Pushdown Automata

11 October 2022
Assignment 5

Due today
Corrections due Thursday

Exam 1

Grades out later today
Review the example solutions
Presentation on the AIT Budapest program for CS majors
We welcome Gabor Bojar, AIT-Budapest
Date: Wednesday, October 26, 2022
Time: 3:15pm
Place: SP 105

https://www.ait-budapest.com/about-ait/overview
Daniela Rus

Daniela Rus is the Andrew (1956) and Erna Viterbi Professor of Electrical Engineering and Computer Science, Director of the Computer Science and Artificial Intelligence Laboratory (CSAIL) at MIT, and Deputy Dean of Research in the Schwarzman College of Computing at MIT.

Daniella's research interests are in robotics and artificial intelligence. The key focus of her research is to develop the science and engineering of autonomy. Rus is a MacArthur Fellow, a fellow of ACM, AAAI and IEEE, a member of the National Academy of Engineering, and of the American Academy of Arts and Sciences. She is a senior visiting fellow at MITRE Corporation. She is the recipient of the Engelberger Award for robotics and the IEEE RAS Pioneer award. She earned her PhD in Computer Science from Cornell University.
Where are we?
Our study of the *regular languages* gives us an exact characterization of problems that can be solved by finite computers.
All languages

Regular languages

Context-free languages

All languages
We can describe context-free languages – including nonregular languages like \( \{a^n b^n \mid n \in \mathbb{N}_0\} \) – using context-free grammars.
Grammars are *language generators*.

It’s not immediately clear how they might be used as *language recognizers* – though we’ll return to that question!
Finite-state control

input

aabac
To recognize context-free languages, we may need unbounded memory.

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

Finite-state control

Memory device

aabac input
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 might give to an automaton, but one of the simplest is a stack.
Push “a”
Push “b”
Push “c”
Push “d”
Pop “d”
Push “e”
Pop “e”
Pop “c”
Only the top of the stack is visible at any point in time.

New symbols can be \textit{pushed} onto the stack, rendering the previous top symbol inaccessible.

The top symbol of the stack may be \textit{popped}, exposing the symbol below it.
A *pushdown automaton* (PDA) is a finite automaton equipped with stack-based memory.
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.
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 pop a symbol from 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: $a^n b^n$
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: a a a b b b b

Stack:

$$
\begin{align*}
q_0 & \xrightarrow{\varepsilon, \varepsilon} q_1 \\
q_1 & \xrightarrow{a, \varepsilon} A \\
q_2 & \xrightarrow{b, A} \varepsilon \\
q_3 & \xrightarrow{\varepsilon, \varepsilon} \varepsilon \\
q_0 & \xrightarrow{\varepsilon, \varepsilon} $ \\
q_2 & \xrightarrow{\varepsilon, \varepsilon} \varepsilon \\
\end{align*}
$$
Example: $a^n b^n$

Input:

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

Stack: 

```
ε, $ \rightarrow \epsilon$
```

Diagram:

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

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

Input: \texttt{a a a b b b b}

Stack:

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

$q_0$ is our special "bottom of stack" symbol.
Example: $a^n b^n$

Input: a a a b b b b

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

Input: $a a a b b b b$

Stack: $A A A$

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

Input: a a a b b b b

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

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

Stack: 
```
A A $ 
```

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

Input: $aabaabbb$

Stack: $A$

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

Transition Diagram:
- Start in $q_0$
- From $q_0$ on $a, \varepsilon \rightarrow A$
- From $q_1$ on $b, A \rightarrow \varepsilon$
- From $q_2$ on $\varepsilon, \varepsilon \rightarrow \$$$
- From $q_3$ on $\varepsilon, \$$ \rightarrow \varepsilon$
Example: $a^n b^n$
Example: $a^n b^n$

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

**Stack:**

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

**States:**
- $q_0$: Start
- $q_1$: Transition on $a, \varepsilon \rightarrow A$
- $q_2$: Transition on $b, A \rightarrow \varepsilon$
- $q_3$: Accepting state
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 accept state.

The language of a PDA is the set of strings that 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 $L$. 

The stack can contain any number of symbols when the machine accepts.
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 \textit{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 a PDA’s stack 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,
- $\delta : Q \times \Sigma \times \Gamma \rightarrow \mathcal{P}(Q \times \Gamma)$ 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,
- \(\delta: Q \times \Sigma \times \Gamma \rightarrow \mathcal{P}(Q \times \Gamma)\) 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,
- $\delta : Q \times \Sigma \times \Gamma \rightarrow \mathcal{P}(Q \times \Gamma)$ 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,
- $\delta: Q \times \Sigma \times \Gamma \epsilon \rightarrow \mathcal{P}(Q \times \Gamma \epsilon)$ is the *transition function*,
- $q_0 \in Q$ is the *start state*, and
- $F \subseteq Q$ is the set of *accept 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 *accept states*.

The automaton accepts if it ends in an accept state with no input remaining.
Beware!

While finite automata are highly standardized, there are many different – but equivalent – definitions of PDAs.

The one we’ll use matches Sipser.

Avoid other materials – videos, textbooks, etc. – or you may get confused.
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
- abcba
- bbcbb
- ...
Example: Palindromes

Input: a a a b c b a a a

Stack:

Transition Rules:
- $, \epsilon \rightarrow A$
- $b, \epsilon \rightarrow B$
- $a, A \rightarrow \epsilon$
- $b, B \rightarrow \epsilon$
- $\epsilon, \epsilon \rightarrow \epsilon$
- $\epsilon, \epsilon \rightarrow \epsilon$
- $\epsilon, \epsilon \rightarrow \epsilon$
- $\epsilon, \epsilon \rightarrow \epsilon$
- $\epsilon, \epsilon \rightarrow \epsilon$
- $\epsilon, \epsilon \rightarrow \epsilon$
Example: Palindromes

Input: \[\text{a a a b c b a a a}\]

Stack: \[\text{\$}\]

Transition Rules:
- \(a, \varepsilon \rightarrow A\)
- \(b, \varepsilon \rightarrow B\)
- \(a, A \rightarrow \varepsilon\)
- \(b, B \rightarrow \varepsilon\)
Example: Palindromes

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: A A $
Example: Palindromes

Input: a a a b c b a a a

Stack: A A A A $
Example: Palindromes

\[
\begin{align*}
&\text{Input} & & \text{Stack} \\
\end{align*}
\]

\[
\begin{align*}
&\text{Graph} \\
&\text{start} & \rightarrow & \epsilon, \epsilon \rightarrow \$ & & \rightarrow & \epsilon, \$ \rightarrow \epsilon & & \rightarrow & \epsilon, \epsilon \rightarrow \epsilon & & \rightarrow & \epsilon, \epsilon \rightarrow \epsilon & & \rightarrow & \epsilon, \epsilon \rightarrow \epsilon & & \rightarrow & \epsilon, \epsilon \rightarrow \epsilon & & \rightarrow & \epsilon, \epsilon \rightarrow \epsilon \\
\end{align*}
\]

\[
\begin{align*}
&\text{Rules} \\
&a, \epsilon \rightarrow A \\
b, \epsilon \rightarrow B \\
a, A \rightarrow \epsilon \\
b, B \rightarrow \epsilon \\
c, \epsilon \rightarrow \epsilon \\
\end{align*}
\]
Example: Palindromes

Input: a a a b c b a a a

Stack: B A A A A $
Example: Palindromes

$\begin{align*}
\text{Input:} & \quad a \ a \ a \ b \ c \ b \ a \ a \ a \\
\text{Stack:} & \quad A \ A \ A \ A \ \$ \\
\end{align*}$
Example: Palindromes

\[ \text{Stack} \]

\[ \text{Input} \]

\[ a \ a \ a \ b \ c \ b \ a \ a \ a \]

\[ \text{A} \ A \$ \]
Example: Palindromes

Input: a a a b c b a a a a

Stack: A $
Example: Palindromes

Input: a a a b c b a a a

Stack:

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

Input

a a a b c b a a a

Stack

Accept!
What about building a PDA 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!
\[
\begin{align*}
\epsilon, \epsilon \rightarrow A & \quad a, \epsilon \rightarrow A \\
\epsilon, \epsilon \rightarrow B & \quad b, \epsilon \rightarrow B \\
\epsilon, \epsilon \rightarrow \epsilon & \quad a, \epsilon \rightarrow \epsilon, b, \epsilon \rightarrow \epsilon
\end{align*}
\]
ε, ε → $,

a, ε → A,
b, ε → B

ε, ε → ε
a, ε → ε
b, ε → ε

ε, ε → ε

a, A → ε
b, B → ε

ε, $ → ε
Exercise

Design a PDA to recognize the language

\[ \text{ADD} = \{1^m1^n = 1^m+n \mid m, n \in \mathbb{N}_0\}. \]
Exercise

Design a PDA to recognize the language

$$ADD = \{1^m1^n = 1^{m+n} | m, n \in \mathbb{N}_0\}.$$
A PDA for arithmetic

Let $\Sigma = \{\text{int}, +, \times, (, )\}$.

Consider the language

$$ARITH = \{w \in \Sigma^* | w \text{ is a legal arithmetic expression}\}.$$ 

E.g.,

\[
\begin{align*}
\text{int} &+ \text{int} \times \text{int} \\
((\text{int} + \text{int}) \times (\text{int} + \text{int})) &+ (\text{int})
\end{align*}
\]

Can we build a PDA for $ARITH$?
A PDA for arithmetic
A PDA for arithmetic
A PDA for arithmetic
A PDA for arithmetic
A PDA for arithmetic

\[
\begin{align*}
\varepsilon, \varepsilon & \rightarrow \$ \\
(, \varepsilon & \rightarrow ( \\
\text{int, } \varepsilon & \rightarrow \varepsilon \\
\text{+}, \varepsilon & \rightarrow \varepsilon \\
\times, \varepsilon & \rightarrow \varepsilon
\end{align*}
\]
A PDA for arithmetic

\[
\begin{align*}
\text{start} & \quad \varepsilon, \varepsilon \rightarrow \$ \\
\varepsilon, \varepsilon & \rightarrow (, (, \varepsilon) \rightarrow \varepsilon, ( \rightarrow \varepsilon \\
\text{int, } \varepsilon & \rightarrow \varepsilon \\
\text{+, } \varepsilon & \rightarrow \varepsilon \\
\text{x, } \varepsilon & \rightarrow \varepsilon
\end{align*}
\]
A PDA for arithmetic
A PDA for arithmetic

Input

Stack

\text{int + int} \times \text{int}
A PDA for arithmetic

Input

\[
\text{int } + \text{ int } \times \text{ int}
\]

Stack
A PDA for arithmetic

Input

int + int × int

Stack

$
A PDA for arithmetic

$$\begin{align*}
\text{Input} & \quad \text{Stack} \\
\text{start} & \quad \varepsilon, \varepsilon \rightarrow \$ \\
\varepsilon, \varepsilon \rightarrow \$ & \quad \text{int, } \varepsilon \rightarrow \varepsilon \\
\$ & \quad \varepsilon, \$ \rightarrow \varepsilon \\
+ & \quad \varepsilon, \varepsilon \rightarrow \varepsilon \\
\times & \quad \varepsilon, \varepsilon \rightarrow \varepsilon
\end{align*}$$
A PDA for arithmetic

\[+, \varepsilon \rightarrow \varepsilon\]
\[\times, \varepsilon \rightarrow \varepsilon\]
\[\text{int}, \varepsilon \rightarrow \varepsilon\]

Input: \text{int + int \times int}

Stack: $
A PDA for arithmetic

Input

\text{int + int} \times \text{int}

Stack

$$_{\text{start}}$$
A PDA for arithmetic

Input

Stack

\[ \text{int + int} \times \text{int} \]
A PDA for arithmetic

\[
\begin{align*}
\text{Input} & : \text{int } + \text{ int } \times \text{ int} \\
\text{Stack} & : \epsilon, \epsilon \rightarrow \epsilon, (, \epsilon \rightarrow (, ), ( \rightarrow \epsilon, \text{int}, \epsilon \rightarrow \epsilon, +, \epsilon \rightarrow \epsilon, \times, \epsilon \rightarrow \epsilon, \epsilon, \epsilon \rightarrow \epsilon, \epsilon, \epsilon \rightarrow \epsilon, \epsilon, \epsilon \rightarrow \epsilon
\end{align*}
\]
A PDA for arithmetic

Input: int + int × int

Stack:
A PDA for arithmetic

\[ \text{Input} \quad \varepsilon, \varepsilon \rightarrow \$ \]

\[ \text{Stack} \quad \varepsilon, \$ \rightarrow \varepsilon \]

\[ (, \varepsilon \rightarrow ( \quad )), ( \rightarrow \varepsilon \quad \text{int}, \varepsilon \rightarrow \varepsilon \]

\[ +, \varepsilon \rightarrow \varepsilon \]

\[ \times, \varepsilon \rightarrow \varepsilon \]
A PDA for arithmetic

```
int + ( ( int × int ) + int )
```
A PDA for arithmetic

Input

\[ \text{int} + ( ( \text{int} \times \text{int} ) + \text{int} ) \]
A PDA for arithmetic

Input

Stack

```
int + ( ( int × int ) + int )
```

```
ε, ε → ε
int, ε → ε
+, ε → ε
×, ε → ε
```

```
(, ε → ( )
), ( → ε
```

```
start
ε, ε → $
ε, $ → ε
```
A PDA for arithmetic

Input

\[
\text{int} + ( ( \text{int} \times \text{int} ) + \text{int} )
\]
A PDA for arithmetic

Input

\[ \text{int } + ( ( \text{int } \times \text{int}) + \text{int} ) \]

Stack

\[
\begin{align*}
\text{start} & \quad \varepsilon, \varepsilon \rightarrow \$ \\
(, \varepsilon & \rightarrow ( \quad \text{int}, \varepsilon \rightarrow \varepsilon \\
) & \rightarrow \varepsilon \\
\text{start} & \quad \varepsilon, \varepsilon \rightarrow \$ \\
+ & \rightarrow \varepsilon \\
\varepsilon & \rightarrow \varepsilon \\
\varepsilon, \$ & \rightarrow \varepsilon
\end{align*}
\]
A PDA for arithmetic

Input

\[ \text{int} + \left( ( \text{int} \times \text{int} ) + \text{int} \right) \]
A PDA for arithmetic

Input

\[ \text{int} + ( ( \text{int} \times \text{int} ) + \text{int} ) \]

Stack

\( ( ( \$ ) ) \)
A PDA for arithmetic

Input

\[ \text{int} + ( ( \text{int} \times \text{int} ) + \text{int} ) \]

Stack

[ ( ( $ ]
A PDA for arithmetic

Input

```
int + ( ( int × int ) + int )
```

Stack

```
( ( $
```

\[\begin{align*}
\text{start} & \rightarrow \varepsilon, \varepsilon \rightarrow \$
\end{align*}\]
A PDA for arithmetic

Input

int + ( ( int × int ) + int )

Stack

( ( $)
A PDA for arithmetic

Input

int + ( ( int × int ) + int )

Stack

( $
A PDA for arithmetic

Input

\[ \text{int + ( ( int } \times \text{ int ) + \text{ int } )} \]

Stack

( $}
A PDA for arithmetic

Input

\[
\text{int } + ( ( \text{int } \times \text{int} ) + \text{int} )
\]

Stack
A PDA for arithmetic

\[
\begin{align*}
\text{Input} & : \quad \text{int} + ( ( \text{int} \times \text{int} ) + \text{int} ) \\
\text{Stack} & : \quad \varepsilon, \varepsilon \rightarrow \$ \\
\end{align*}
\]
A PDA for arithmetic

Input

int + ( ( int × int ) + int )

Stack

Accept!
Shorthand for pushing multiple symbols to the stack

\[(r, \text{xyz}) \in \delta(q, a, s)\]

- \(q\) is the current state
- \(a\) is the next input symbol
- \(s\) is on the top of the stack

Do the following:
- Read \(a\)
- Pop \(s\)
- Push \(\text{xyz}\)
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