Regular Languages = Finite Automata
Context-Free Languages = ??? Automata

Regular languages (corresponding, as we saw, to left-linear or right-linear grammars) were recognized by Finite Automata, deterministic and non-deterministic. In fact, the two classes of automata were proven equivalent.

Question: is there a class of automata that recognizes exactly those languages generated by Context-Free grammars?

Push-Down Automaton: this is a non-deterministic finite automaton to which is added a stack, and the capacity to manipulate the top of the stack (push and pop symbols).

**Def.:** a Push-Down Automaton is a six-tuple (sextuple)

\[ M = (Q, \Sigma, \Gamma, \delta, s, F) \]

where

1. \( Q \) is a set of states
2. \( \Sigma \) is the alphabet for the strings of the language
3. \( \Gamma \) is the alphabet of stack symbols
4. \( \delta : Q \times (\Sigma \cup \{\varepsilon\}) \times (\Gamma \cup \{\varepsilon\}) \rightarrow 2^Q \times 2^{\Gamma \cup \{\varepsilon\}} \)
5. \( s \) is the start state
6. \( F \) is the set of final (or accepting) states.

Note that, for each triple (state, input symbol, top-of-stack symbol), \( \delta \) produces a set of pairs : \( \{(\text{state, top-of-stack symbol})\} \). For each pair, the new tos symbol replaces the old one. The presence of \( \varepsilon \) for a tos symbol in both input to and output from \( \delta \) simply means that a transition need not consume (pop) the tos nor write (push) a new tos.
since, besides consuming characters, we must include stack
the characters consumed in the transition. In this case we
A useful notation for
that starts with
Def: configuration and
C
The language recognized is
undo the stack: on seeing an
The moment it sees a
Ex. 3.28
\( M = (\{ q, p \}, \{ a, b, c \}, \{ a, b \}, \delta, q, (p) ) \), where
\( \delta(q, a, \epsilon) = (q, a) \); \( \delta(p, a, a) = (p, \epsilon) \); \( \delta(q, b, \epsilon) = (q, b) \);
\( \delta(p, b, b) = (p, \epsilon) \); \( \delta(q, c, \epsilon) = (p, \epsilon) \).
Determine what \( L(M) \) is.
Soln.: as long as \( M \) keeps reading \( a \) or \( b \) (having started in state \( q \)), it
keeps pushing the input symbols on the stack, remaining in state \( p \).
The moment it sees a \( c \), it moves to state \( p \) and expects to be able to
undo the stack: on seeing an \( a \) (or \( b \)) with \( \epsilon \) (or \( b \)) as tos, it will pop it.
The language recognized is \( L(M) = \{ w \in \{ a, b \}^* \} \).

Def.: a sequence of configurations of \( M \) is a computation path of \( M \) if
\( C_0 \) is an initial configuration \( (s, x, \epsilon) \), \( C_0 \) does not have a successor
configuration and \( C_i \vdash C_{i+1} \forall i = 0, 1, \ldots, n-1 \).
Def.: a PDA \( M \) accepts a string \( x \) if there is a computation path of \( M 
that starts with \( (s, x, \epsilon) \) and ends with \( (f, \epsilon, \epsilon) \), with \( f \in F \):
\( L(M) = \{ x \in \Sigma^* | \exists f \in F \exists (s, x, \epsilon) \vdash (f, \epsilon, \epsilon) \} \).
A useful notation for DFAs and NFAs was that of a transition diagram:
circles for states, labeled arrows for transitions. The labels were just
the characters consumed in the transition. In this case we need more,
since, besides consuming characters, we must include stack
information.

Ex. 3.28. Let \( M = (\{ q, p \}, \{ a, b, c \}, \{ a, b \}, \delta, q, (p) ) \), where
\( \delta(q, a, \epsilon) = (q, a) \); \( \delta(p, a, a) = (p, \epsilon) \); \( \delta(q, b, \epsilon) = (q, b) \);
\( \delta(p, b, b) = (p, \epsilon) \); \( \delta(q, c, \epsilon) = (p, \epsilon) \).
\( b, \epsilon/\epsilon \)
\( a, \epsilon/\epsilon \)
\( c, \epsilon/\epsilon \)

\( a, \epsilon/\epsilon \)
\( b, \epsilon/\epsilon \)

The transition diagram for the PDA: the stack information must be
maintained correctly.
Manipulating the stack.

Which do not consume input, and use non-determinism, while transition. How do we solve the problem? Add intermediate states, only problem is that we can push or pop at most one character in each transition. The solution involves pushing 2

Ex. 3.29. Construct an accepting PDA for \( \{ w \in \{ a, b \}^* \mid w \text{ has even length} \} \).

Soln. This is almost identical to 3.28, except that there is no \( c \) to alert us to move from state \( q \) to state \( p \). Non-determinism solves the problem:

\[
\begin{align*}
    & b, v/b \\
    & a, v/a \\
    & b, h/c \\
    & a, a/t \\
    & \varepsilon, v/t \\
    \end{align*}
\]

We trace three computations paths, one accepting and two rejecting:

- \((q, abba, \varepsilon) \vdash (q, bbba, a) \vdash (q, ba, ba) \vdash (p, ba, ba) \vdash (p, a, a) \vdash (p, \varepsilon, \varepsilon)\)
- \((q, abba, \varepsilon) \vdash (q, bbba, a) \vdash (q, ba, ba) \vdash (p, a, a) \vdash (p, \varepsilon, \varepsilon)\)
- \((q, abba, \varepsilon) \vdash (q, bbba, a) \vdash (p, ba, ba) \vdash (p, \varepsilon, \varepsilon)\)

Ex. 3.31. Construct an accepting PDA for \( \{ a^i b^j c^k \mid i, j, k \geq 0, i + j = k \} \).

Soln. We can construct a 3-state system:

\[
\begin{align*}
    & a, \varepsilon/a \\
    & b, \varepsilon/b \\
    & c, \varepsilon/c \\
    & \varepsilon, \varepsilon/\varepsilon \\
    \end{align*}
\]

This involves solving a simpler problem and then modifying this solution. We start with the language \( L_1 = \{ a^i b^i \mid i \geq 0 \} \). This is not the complement of the original language: anyway, the complement of a CFL is, in general, not a CFL - unlike the RL case.

The solution involves pushing 2 as on the stack for each \( a \) seen in the input string, while popping 3 as for each \( b \) seen in the input string. The main problem is that we can push or pop at most one character in each transition. How do we solve the problem? Add intermediate states, which do not consume input, and use non-determinism, while manipulating the stack.

Ex. 3.30. Construct an accepting PDA for \( \{ a^i b^i c^j \mid i, j \geq 0, i = 2j \} \).

If this were a DFA, we would simply swap accepting and non-accepting states and be done (not quite, why?). But...

The main point is that state \( q \) cannot be an accepting state, and that the inequality requires at least one \( a \) or one \( b \).

This will clearly recognize strings of the form \( a^n, b^n, n > 0 \), and \( a^n b^{2n} a b \), where \( 2n/3 \) is an integer and \( k \neq 0 \).
Now put the two together:

Non-deterministically, each \( a \) in the input gives rise to one or two \( a \)s on the stack - and the result follows.

When \( 2a/3 \) is not an integer our last pass through state \( q \) has one or two \( a \)s on the stack, with a \( b \) on the input tape, so this PDA will hang in states \( q_1 \) or \( q_2 \) rather than accepting. Modify it:

The only way we can end in a final state with an empty stack is if every \( a \) pushed by a corresponding \( b \) and vice versa. No other path will lead to an empty stack.
PDAs & CFGs

3. \( Y_{\ell} = \langle A \rightarrow w_1w_2 \cdots w_\ell \rangle \in R \), with \( w_j \in \{x \cup \Sigma^* \} \), create \( k-1 \) new states \( q_{i-1} \), \( q_{i-2} \), \ldots , \( q_{1} \), \( q_{0} \), and define
1. \( \delta(q, \varepsilon, A) = (q_{i-1}, w_1) \)
2. \( \delta(q, \varepsilon, t, e) = ((q, w_i), w_i) \) for \( 1 \leq i \leq k-2 \).
3. \( \delta(q_{0}, \varepsilon, \gamma) = ((q, w_i), w_i) \) if \( i \neq 0 \). Let \( q \), \( \gamma \) be a computation path (for \( M \)) for some \( y \in \Sigma^* \) and \( \gamma \in (V \cup \Sigma)^* \). Then there is a leftmost derivation \( S \Rightarrow_{\ast} y \gamma \) in \( G \).

Pf. Each step in the computation path performs one of two actions:
1. it consumes an input character and pops an identical character from the stack;
2. it consumes no input character and pushes a character from the right-hand-side of a grammar rule onto the stack, in right-to-left order, so that the leftmost non-terminal is nearest to when finished.

3/08  FCS  17

PDAs & CFGs

4. Assume all computation paths of length \( s \) correspond to leftmost derivations. Consider the computation path \( (q, y, S) \Rightarrow^{*} (q, t, \gamma) \), of length \( n+1 \). Assume further that \( y = x \alpha \), with \( y \in \Sigma^* \) and \( \alpha \in \Sigma \). The last step of this computation path must be \( (q, \alpha, a) \Rightarrow (q, \alpha, a \gamma) \) and \( (q, y, S) \Rightarrow (q, y, \gamma) \Rightarrow (q, t, \gamma) \). Either \( (q, y, S) \Rightarrow (q, t, \gamma) \) as soon as \( x \) is read, or \( (q, x, S) \Rightarrow (q, t, \gamma) \) where the leftmost character of \( \gamma \) (top) is a non-terminal. Expanding it (leftmost derivation) will lead to the desired form, and a corresponding sequence of leftmost derivations, since, in either case, the length of the computation path has been reduced by at least 1.

Claim: suppose \( S \Rightarrow_{\ast} y \gamma \) in \( G \), where \( y \in \Sigma^* \) and \( \gamma \in (V \cup \Sigma)^* \). Then there is a computation path in \( M \) from \( (q, y, S) \) to \( (q, t, \gamma) \).

Pf.: on your own.

3/08  FCS  19

PDAs & CFGs

Let us assume that the computation path terminates as soon as the input is consumed. We can set up the induction as follows:
1. A computation path of length 1 must have the form \( (q, \varepsilon, S) \Rightarrow (q, \varepsilon, \varepsilon) \), and this clearly corresponds to a leftmost derivation.
2. A computation path of length 2 must have the form \( (q, y, S) \Rightarrow (q, y, \gamma) \Rightarrow (q, t, \varepsilon) \), \( y \neq \varepsilon \) also corresponding to a leftmost derivation.
3. If \( y \in \Sigma \) and \( (q, y, S) \Rightarrow (q, \varepsilon, \gamma) \) then \( (q, y, S) \Rightarrow (q, y, y) \Rightarrow (q, t, \varepsilon) \), where \( (q, y, S) \Rightarrow (q, y, y \gamma) \) is a computation path that replaces tos non-terminals by the right-hand-sides of the rules until the tos is a matching terminal. These all correspond, by construction, to leftmost derivation steps.

3/08  FCS  18

PDAs & CFGs

Suppose \( x \in L(M) \). \( \exists \) a computation path \( (q, x, S) \Rightarrow^{*} (q, t, \varepsilon) \). By the first claim, \( S \Rightarrow^{\ast} x \) and thus \( x \in L(G) \). Conversely, if \( x \in L(G) \) then, by the second claim, \( S \Rightarrow^{\ast} x \) implies \( (q, x, S) \Rightarrow^{*} (q, t, \varepsilon) \). Then \( x \in L(M) \).

3/08  FCS  20
Ex. 3.36. Let $G = \langle \{S\}, \{a, b\}, \{S \rightarrow aS, aSbS, \varepsilon\}, S\rangle$. Construct the PDA.

**Soln.** We need a start state $s$, to push $S$ onto the stack before we begin reading input, and a state $q$ to be the accepting state.

The first grammar rule $S \rightarrow aS$ makes us add the loop, with one intermediate state:

```
    s
    v, v/S
    v, S/S  \
    q
    v, v/a
```

The "consumption" of input takes place only in state $q$, so the final PDA will be:

```
    s
    v, v/S
    v, S/S  \
    q
    v, v/a
    v, v/S
    v, S/S  \
    q_{2,1}  \
    q_{2,2}
```

**Theorem 3.37.** For every PDA $M$, there is a CFG $G$ s.t. $L(G) = L(M)$.

**Proof.** The central idea is to derive a grammar rule for each transition in the PDA, and then prove that the corresponding languages match (as we did for the previous theorem). One of the problems is that a PDA can have transitions that do not look at the top of the stack - that's one of the ways to avoid "hanging" on an empty stack. So we first modify $M$, to always look at $\text{tos}$, while accepting the same language.

Let $M = \langle Q, \Sigma, \Gamma, \delta, s, F\rangle$. Add a new start state $s_1$ and a new final state $f$, plus a new stack symbol $\$$ that will denote the bottom of the stack (= stack-empty for $M$). With these states, we add transitions

1. $\delta(s_1, \varepsilon, \gamma) = \{(s, \$$)\} -$ push bos symbol
2. $\delta(q, \$) = \{(f, \varepsilon)\}$ for all $q \in F$ - unique accepting state.
PDAs & CFGs

**PDAs and CFGs - Completing the Equivalence**

Since each transition looks at $t$ (and thus pops it) we must have a way to push two symbols in the same transition (this does not alter the power of the PDA: Exercise 3.4.3c) in order to add to the top of the stack. With this idea we have the further transitions

3. $(p, B) \in \delta(q, a, A)$ if $(p, B) \in \delta(q, a, A)$ for $A \in \Gamma$ and $B \in \Gamma \cup \{\varepsilon\}$.

4. $(p, BA) \in \delta(q, a, A) \forall A \in \Gamma \cup \{\varepsilon\}$, if $(p, B) \in \delta(q, a, A)$, with $B \in \Gamma \cup \{\varepsilon\}$.

This simply means that all transitions that look at $t$ remain the same, while those that don’t will pop $t$, replace it and push the new symbol, if any, that was pushed by the original transition.

It should be clear that $M$ and $M' = (Q \cup \{f\}, \Sigma, \Gamma \cup \{\varepsilon\}, \delta', s, \{f\})$ recognize exactly the same language.

Now for the grammar. We need to define the non-terminal symbols - based on the stack symbols and transitions...

---

3/2/08  
FCS  
25

---
PDAs & CFGs

CFLs - Closure Properties

**Corollary 3.39.** If \( A \in \text{RL} \) and \( B \in \text{CFL} \) then \( A \cap B \in \text{CFL} \).

*Proof.* This is an extension of the idea used for the intersection of regular languages: construct the appropriate product automaton.

Let \( M_A = (Q, \Sigma, \delta_A, q_1, F_A) \) be a DFA accepting \( A \) and \( M_B = (Q, \Sigma, \delta_B, q_2, F_B) \) be a PDA accepting \( B \). We construct a product PDA \( M \) simulating \( M_A \) and \( M_B \) in parallel:

\[ M = (Q \times Q, \Sigma, \delta, q_1 \times q_2, F), \]

where \( \delta \) is defined as follows:

\[ \delta((p, q), \epsilon) = \begin{cases} (\epsilon, \epsilon) & \text{if } p = q, \\ \delta(p, a) \times \delta(q, \epsilon) & \text{if } p \neq q. \end{cases} \]

**Note:** If \( A \) and \( B \) were both true CFLs, then the product would have to maintain TWO stacks. PDAs with two stacks are STRICTLY more powerful than PDAs with ONE stack.

PDAs & CFGs

CFLs - Closure Properties

**Corollary 3.41.** If \( L_1 \in \text{CFL} \) and \( L_2 \in \text{RL} \), then \( L_1 / L_2 \in \text{CFL} \).

*Proof.* Recall: \( x \in L_1 / L_2 \Leftrightarrow \exists y \in L_2, sy \in L_1 \). So we are looking for the prefix of a string in \( L_1 \) that has a suffix in \( L_2 \).

Usually, we claim success when the input is consumed and the stack is empty. One of the problems is that we cannot check, during the computation, whether the input is empty - so we introduce a new symbol, \( S \), the end-of-string symbol.

Our test string \( x \) will become \( sx \) when we reach the \( S \) using machine \( M_1 \); it is time to start consuming the \( y \) string via both \( M_1 \) and \( M_2 \). This will prove that \( (L_1 / L_2)S \) is context-free, although we still don’t know how to get \( y \) non-determinism will come to the rescue.

PDAs & CFGs

CFLs - Closure Properties

**Corollary 3.40.** If \( L \in \text{RL} \) and \( L_2 \in \text{CFL} \), then \( L \cap L_2 \in \text{CFL} \).

*Proof.* Recall: \( x \in L \cap L_2 \Leftrightarrow \exists y \in L_2, yx \in L \). So we are looking for the suffix of a string in \( L_2 \) that has a prefix in \( L \).

Usually, we claim success when the input is consumed and the stack is empty. One of the problems is that we cannot check, during the computation, whether the input is empty - so we introduce a new symbol, \( S \), the end-of-string symbol.

Our test string \( x \) will become \( sxs \) when we reach the \( S \) using machine \( M_1 \); it is time to start consuming the \( y \) string via both \( M_1 \) and \( M_2 \). This will prove that \( (L \cap L_2)S \) is context-free, although we still don’t know how to get \( y \) non-determinism will come to the rescue.

**Claim:** \( M_1 \) accepts \( sx \Leftrightarrow \exists a \in \text{S} \) a computation path of \( M_1 \) that accepts \( sy \) for some \( y \in L_2 \).