Pushdown Automata
Announcements

• Problem Set 5 due this Friday at 12:50PM.

• Late day extension: Using a 72-hour late day now extends the due date to 12:50PM on Tuesday, February 19th.
The Weak Pumping Lemma

- The **Weak Pumping Lemma for Regular Languages** states that

**For any** regular language $L$,

*There exists* a positive natural number $n$ such that

**For any** $w \in L$ with $|w| \geq n$,

*There exists* strings $x$, $y$, $z$ such that

**For any** natural number $i$,

$$w = xyz,$$  $w$ can be broken into three pieces,

$$y \neq \varepsilon$$  where the middle piece isn’t empty,

$$xy^iz \in L$$  where the middle piece can be replicated zero or more times.
Counting Symbols

- Consider the alphabet \( \Sigma = \{ 0, 1 \} \) and the language
  \[
  L = \{ w \in \Sigma^* \mid w \text{ contains an equal number of 0s and 1s.} \}
  \]

- For example:
  - \( 01 \in L \)
  - \( 110010 \in L \)
  - \( 11011 \notin L \)

- **Question**: Is \( L \) a regular language?
The Weak Pumping Lemma

\[ L = \{ \, w \in \{0, 1\}^* \mid w \text{ contains an equal number of } 0\text{s and } 1\text{s}. \, \} \]
An Incorrect Proof

Theorem: $L$ is regular.

Proof: We show that $L$ satisfies the condition of the pumping lemma. Let $n = 2$ and consider any string $w \in L$ such that $|w| \geq 2$. Then we can write $w = xyz$ such that $x = z = \varepsilon$ and $y = w$, so $y \neq \varepsilon$. Then for any natural number $i$, $xy^iz = w^i$, which has the same number of 0s and 1s. Since $L$ passes the conditions of the weak pumping lemma, $L$ is regular. ■
The Weak Pumping Lemma

- The **Weak Pumping Lemma for Regular Languages** states that

  **For any** regular language \( L \),

  **There exists** a positive natural number \( n \) such that

  **For any** \( w \in L \) with \( |w| \geq n \),

  **There exists** strings \( x, y, z \) such that

  **For any** natural number \( i \),

  \[
  w = xyz, \quad w \text{ can be broken into three pieces,}
  \]

  \[
  y \neq \varepsilon \quad \text{where the middle piece isn't empty,}
  \]

  \[
  xy^iz \in L \quad \text{where the middle piece can be replicated zero or more times.}
  \]
Caution with the Pumping Lemma

- The weak and full pumping lemmas describe a necessary condition of regular languages.
  - If $L$ is regular, $L$ passes the conditions of the pumping lemma.
- The weak and full pumping lemmas are not a sufficient condition of regular languages.
  - If $L$ is not regular, it still might pass the conditions of the pumping lemma!
- If a language fails the pumping lemma, it is definitely not regular.
- If a language passes the pumping lemma, we learn nothing about whether it is regular or not.
$L$ is Not Regular

- The language $L$ can be proven not to be regular using a stronger version of the pumping lemma.
- To see the full pumping lemma, we need to revisit our original insight.
An Important Observation
Pumping Lemma Intuition

- Let $D$ be a DFA with $n$ states.
- Any string $w$ accepted by $D$ that has length at least $n$ must visit some state twice \textbf{within its first $n$ characters}.
  - Number of states visited is equal $n + 1$.
  - By the pigeonhole principle, some state is duplicated.
- The substring of $w$ in-between those revisited states can be removed, duplicated, tripled, quadrupled, etc. without changing the fact that $w$ is accepted by $D$. 
The Pumping Lemma

For any regular language \( L \),

There exists a positive natural number \( n \) such that

For any \( w \in L \) with \( |w| \geq n \),

There exists strings \( x, y, z \) such that

For any natural number \( i \),

\[
    w = xyz, \quad w \text{ can be broken into three pieces},
\]
\[
    |xy| \leq n, \quad \text{where the first two pieces occur at the start of the string},
\]
\[
    y \neq \varepsilon \quad \text{where the middle piece isn't empty},
\]
\[
    xy^iz \in L \quad \text{where the middle piece can be replicated zero or more times}.
\]
Why This Change Matters

• The restriction $|xy| \leq n$ means that we can limit where the string to pump must be.

• If we specifically craft the first $n$ characters of the string to pump, we can force $y$ to have a specific property.

• We can then show that $y$ cannot be pumped arbitrarily many times.
The Pumping Lemma

\[ L = \{ w \in \{0, 1\}^* | w \text{ contains an equal number of 0s and 1s.} \} \]

Suppose the pumping length is 4.

Since \(|xy| \leq 4\), the string to pump must be somewhere in here.
\[ L = \{ \ w \in \{0, 1\}^* \mid w \text{ has an equal number of } 0\text{s and } 1\text{s} \} \]

**Theorem:** \( L \) is not regular.

**Proof:** By contradiction; assume that \( L \) is regular. Let \( n \) be the length guaranteed by the pumping lemma. Consider the string \( w = 0^n1^n \). Then \( |w| = 2n \geq n \) and \( w \in L \). Therefore, there exist strings \( x, y, \) and \( z \) such that \( w = xyz, \ |xy| \leq n, \ y \neq \epsilon, \) and for any natural number \( i, \ xy^i z \in L \). Since \( |xy| \leq n \), \( y \) must consist solely of \( 0\)s. But then \( xy^2z = 0^{n+|y|}1^n \), and since \( |y| > 0 \), we have that \( xy^2z \notin L \).

We have reached a contradiction, so our assumption was wrong and \( L \) is not regular. \( \blacksquare \)
Summary of the Pumping Lemma

- Using the pigeonhole principle, we can prove the **weak pumping lemma** and **pumping lemma**.
- These lemmas describe essential properties of the regular languages.
- Any language that fails to have these properties cannot be regular.
Beyond Finite Automata
Where We Are

- Our study of the regular languages gives us an exact characterization of problems that can be solved by finite computers.
- Not all languages are regular.
- How do we build more powerful computing devices?
The Problem

- Finite automata accept precisely the regular languages.
- We may need unbounded memory to recognize context-free languages.
  - e.g. \{ 0^n1^n | n \in \mathbb{N} \} requires unbounded counting.
- How do we build an automaton with finitely many states but unbounded memory?
The finite-state control acts as a finite memory.

The input tape holds the input string.

We can add an infinite memory device the finite-state control can use to store information.
Adding Memory to Automata

- We can augment a finite automaton by adding in a **memory device** for the automaton to store extra information.
- The finite automaton now can base its transition on both the current symbol being read and values stored in memory.
- The finite automaton can issue commands to the memory device whenever it makes a transition.
  - e.g. add new data, change existing data, etc.
Stack-Based Memory

- There are many types of memory that we might give to an automaton.
  - We'll see at least two this quarter.
- One of the simplest types of memory is a stack.
Stack-Based Memory

- Only the top of the stack is visible at any point in time.
- New symbols may be **pushed** onto the stack, which cover up the old stack top.
- The top symbol of the stack may be **popped**, exposing the symbol below it.
Pushdown Automata

• A \textbf{pushdown automaton} (PDA) is a finite automaton equipped with a stack-based memory.

• Each transition
  
  • is based on the current input symbol and the top of the stack,
  
  • optionally pops the top of the stack, and
  
  • optionally pushes new symbols onto the stack.

• Initially, the stack holds a special symbol \(z_0\) that indicates the bottom of the stack.
Our First PDA

• Consider the language

\[ L = \{ w \in \Sigma^* \mid w \text{ is a string of balanced digits} \} \]

over \( \Sigma = \{ 0, 1 \} \)

• We can exploit the stack to our advantage:
  • Whenever we see a 0, push it onto the stack.
  • Whenever we see a 1, pop the corresponding 0 from the stack (or fail if not matched)
  • When input is consumed, if the stack is empty, accept.
A Simple Pushdown Automaton

\[
\begin{align*}
0, Z_0 & \rightarrow 0Z_0 \\
0, 0 & \rightarrow 00 \\
1, 0 & \rightarrow \varepsilon \\
\varepsilon, Z_0 & \rightarrow \varepsilon
\end{align*}
\]
A Simple Pushdown Automaton

0, Z₀ → 0Z₀
0, 0 → 00
0, Z₀ → ε
1, 0 → ε

To find an applicable transition, match the current input/stack pair.

A transition of the form

\[
a, b \rightarrow z
\]

Means “If the current input symbol is a and the current stack symbol is b, then follow this transition, pop b, and push the string z.”
A Simple Pushdown Automaton

If a transition reads the top symbol of the stack, it **always** pops that symbol (though it might replace it).

0, $Z_0 \rightarrow 0Z_0$
0, 0 → 00
1, 0 → $\varepsilon$

0 0 0 1 1 1
A Simple Pushdown Automaton

Each transition then pushes some (possibly empty) string back onto the stack. Notice that the leftmost symbol is pushed onto the top.
We now push the string $\varepsilon$ onto the stack, which adds no new characters. This essentially means “pop the stack.”
A Simple Pushdown Automaton

This transition can be taken at any time $Z_0$ is atop the stack, but we've nondeterministically guessed that this would be a good time to take it.

$0, Z_0 \rightarrow 0Z_0$

$0, 0 \rightarrow 00$

$1, 0 \rightarrow \varepsilon$

$\varepsilon, Z_0 \rightarrow \varepsilon$
A Simple Pushdown Automaton

0, \( Z_0 \rightarrow 0Z_0 \)
0, 0 \( \rightarrow 00 \)
1, 0 \( \rightarrow \varepsilon \)

\( \varepsilon, Z_0 \rightarrow \varepsilon \)
The Language of a PDA

- Given a PDA $P$ and a string $w$, $P$ accepts $w$ iff there is some series of choices such that when $P$ is run on $w$, it ends in an accepting state.
  - The stack can contain any number of symbols when the machine accepts.
- The language of a PDA is the set of strings that the PDA accepts:
  \[ \mathcal{L}(P) = \{ w \in \Sigma^* \mid P \text{ accepts } w \} \]
- If $P$ is a PDA where $\mathcal{L}(P) = L$, we say that $P$ recognizes $L$. 
A Note on Terminology

- Finite automata are highly standardized.
- There are many equivalent but different definitions of PDAs.
- The one we will use is a slight variant on the one described in Sipser.
  - Sipser does not have a start stack symbol.
  - Sipser does not allow transitions to push multiple symbols onto the stack.
- Feel free to use either this version or Sipser's; the two are equivalent to one another.
A PDA for Palindromes

- A **palindrome** is a string that is the same forwards and backwards.
- Let $\Sigma = \{0, 1\}$ and consider the language
  \[
  \text{PALINDROME} = \{ w \in \Sigma^* \mid w \text{ is a palindrome} \}.
  \]
- How would we build a PDA for $\text{PALINDROME}$?
- **Idea**: Push the first half of the symbols on to the stack, then verify that the second half of the symbols match.
- **Nondeterministically** guess when we've read half of the symbols.
- This handles even-length strings; we'll see a cute trick to handle odd-length strings in a minute.
A PDA for Palindromes

This transition indicates that the transition does not pop anything from the stack. It just pushes on a new symbol instead.
A PDA for Palindromes

The $\Sigma$ here refers to the same symbol in both contexts. It is a shorthand for “treat any symbol in $\Sigma$ this way.”
A PDA for Palindromes

This transition means "don’t consume any input, don’t change the top of the stack, and don’t add anything to a stack. It’s the equivalent of an $\varepsilon$-transition in an NFA."
A PDA for Palindromes

\[
\begin{align*}
\Sigma, \varepsilon & \rightarrow \Sigma \\
\Sigma, \Sigma & \rightarrow \varepsilon \\
\varepsilon, Z_0 & \rightarrow \varepsilon
\end{align*}
\]

start

\[\varepsilon, \varepsilon \rightarrow \varepsilon\]
A PDA for Palindromes

This transition lets us consume one character before we start matching what we just saw. This lets us match odd-length palindromes.
A Note on Nondeterminism

- In an NFA, we could interpret nondeterminism as being in multiple states simultaneously.
- This is only possible because NFAs have no extra storage.
A Note on Nondeterminism

- In a PDA, if there are multiple nondeterministic choices, you **cannot** treat the machine as being in multiple states at once.
  - Each state might have its own stack associated with it.
- Instead, there are multiple parallel copies of the machine running at once, each of which has its own stack.
A PDA for Arithmetic

• Let \( \Sigma = \{ \text{int, +, *, (, )} \} \) and consider the language
  \[ ARITH = \{ w \in \Sigma^* | w \text{ is a legal arithmetic expression} \} \]

• Examples:
  \[
  \text{int + int * int} \\
  ((\text{int + int}) \ast (\text{int + int})) + (\text{int})
  \]

• Can we build a PDA for \( ARITH \)?
A PDA for Arithmetic

int, ε → ε
+
*, ε → ε
A PDA for Arithmetic

\[
\begin{align*}
\text{start} & \quad \rightarrow \quad \text{int, } \varepsilon & \rightarrow & \varepsilon \\
\text{int} & \quad \rightarrow \quad (, \varepsilon & \rightarrow & ( \\
) & \quad \rightarrow \quad \varepsilon \\
\varepsilon, Z_0 & \quad \rightarrow \quad \varepsilon
\end{align*}
\]
The Power of PDAs
Classes of Languages

- Recall: A language is **regular** iff there is a DFA, NFA, or regular expression for it.
- A language is called **context-free** iff there is a PDA for it.
  - More on that terminology next time.
- We have seen at least one language (palindromes) that is context-free but not regular.
- How do these classes relate to one another?
Regular and Context-Free Languages

**Theorem:** Any regular language is context-free.

**Proof Sketch:** Let $L$ be any regular language and consider a DFA $D$ for $L$. Then we can convert $D$ into a PDA for $L$ by converting any transition on a symbol $a$ into a transition $a, \varepsilon \rightarrow \varepsilon$ that ignores the stack. This new PDA accepts $L$, so $L$ is context-free. ■-ish
Regular Languages

CFLs

All Languages
Refining the Context-Free Languages
NPDAs and DPDAs

• With finite automata, we considered both deterministic (DFAs) and nondeterministic (NFAs) automata.

• So far, we've only seen nondeterministic PDAs (or NPDAs).

• What about deterministic PDAs (DPDAs)?
DPDAs

- A **deterministic pushdown automaton** is a PDA with the extra property that

  For each state in the PDA, and for any combination of a current input symbol and a current stack symbol, there is **at most** one transition defined.

- In other words, there is **at most** one legal sequence of transitions that can be followed for any input.

- This does **not** preclude ε-transitions, as long as there is never a conflict between following the ε-transition or some other transition.

- However, there can be **at most** one ε-transition that could be followed at any one time.

- This does **not** preclude the automaton “dying” from having no transitions defined; DPDAs can have undefined transitions.
Is this a DPDA?

This $\varepsilon$-transition is allowable because no other transitions in this state use the input symbol 0.

This $\varepsilon$-transition is allowable because no other transitions in this state use the stack symbol $Z_0$.
Why DPDAs Matter

- Because DPDAs are deterministic, they can be simulated efficiently:
  - Keep track of the top of the stack.
  - Store an action/goto table that says what operations to perform on the stack and what state to enter on each input/stack pair.
  - Loop over the input, processing input/stack pairs until the automaton rejects or ends in an accepting state with all input consumed.
- If we can find a DPDA for a CFL, then we can recognize strings in that language efficiently.
If we can find a DPDA for a CFL, then we can recognize strings in that language efficiently.

Can we guarantee that we can always find a DPDA for a CFL?
The Power of Nondeterminism

- When dealing with finite automata, there is no difference in the power of NFAs and DFAs.

- However, when dealing with PDAs, there are CFLs that can be recognized by NPDAs that cannot be recognized by DPDAs.

- Simple example: The language of palindromes.
  - How do you know when you've read half the string?

- NPDAs are more powerful than DPDAs.
Deterministic CFLs

A context-free language $L$ is called a **deterministic context-free language (DCFL)** if there is some DPDA that recognizes $L$.

Not all CFLs are DCFLs, though many important ones are.

- Balanced parentheses, most programming languages, etc.

Why are all regular languages DCFLs?
Separating DCFLs and CFLs

- It is *extremely difficult* to prove that a given CFL is not a DCFL.
- Challenge problem:

  Prove that the language of all palindromes over \( \Sigma = \{0, 1\} \) is not deterministic context-free.
Summary

• Automata can be augmented with a memory storage to increase their power.
• PDAs are finite automata equipped with a stack.
• PDAs accept precisely the context-free languages, which are a strict superset of the regular languages.
• Deterministic PDAs are strictly weaker than nondeterministic PDAs.
Next Time

- **Context-Free Grammars**
  - A different formalism for context-free languages.

- **The Limits of CFLs**
  - What problems cannot be solved by PDAs?