $A$, pop it, and non-deterministically choose a rule with $A$ on left-hand side, and push the right-hand side onto the stack
    + If top of the stack is a terminal, pop it, and read that character from input (if they don’t match, then this branch of nondeterminism dies)
    + If the top of stack is a ‘$’, enter the accept state (doing so accepts the input if it is entirely read in at that point; otherwise this branch will die)

Rules

- If top of the stack is a variable $A$, pop it, and non-deterministically choose a rule with $A$ on left-hand side, and push the right-hand side onto the stack
- For each rule, $A \rightarrow xyz$ where $x, y, z$ are terminals or symbols
  - Have a sequence of states in which the first state is transitioned to if the top of the stack is $A$ (and pops it) and pushes $z$ onto the stack
    + Subsequence transitions to the subsequent states push $y$ and then push $x$
    + Final transition returns to the main looping state