Push-Down Automata (PDA)

- Finite automata where the machine interpretation of regular languages.
- **Push-Down Automaton** are the machine interpretation for grammars.
- The problem of finite automata was that they couldn't handle languages that needed some sort of unbounded memory... something that could be implemented easily by a single (unbounded) integer **register**!
- Example: To recognize the language $L = \{0^n 1^n \mid n \geq 0\}$, all you need is to count how many 0's you have seen so far...
- Push-Down Automata allow even more than a register: a full **stack**!

Recursive Algorithms and Stacks

- A stack allows the following basic operations
  - **Push**, pushing a new element on the top of the stack.
  - **Pop**, removing the top element from the stack (if there is one).
  - **Peek**, checking the top element without removing it.

- General Principle in Programming:
  Any recursive algorithm can be turned into a non-recursive one using a stack and a while-loop which exits only when stack is empty.

Recursive Algorithms and Stacks

- A stack allows the following basic operations
  - **Push**, pushing a new element on the top of the stack.
  - **Pop**, removing the top element from the stack (if there is one).
  - **Peek**, checking the top element without removing it.

- General Principle in Programming:
  Any recursive algorithm can be turned into a non-recursive one using a stack and a while-loop which exits only when stack is empty.

- It seems that with a stack at our fingertips we can even recognize **palindromes**! Yoo-hoo!
  - Palindromes are generated by the grammar $S \to \varepsilon \mid aSa \mid bSb$.
  - Let's simplify for the moment and look at $S \to \# \mid aSa \mid bSb$.

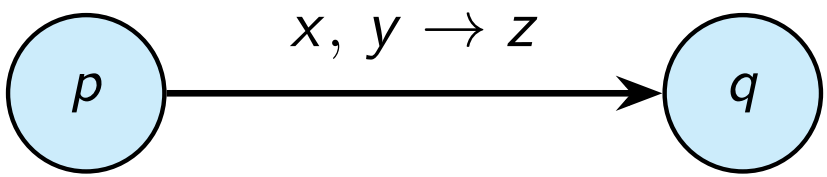
From CFG's to Stack Machines

- The CFG $S \rightarrow \# \mid aSa \mid bSb$ describes palindromes containing exactly 1 $\#$.
- Question: Using a stack, how can we recognize such strings?

PDA's à la Sipser

- To aid analysis, theoretical stack machines restrict the allowable operations. Each text-book author has his/her own version.
- Sipser's machines are especially simple:
  - Push/Pop rolled into a single operation: **replace top stack symbol**.
  - In particular, replacing top by ε is a pop.
- No intrinsic way to test for empty stack.
  - Instead often push a special symbol ("$") as the very first operation!
- Epsilon's used to increase functionality
  - result in default **nondeterministic** machines.

Sipser's PDA Version



If at state p **and** next input is x **and** top stack is y,
then go to state q **and** replace y by z on stack.

- $x = \varepsilon$: ignore input, don't read
- $y = \varepsilon$: ignore top of stack and push z
- $z = \varepsilon$: pop y

In addition, push "$" initially to detect the empty stack.

PDA: Formal Definition

- Definition: A **pushdown automaton** (PDA) is a 6-tuple $M = (Q, \Sigma, \Gamma, \delta, q_0, F)$:
  - Q, Σ, and q_0, and F are defined as for an FA.
  - Γ is the **stack alphabet**.
  - δ is as follows:
    Given a state p, an input symbol x and a stack symbol y, $\delta(p, x, y)$ returns all (q, z) where q is a target state and z a stack replacement for y.

$$\delta\colon Q \times \Sigma_\varepsilon \times \Gamma_\varepsilon \to P(Q \times \Gamma_\varepsilon)$$

PDA Exercises

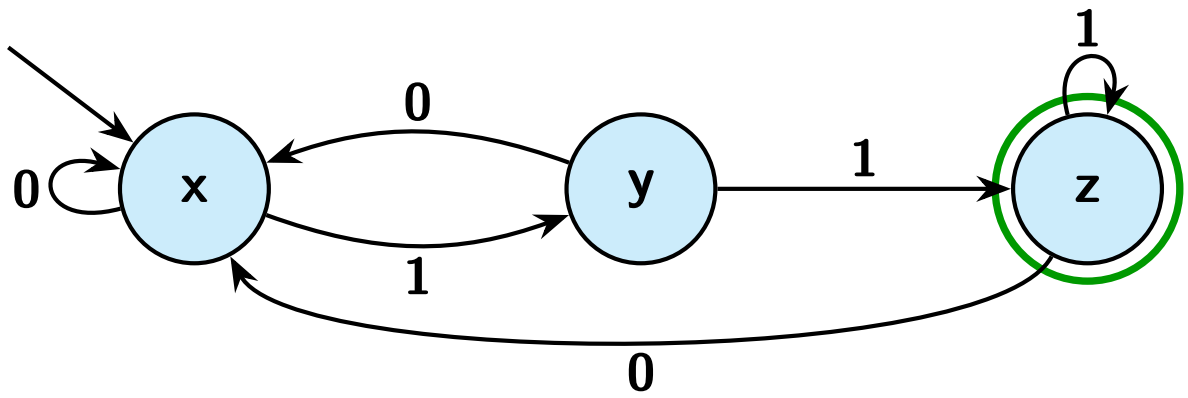
- Draw the PDA $\{a^i b^j c^k \mid i, j, k \geq 0 \text{ and } i = j \text{ or } i = k\}$

- Draw the PDA for $L = \{ x \in \{a, e\}^* \mid n_a(x) = 2n_e(x) \}$

Model Robustness

- The class of regular languages was quite **robust**
  - Allows multiple ways for defining languages (automaton vs. regexp)
  - Slight perturbations of model do not change result (non-determinism)
- The class of context free languages is also robust:
  you can use either PDA's or CFG's to describe the languages.
- However, it is less robust than regular languages when it comes to slight perturbations of the model:
  - Smaller classes
    - Right-linear grammars
    - Deterministic PDA's
  - Larger classes
    - Context Sensitive Grammars

Right Linear Grammars vs. Regular Languages



- The DFA above can be simulated by the grammar
  - $x \rightarrow 0x \mid 1y$
  - $y \rightarrow 0x \mid 1z$
  - $z \rightarrow 0x \mid 1z \mid \varepsilon$

- Definition: A **right-linear grammar** is a CFG such that every production is of the form $A \rightarrow uB$, or $A \rightarrow u$ where u is a terminal string, and A, B are variables.

Right Linear Grammars vs. Regular Languages

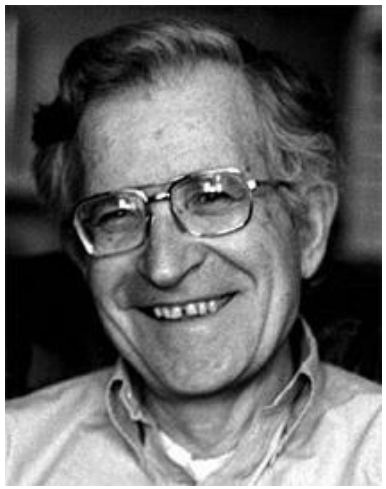
- Theorem: If $M = (Q, \Sigma, \delta, q_0, F)$ is an NFA then there is a right-linear grammar $G(M)$ which generates the same language as M.
- *Proof*:
  - Variables are the states: $V = Q$
  - Start symbol is start state: $S = q_0$
  - Same alphabet of terminals Σ
  - A transition $q_1 \xrightarrow{a} q_2$ becomes the production $q_1 \rightarrow aq_2$
  - For each transition, $q_1 \rightarrow aq_2$ where q_2 is an accept state, add $q_1 \rightarrow a$ to the grammar
- Homework: Show that the reverse holds. Right-linear grammar can be converted to a FSA. This implies that RL $\approx$ Right-linear CFL.

Right Linear Grammars vs. Regular Languages

- Theorem: If $M = (Q, \Sigma, \delta, q_0, F)$ is an NFA then there is a right-linear grammar $G(M)$ which generates the same language as M.
- *Proof*:
  - Variables are the states: $V = Q$
  - Start symbol is start state: $S = q_0$
  - Same alphabet of terminals Σ
  - A transition $q_1 \xrightarrow{a} q_2$ becomes the production $q_1 \rightarrow aq_2$
  - For each transition, $q_1 \rightarrow aq_2$ where q_2 is an accept state, add $q_1 \rightarrow a$ to the grammar
- Homework: Show that the reverse holds. Right-linear grammar can be converted to a FSA. This implies that RL $\approx$ Right-linear CFL.
- **Question: Can every CFG be converted into a right-linear grammar?**

Chomsky Normal Form

- Chomsky came up with an especially simple type of context free grammars which is able to capture all context free languages, the **Chomsky normal form** (CNF).
- Chomsky's grammatical form is particularly useful when one wants to prove certain facts about context free languages. This is because assuming a much more restrictive kind of grammar can often make it easier to prove that the generated language has whatever property you are interested in.
- Noam Chomsky, linguist at MIT, creator of the Chomsky hierarchy, a classification of formal languages. Chomsky is also widely known for his left-wing political views and his criticism of the foreign policy of U.S. government.



Chomsky Normal Form

- Definition: A CFG is said to be in **Chomsky Normal Form** if every rule in the grammar has one of the following forms:
  - $S \to \varepsilon$ (ε for epsilon's sake only)
  - $A \to BC$ (dyadic variable productions)
  - $A \to a$ (unit terminal productions)

  where S is the start variable, A, B, C are variables and a is a terminal.

- Thus epsilons may only appear on the right hand side of the start symbol and other rights are either 2 variables or a single terminal.

CFG $\rightarrow$ CNF

- Converting a general grammar into Chomsky Normal Form works in four steps:
  1. Ensure that the **start** variable doesn't appear on the right hand side of any rule.
  2. Remove all **epsilon** productions, except from start variable.
  3. Remove unit variable productions of the form $A \rightarrow B$ where A and B are variables.
  4. Add variables and dyadic variable rules to replace any **longer** non-dyadic or non-variable productions

CFG $\rightarrow$ CNF: Example

$$S \rightarrow \varepsilon \mid a \mid b \mid aSa \mid bSb$$

1. No start variable on right hand side

$$S' \rightarrow S$$
$$S \rightarrow \varepsilon \mid a \mid b \mid aSa \mid bSb$$

2. Only start state is allowed to have ε

$$S' \rightarrow S \mid \varepsilon$$
$$S \rightarrow \not{\varepsilon} \mid a \mid b \mid aSa \mid bSb \mid aa \mid bb$$

3. Remove unit variable productions of the form $A \rightarrow B$.

$$S' \rightarrow \not{S} \mid \varepsilon \mid a \mid b \mid aSa \mid bSb \mid aa \mid bb$$
$$S \rightarrow a \mid b \mid aSa \mid bSb \mid aa \mid bb$$

CFG $\rightarrow$ CNF: Example continued

$$
\begin{aligned}
S' &\rightarrow \cancel{S} \mid \varepsilon \mid a \mid b \mid aSa \mid bSb \mid aa \mid bb \\
S &\rightarrow a \mid b \mid aSa \mid bSb \mid aa \mid bb
\end{aligned}
$$

4. Add variables and dyadic variable rules to replace any longer productions.

$$
\begin{aligned}
S' &\rightarrow \varepsilon \mid a \mid b \mid \cancel{aSa} \mid \cancel{bSb} \mid \cancel{aa} \mid \cancel{bb} \mid AB \mid CD \mid AA \mid CC \\
S &\rightarrow a \mid b \mid \cancel{aSa} \mid \cancel{bSb} \mid \cancel{aa} \mid \cancel{bb} \mid AB \mid CD \mid AA \mid CC \\
A &\rightarrow a \\
B &\rightarrow SA \\
C &\rightarrow b \\
D &\rightarrow SC
\end{aligned}
$$

Why a Normal Form?

- It restricts the *shape* of the rules, not the *languages*: every CFG can be rewritten in CNF.
- Parse trees become *binary*: a string of length n is derived in exactly $2n - 1$ steps, and every rule $A \to BC$ splits a substring into *two* parts.
  Algorithms exploit this fixed shape: the **CYK algorithm** decides "is $w \in L(G)$?" in time $O(n^3)$ by dynamic programming over substrings.
- No ε-rules, no unit rules $A \to B$: every step makes *progress* (the form gets longer, or a variable becomes a terminal).
- Proofs get simpler, e.g. the pumping lemma for CFL's (later in this chapter).

CFG $\leftrightarrow$ PDA



- For **regular** languages we saw three equivalent descriptions: DFA, NFA, and REX.
- Context-free languages also come with two descriptions: a **generator** (the CFG) and a **machine** (the PDA).
- **Program:** show that these two are equivalent as well:
  - CFG $\rightarrow$ PDA (constructive)
  - PDA $\rightarrow$ CFG (we will only state it)

CFG $\rightarrow$ PDA

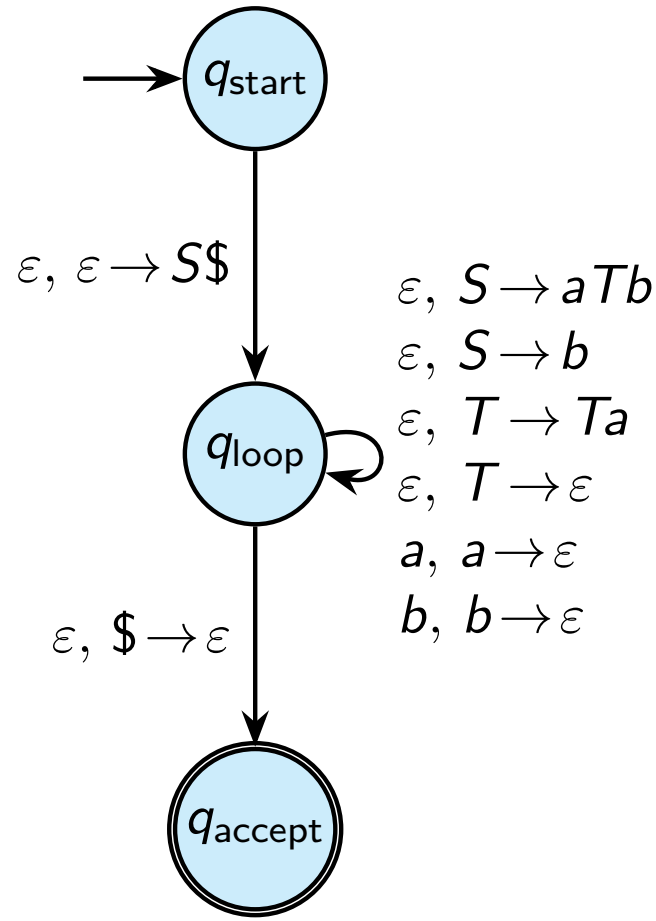
- CFG's can be converted into PDA's.
- In "NFA $\rightarrow$ REX" it was useful to consider GNFA's as a middle stage. Similarly, it's useful to consider **Generalized PDA's** here.
- A Generalized PDA (GPDA) is like a PDA, except it allows the top stack symbol to be replaced by a **whole string**, not just a single character or the empty string. It is easy to convert a GPDA's back to PDA's by changing each compound push into a sequence of simple pushes.

CFG $\rightarrow$ GPDA Recipe

1. Push the marker symbol \$ and the start symbol S on the stack.
2. Repeat the following steps forever
   a. If the top of the stack is the **variable** symbol A, nondeterministically select a rule of A, and substitute A by the string on the right-hand-side of the rule.
   b. If the top of the stack is a **terminal** symbol a, then read the next symbol from the input and compare it to a. If they match, continue. If they do not match reject this branch of the execution.
   c. If the top of the stack is the symbol **\$**, enter the accept state. (Note that if the input was not yet empty, the PDA will still reject this branch of the execution.)

CFG $\rightarrow$ GPDA $\rightarrow$ PDA: Example

- $S \rightarrow aTb \mid b$
- $T \rightarrow Ta \mid \varepsilon$



CFG $\rightarrow$ PDA: Now you try!

- Convert the grammar $S \rightarrow \varepsilon \mid a \mid b \mid aSa \mid bSb$

PDA $\rightarrow$ CFG

- To convert PDA's to CFG's we'll need to simulate the stack inside the productions.
- Unfortunately, in contrast to our previous transitions, this is not quite as constructive. We will therefore only state the theorem.
- Theorem: For each push-down automation there is a context-free grammar which accepts the same language.
- Corollary: **PDA** $\approx$ **CFG**.

Are all languages context-free?

- Design a CFG (or PDA) for the following languages:
- $L = \{ w \in \{0,1,2\}^* \mid \text{there are } k \text{ 0's, } k \text{ 1's, and } k \text{ 2's for } k \geq 0 \}$
- $L = \{ w \in \{0,1,2\}^* \mid \text{with } |0| = |1| \text{ or } |0| = |2| \text{ or } |1| = |2| \}$
- $L = \{ 0^k 1^k 2^k \mid k \geq 0 \}$