Pushdown Automata

12 October 2021
Assignment 4

Due today
Corrections due Thursday

Grading
Assignment 4

Due today
Corrections due Thursday

Grading
Assignment 4

Due today
Corrections due Thursday

Grading

Percent graded
Extra credit

I’ll be releasing a (small) programming assignment that counts for extra credit. You’ll be able to do it over October Break if you want to.
Where are we? Regular language vs context-free languages
Possible relations

- REG and CFL
- REG and CFL
- REG and CFL
- REG and CFL
- REG and CFL
- REG and CFL
- REG and CFL
- REG and CFL
Beware!

Consider the CFG $G$:

$$S \rightarrow a*b$$

$L(G)$ contains one string, $a*b$.

$L(G) \neq \{a^n b \mid n \in \mathbb{N}_0\}$
CFGs just consist of production rules with a concatenation of terminals and nonterminals.

They don’t have the regular expression operators Kleene star, union, or parenthesized expressions.

But we can convert regular expressions to CFGs!
THEOREM. Every regular language is context-free.

PROOF IDEA. Show how to convert an arbitrary regular expression for $L$ into a CFG for $L$. 
In regular expressions, we could write \* for 0-or-more repetitions, e.g.,

\[ S \rightarrow a^*b \]
In regular expressions, we could write * for 0-or-more repetitions, e.g.,

\[ S \rightarrow a^* b \]
In regular expressions, we could write * for 0-or-more repetitions, e.g.,

\[ S \rightarrow a^*b \]

\[ A \rightarrow Aa \mid \varepsilon \]
In regular expressions, we could write * for 0-or-more repetitions, e.g.,

\[ S \rightarrow a^*b \]

\[ A \rightarrow Aa \mid \varepsilon \]
In regular expressions, we could write * for 0-or-more repetitions, e.g.,

\[ S \rightarrow Ab \]

\[ A \rightarrow Aa \mid \varepsilon \]
In regular expressions, we could write parentheses to group expressions and \( \cup \) for alternatives, e.g.,

\[
S \rightarrow a(b \cup c*)
\]
In regular expressions, we could write parentheses to group expressions and \( \cup \) for alternatives, e.g.,

\[
S \rightarrow a(b \cup c*) \\
X \rightarrow b | c*
\]
In regular expressions, we could write parentheses to group expressions and \( \cup \) for alternatives, e.g.,

\[
S \rightarrow a(b \cup c*) \\
X \rightarrow b | c*
\]
In regular expressions, we could write parentheses to group expressions and ∪ for alternatives, e.g.,

\[ S \rightarrow aX \]

\[ X \rightarrow b | c* \]
In regular expressions, we could write parentheses to group expressions and $\cup$ for alternatives, e.g.,

$$S \to aX$$

$$X \to b \mid c^*$$
In regular expressions, we could write parentheses to group expressions and $\cup$ for alternatives, e.g.,

$$S \rightarrow aX$$

$$X \rightarrow b \mid c^*$$

$$C \rightarrow Cc \mid \varepsilon$$
In regular expressions, we could write parentheses to group expressions and \( \cup \) for alternatives, e.g.,

\[
S \rightarrow aX \\
X \rightarrow b \mid c^* \\
C \rightarrow Cc \mid \varepsilon
\]
In regular expressions, we could write parentheses to group expressions and $\cup$ for alternatives, e.g.,

\[
S \rightarrow aX \\
X \rightarrow b \mid C \\
C \rightarrow Cc \mid \varepsilon
\]
Possible relations
Consider the following CFG $G$:

$$S \rightarrow aSb \mid \varepsilon$$

What strings can this generate?
Consider the following CFG $G$:

$$S \rightarrow aSb \mid \varepsilon$$

What strings can this generate?
Consider the following CFG $G$:

$$S \rightarrow aSb \mid \varepsilon$$

What strings can this generate?

\[
\begin{array}{ccc}
a & s & b \\
\end{array}
\]
Consider the following CFG $G$:

$$S \rightarrow aSb \mid \varepsilon$$

What strings can this generate?
Consider the following CFG $G$:

$$S \rightarrow aSb \mid \varepsilon$$

What strings can this generate?

a a s b b
Consider the following CFG $G$:

$$S \rightarrow aSb \mid \varepsilon$$

What strings can this generate?
Consider the following CFG $G$:

$$S \rightarrow aSb \mid \varepsilon$$

What strings can this generate?

```
 a   a   a   s   b   b   b
```
Consider the following CFG $G$:

$$S \rightarrow aSb \mid \varepsilon$$

What strings can this generate?

```
  a a a
     S
    b b b
```
Consider the following CFG $G$:

$$S \rightarrow aSb \mid \varepsilon$$

What strings can this generate?

\[
\begin{array}{cccccccc}
\text{a} & \text{a} & \text{a} & \text{a} & \text{s} & \text{b} & \text{b} & \text{b} \\
\end{array}
\]
Consider the following CFG $G$:

$$ S \rightarrow aSb \mid \varepsilon $$

What strings can this generate?

| a | a | a | a | a | b | b | b | b | b |
Consider the following CFG $G$:

$$S \rightarrow aSb \mid \varepsilon$$

What strings can this generate?

```
  a a a a b b b b b
```
Consider the following CFG $G$:

$$S \rightarrow aSb \mid \varepsilon$$

What strings can this generate?

$L(G) = \{a^n b^n \mid n \in \mathbb{N}_0\}$
Regular languages

Context-free languages

All languages
Why do CFGs have more power than regular expressions?

*Intuition:* Derivations of strings have unbounded “memory”.
Why do CFGs have more power than regular expressions?

*Intuition:* Derivations of strings have unbounded “memory”.

\[ S \rightarrow aSb \mid \varepsilon \]
Why do CFGs have more power than regular expressions?

*Intuition:* Derivations of strings have unbounded “memory”.

\[ S \rightarrow aSb \mid \varepsilon \]
Why do CFGs have more power than regular expressions?

*Intuition*: Derivations of strings have unbounded “memory”.

\[ S \rightarrow aSb | \epsilon \]
Why do CFGs have more power than regular expressions?

*Intuition:* Derivations of strings have unbounded “memory”.

\[ S \rightarrow aSb \mid \varepsilon \]
Why do CFGs have more power than regular expressions?

*Intuition:* Derivations of strings have unbounded “memory”.

\[ S \rightarrow aSb \mid \epsilon \]
Why do CFGs have more power than regular expressions?

**Intuition:** Derivations of strings have unbounded “memory”.

\[ S \rightarrow aSb \mid \epsilon \]

\[
\begin{array}{ccc}
    a & a & S \\
 \hline
    b & b & b \\
\end{array}
\]
Why do CFGs have more power than regular expressions?

*Intuition*: Derivations of strings have unbounded “memory”.

\[ S \rightarrow aSb \mid \epsilon \]

\[
\begin{array}{cccccc}
\text{a} & \text{a} & \text{a} & \text{s} & \text{b} & \text{b} & \text{b}
\end{array}
\]
Why do CFGs have more power than regular expressions?

*Intuition:* Derivations of strings have unbounded “memory”.

\[ S \rightarrow aSb \mid \varepsilon \]

| a | a | a | s | b | b | b | b |
Why do CFGs have more power than regular expressions?

*Intuition:* Derivations of strings have unbounded “memory”.

\[ S \rightarrow aSb \mid \varepsilon \]
Why do CFGs have more power than regular expressions?

*Intuition:* Derivations of strings have unbounded “memory”.

$$S \rightarrow aSb \mid \varepsilon$$

| a | a | a | a | a | b | b | b | b | b |
Why do CFGs have more power than regular expressions?

*Intuition*: Derivations of strings have unbounded “memory”.

\[ S \rightarrow aSb \mid \varepsilon \]
Pushdown automata
Finite automata accept precisely the regular languages.

We may need unbounded memory to recognize context-free languages.

E.g., \( \{0^n1^n \mid n \in \mathbb{N}_0\} \) requires unbounded counting.

How do we build an automaton with finitely many states but unbounded memory?
*Pushdown automata* (PDAs) are finite automata like DFAs and NFAs, but with the addition of a *stack.*
Schematic of a finite automaton

Finite-state control

Input: a a b a c
Schematic of a finite automaton

Finite-state control

An infinite memory device the finite-state control can use to store information

input

stack
Now the finite automaton can base its transition on both the current symbol being read and values stored in memory.

It can issue commands to read or write from this memory whenever it makes a transition.
There are many types of memory we can give to an automaton.

We’ll consider two in this class, starting with *stack-based memory* for pushdown automata.
Push “a”
Push “b”
Push “c”
Push “d”
Pop “d”
Push “e”
Pop “e”
Pop “c”
In stack-based memory, only the top of the stack is visible at any point in time.

New symbols may be *pushed* onto the stack, rendering the previous top symbol inaccessible.

The top symbol of the stack may be *popped*, exposing the symbol below it.
We’ll be using **nondeterministic** PDAs, which are equivalent in power to CFGs.

Different perspectives: CFGs generate strings; PDAs recognize them. There **are** deterministic PDAs, but they’re less powerful than the nondeterministic ones – not like DFAs and NFAs, which are equivalent!
Notation

At state $p$, if you can

read the symbol $x$ from the input and
pop the symbol $y$ from the stack.

then you can

enter state $q$ and
push the symbol $z$ onto the stack.
Notation

At state $p$, if you can

read the symbol $x$ from the input and
pop the symbol $y$ from the stack.

then you can

enter state $q$ and
push the symbol $z$ onto the stack.

Don’t read a symbol from the input

$\epsilon, y \rightarrow z$
Notation

At state \( p \), if you can

read the symbol \( x \) from the input and
pop the symbol \( y \) from the stack.

then you can

enter state \( q \) and
push the symbol \( z \) onto the stack.
Notation

At state \( p \), if you can

read the symbol \( x \) from the input and
pop the symbol \( y \) from the stack.

then you can

enter state \( q \) and
push the symbol \( z \) onto the stack.

Don’t push a symbol onto the stack
Example: $a^n b^n$

Input:

```
a a a b b b
```

Stack:

```
q0
```

Transition rules:

- $a, \varepsilon \rightarrow A$
- $b, A \rightarrow \varepsilon$
- $\varepsilon, \varepsilon \rightarrow \$
- $\varepsilon, \$ \rightarrow \varepsilon$

States:

- $q_0$: Start
- $q_1$
- $q_2$
- $q_3$
What’s with the weird spring diagram?
What’s with the weird spring diagram?
What’s with the weird spring diagram?
What’s with the weird spring diagram?

The spring of a stack of plates, shown on its side
Example: $a^n b^n$

```
Input

Stack

$q_0 \xrightarrow{\epsilon, \epsilon} \epsilon$
$q_1 \xrightarrow{a, \epsilon} A \xrightarrow{b, A} \epsilon$
$q_2 \xrightarrow{b, A} \epsilon$
$q_3 \xrightarrow{\epsilon, \epsilon} \epsilon$
```

Transition rules:
- $a, \epsilon \rightarrow A$
- $b, A \rightarrow \epsilon$
- $\epsilon, \epsilon \rightarrow \$
- $\epsilon, \$ \rightarrow \epsilon$
- $\epsilon \rightarrow \epsilon$

Graph:
- $q_0$ (start state)
- $q_1$
- $q_2$
- $q_3$

Input sequence: a a a b b b b
Example: $a^nb^n$

Input

Stack

$\varepsilon, \varepsilon \rightarrow \varepsilon$

$q_0 \rightarrow q_1$

$q_1 \rightarrow q_2$

$q_2 \rightarrow q_3$

$a, \varepsilon \rightarrow A$

$b, A \rightarrow \varepsilon$

$\varepsilon, \varepsilon \rightarrow \varepsilon$

$\varepsilon, \$ \rightarrow \varepsilon$

$\varepsilon, \$ \rightarrow \varepsilon$

$aabbb$
Example: $a^n b^n$

Input: a a a b b b b

Start

Stack

$ is our special “bottom of stack” symbol

Graph:
- Initial state: $q_0$
- States: $q_0, q_1, q_2, q_3$
- Transitions:
  - $a, \epsilon \rightarrow A$
  - $b, A \rightarrow \epsilon$
  - $\epsilon, \epsilon \rightarrow \$
  - $\epsilon, \$ \rightarrow \epsilon$
  - $b, A \rightarrow \epsilon$

Stack transitions:
- $\epsilon \rightarrow \$
- $\epsilon \rightarrow \$
Example: $a^n b^n$

**Input:**
```
alaabb
```

**Stack:**
```
A $)
```
Example: $a^n b^n$

Input: `a a a b b b b`  
Stack: `A A A $`
Example: $a^n b^n$

Input:
```
a a a b b b b
```

Stack:
```
A A A A $
```

Rules:
- $a, \varepsilon \rightarrow A$
- $b, A \rightarrow \varepsilon$
- $\varepsilon, \varepsilon \rightarrow \$
- $\varepsilon, \$ \rightarrow \varepsilon$
Example: $a^n b^n$

Input: $a a a b b b$

Stack: $A A A$

Transition Rules:
- $a, \varepsilon \rightarrow A$
- $b, A \rightarrow \varepsilon$
- $\varepsilon, \varepsilon \rightarrow \$
- $\varepsilon, \$ \rightarrow \varepsilon$

States:
- $q_0$: Start state
- $q_1$
- $q_2$
- $q_3$
Example: $a^n b^n$

```
a, \epsilon \rightarrow A
b, A \rightarrow \epsilon
\epsilon, \epsilon \rightarrow \$
\epsilon, \$ \rightarrow \epsilon
```

**Input**: a a a b b b

**Stack**: A $
Example: $a^n b^n$

```
Input
a a a b b b b

Stack
$\epsilon$
$\epsilon \rightarrow \epsilon$
$a, \epsilon \rightarrow A$
$b, A \rightarrow \epsilon$
$q_0 \rightarrow q_1$
$q_1 \rightarrow q_2$
$q_2 \rightarrow q_3$
$q_3 \rightarrow q_2$
```

$q_0$ (Start state)
$q_1$
$q_2$
$q_3$
Example: $a^n b^n$

Input: $a a a b b b$

Stack:

Transition rules:
- $a, \varepsilon \rightarrow A$
- $b, A \rightarrow \varepsilon$
- $\varepsilon, \varepsilon \rightarrow \$
- $\varepsilon, \$ \rightarrow \varepsilon$

States:
- $q_0$: Start state
- $q_1$
- $q_2$
- $q_3$: Accept state
Exercise

Design a PDA to recognize the language $L = \{a^ib^jc^k \mid i = j + k \}$
Exercise

Design a PDA to recognize the language \( L = \{a^ib^jc^k \mid i = j + k \} \)
**Exercise**

Design a PDA to recognize the language $L = \{a^ib^jc^k \mid i = j + k \}$

*Or, exploiting non-determinism,*
Example: Palindromes

Recall: A *palindrome* is a string that is the same forwards and backwards.

Let $\Sigma = \{a, b, c\}$ and consider the language

$L = \{wcw^R \mid w \in \{a, b\}^*\}$

E.g.,

- $c$
- $aca$
- $bcb$
- $abcba$
- $bbcba$
- $bbcbb$
- ...


Example: Palindromes

Example of a context-free grammar for palindromes:

\[
\begin{align*}
\text{start} & \rightarrow \varepsilon, \varepsilon \rightarrow \$ \\
a, \varepsilon & \rightarrow A \\
b, \varepsilon & \rightarrow B \\
a, A & \rightarrow \varepsilon \\
b, B & \rightarrow \varepsilon \\
\end{align*}
\]

Input: \text{a a a b c b a a a} \\
Stack: \text{ }
Example: Palindromes

Input

a a a b c b a a a

Stack

$
Example: Palindromes

Input

Stack

```
a a a b c b a a a
```

```
A $)
```
Example: Palindromes

Input: a a a b c b a a a

Stack: A A $
Example: Palindromes

Input: a a a b c b a a a

Stack: A A A $
Example: Palindromes

\[
\begin{align*}
\text{Input} & : & \text{Stack} \\
a & a & a & b & c & b & a & a & a & | & B & A & A & A & $ \\
\end{align*}
\]
Example: Palindromes

Input: a a a b c b a a a

Stack: B A A A A $
Example: Palindromes

$$\begin{align*}
\varepsilon, \varepsilon &\rightarrow \$ \\
a, \varepsilon &\rightarrow A \\
b, \varepsilon &\rightarrow B \\
a, A &\rightarrow \varepsilon \\
b, B &\rightarrow \varepsilon \\
\varepsilon, \varepsilon &\rightarrow \varepsilon \\
\varepsilon &\rightarrow A \\
\varepsilon &\rightarrow B \\
\varepsilon &\rightarrow \varepsilon
\end{align*}$$

Input: a a a b c b a a a

Stack: A A A A $
Example: Palindromes

**Input**

```
 a a a b c b a a a
```

**Stack**

```
A A $ (empty stack)
```

Transition rules:
- $, $ → $ (non-deterministic start)
- $, ε → A
- A, ε → ε
- B, ε → ε
- C, ε → ε
- a, ε → A
- a, A → ε
- b, ε → B
- b, B → ε
- c, ε → ε
- ε, $ → $ (acceptance state)
Example: Palindromes

\[
\begin{align*}
a, \varepsilon &\rightarrow A \\
b, \varepsilon &\rightarrow B \\
a, A &\rightarrow \varepsilon \\
b, B &\rightarrow \varepsilon
\end{align*}
\]

Input: a a a b c b a a a

Stack: A $
Example: Palindromes

Input: a a a b c b a a a

Stack: $
Example: Palindromes

Input: a a a b c b a a a

Stack:

- $ → \varepsilon
- \varepsilon → a, \varepsilon → A
- \varepsilon → b, \varepsilon → B
- c, \varepsilon → \varepsilon
- \varepsilon, \$ → \varepsilon
- a, A → \varepsilon
- b, B → \varepsilon
- Accept!
What about building a DFA to recognize the language of palindromes without a special dividing character like $c$?

Let $\Sigma = \{a, b\}$ and consider the language

$$PALINDROME = \{w \in \Sigma^* \mid w \text{ is a palindrome}\}.$$  

How would we build a PDA for $PALINDROME$?

Nondeterminism to the rescue!
\[ \begin{align*}
\epsilon, \epsilon & \rightarrow \epsilon, \\
\epsilon, \epsilon & \rightarrow \epsilon, \\
a, \epsilon & \rightarrow A, \\
b, \epsilon & \rightarrow B, \\
a, A & \rightarrow \epsilon, \\
b, B & \rightarrow \epsilon
\end{align*} \]
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 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 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.
Formally, a pushdown automaton is a nondeterministic machine defined by the six-tuple $(Q, \Sigma, \Gamma, \delta, q_0, F)$. 
Formally, a *pushdown automaton* is a nondeterministic machine defined by the six-tuple $(Q, \Sigma, \Gamma, \delta, q_0, F)$.

$Q$ is a finite set of states,
Formally, a *pushdown automaton* is a nondeterministic machine defined by the six-tuple $(Q, \Sigma, \Gamma, \delta, q_0, F)$.

- $Q$ is a finite set of states,
- $\Sigma$ is the alphabet for input strings,
Formally, a *pushdown automaton* is a nondeterministic machine defined by the six-tuple $(Q, \Sigma, \Gamma, \delta, q_0, F)$.

- $Q$ is a finite set of states,
- $\Sigma$ is the alphabet for input strings,
- $\Gamma$ is the *stack alphabet* of symbols that can be pushed on the stack.
Formally, a *pushdown automaton* is a nondeterministic machine defined by the six-tuple $(Q, \Sigma, \Gamma, \delta, q_0, F)$.

- $Q$ is a finite set of states,
- $\Sigma$ is the alphabet for input strings,
- $\Gamma$ is the *stack alphabet* of symbols that can be pushed on the stack.

The stack alphabet allows PDAs’ stacks to store extra information that can’t otherwise be encoded by the input string.
Formally, a **pushdown automaton** is a nondeterministic machine defined by the six-tuple \((Q, \Sigma, \Gamma, \delta, q_0, F)\).

- \(Q\) is a finite set of states,
- \(\Sigma\) is the alphabet for input strings,
- \(\Gamma\) is the **stack alphabet** of symbols that can be pushed on the stack,
- \(\delta: Q \times \Sigma \times \Gamma \rightarrow \wp(Q \times \Gamma)\) is the *transition function*
Formally, a *pushdown automaton* is a nondeterministic machine defined by the six-tuple $(Q, \Sigma, \Gamma, \delta, q_0, F)$.

- $Q$ is a finite set of states,
- $\Sigma$ is the alphabet for input strings,
- $\Gamma$ is the *stack alphabet* of symbols that can be pushed on the stack,
- $\delta: Q \times \Sigma \times \varepsilon \times \Gamma \varepsilon \rightarrow \wp(Q \times \Gamma \varepsilon)$ is the *transition function*

Each transition is based on a combination of the current state, the current input symbol, and the current stack symbol.
Formally, a **pushdown automaton** is a nondeterministic machine defined by the six-tuple $(Q, \Sigma, \Gamma, \delta, q_0, F)$.

- $Q$ is a finite set of states,
- $\Sigma$ is the alphabet for input strings,
- $\Gamma$ is the *stack alphabet* of symbols that can be pushed on the stack,
- $\delta : Q \times \Sigma \times \Gamma_\varepsilon \rightarrow \wp(Q \times \Gamma_\varepsilon)$ is the *transition function*.

Each transition is based on a combination of the current state, the current input symbol, and the current stack symbol.

The function maps to a set of state/string pairs, and the string is pushed atop the stack.
Formally, a *pushdown automaton* is a nondeterministic machine defined by the six-tuple $(Q, \Sigma, \Gamma, \delta, q_0, F)$.

- $Q$ is a finite set of states,
- $\Sigma$ is the alphabet for input strings,
- $\Gamma$ is the *stack alphabet* of symbols that can be pushed on the stack,
- $\delta : Q \times \Sigma \times \Gamma \rightarrow \wp(Q \times \Gamma)$ is the *transition function*,
- $q_0 \in Q$ is the *start state*, and
Formally, a *pushdown automaton* is a nondeterministic machine defined by the six-tuple 
\((Q, \Sigma, \Gamma, \delta, q_0, F)\).

- \(Q\) is a finite set of states,
- \(\Sigma\) is the alphabet for input strings,
- \(\Gamma\) is the *stack alphabet* of symbols that can be pushed on the stack,
- \(\delta: Q \times \Sigma \times \Gamma \rightarrow \mathcal{P}(Q \times \Gamma)\) is the *transition function*,
- \(q_0 \in Q\) is the *start state*, and
- \(F \subseteq Q\) is the set of *accepting states*. 
Formally, a *pushdown automaton* is a nondeterministic machine defined by the six-tuple $(Q, \Sigma, \Gamma, \delta, q_0, F)$.

- $Q$ is a finite set of states,
- $\Sigma$ is the alphabet for input strings,
- $\Gamma$ is the *stack alphabet* of symbols that can be pushed on the stack,
- $\delta: Q \times \Sigma \epsilon \times \Gamma \epsilon \rightarrow \wp(Q \times \Gamma \epsilon)$ is the *transition function*,
- $q_0 \in Q$ is the *start state*, and
- $F \subseteq Q$ is the set of *accepting states*.

The automaton accepts if it ends in an accepting state with no input remaining.
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:

$$L(P) = \{w \in \Sigma^* \mid P \text{ accepts } w\}.$$  

If $P$ is a PDA where $L(P) = L$, we say that $P$ **recognizes** the language $L$. 
Acknowledgments

This lecture incorporates material from:

Nancy Ide, Vassar College
Keith Schwarz, Stanford University
Michael Sipser, *Introduction to the Theory of Computation*
Jeffrey Ullman, Stanford University
Jennifer Walter, Vassar College