Fun
Exam 1 is due
Assignment 4 will be out today or tomorrow, due the Thursday after
break – a long time from now!

Fun in class
Context-free grammars – recursive structure allows more power
than regular expressions
Introduced pushdown automata; let’s play with them some more

Future fun
Proving the equivalence of CFGs and PDAs
Proving a language is or isn’t context-free
Exam 2 (for serious)

Semantic satiation
The psychological phenomenon where the repetition of a word or
phrase causes a listener to stop recognizing its meaning and
recognize it only as meaningless sounds.

Fun!
A pushdown automaton (PDA) is a finite automaton equipped with a stack-based memory.

Each transition in a PDA 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.

Example: Palindromes

\[ \Sigma = \{a, b, c\} \]

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

E.g.,
- c
- aca
- bcb
- abcba
- bbcbb
- ...

Example: Palindromes

\[ \epsilon, \epsilon \rightarrow \$ \]

\[ a, \epsilon \rightarrow A \]

\[ b, \epsilon \rightarrow B \]

\[ a, A \rightarrow \epsilon \]

\[ b, B \rightarrow \epsilon \]

\[ \epsilon, \$ \rightarrow \epsilon \]
Example: Palindromes

Input: a a a b c b a a a
Stack: $\text{a a a b c b a a a}$

Example: Palindromes

Input: a a a b c b a a a
Stack: $\text{a a a b c b a a a}$

Example: Palindromes

Input: a a a b c b a a a
Stack: $\text{a a a b c b a a a}$

Example: Palindromes

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

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

Example: Palindromes

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

Example: Palindromes

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

Example: Palindromes

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

Input: $a a a b c b a a a$
Stack: $A$

How can we describe the operation of a pushdown automaton for a given input?
(Other than a lot of slides...)
To know what a pushdown automaton can do at any point in its operation, we need to know its state, the input it can read, and the entire contents of its stack.

We call this combination an instantaneous description (ID), written in the form \((q, w, \alpha)\), where

- \(q\) = current state
- \(w\) = input characters that haven’t been read yet
- \(\alpha\) = stack contents, with the top on the left

**Describing the moves of the PDA**

If \(\delta(q, a, X)\) contains \((p, \alpha)\), then we say

\[(q, aw, X\beta) \rightarrow (p, w, \alpha \beta)\]

the “goes to” relation

We extend this relation to \(\rightarrow^*\) for 0 or more moves.

An input string \(w\) is accepted if \((q_0, w, \varepsilon) \rightarrow^* (p, \varepsilon, \varepsilon)\) for any accepting state \(p\).

\(L(P) = \text{set of strings accepted by } P\)

this is the “accept by empty stack” formulation

**Variations on a theme**

Finite automata are highly standardized.

For PDAs, there are many slightly different definitions, e.g.,

- is there an initial symbol on the stack or do we need to write $? 
- do we accept whenever we’ve read the input and are in an accept state or do we accept when we empty the stack? 
- can we push more than one symbol onto the stack in a single move?

These are all equivalent!

“A foolish consistency is the hobgoblin of little minds, adored by little statesmen and philosophers and divines. With consistency a great soul has simply nothing to do. He may as well concern himself with his shadow on the wall. Speak what you think now in hard words, and to-morrow speak what to-morrow thinks in hard words again, though it contradict every thing you said to-day.”

Ralph Waldo Emerson, Self-Reliance
Emerson continues,

“To be great is to be misunderstood.”

That’s…less desirable.

We can use whatever formulation of PDAs is most convenient, but try to note when we deviate from Sipser.

For $L = \{a^n b^n \mid n \geq 0\}$

$q_0 = \text{starting with empty stack, push }$ to mark bottom of stack

$q_1 = \text{reading } a\text{ and pushing } A\text{ onto the stack}$

$q_2 = \text{reading } b\text{ and popping } A\text{ until the } A\text{ s are all popped}$

$q_3 = \text{no more input and empty stack; accept}$

Exercise

Design the PDA to accept the language

$L = \{a^i b^j c^k \mid i = j + k \}$
A PDA for arithmetic

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

Consider the language $\text{ARITH} = \{w \in \Sigma^* \mid w \text{ is a legal arithmetic expression}\}$

E.g.,

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

Can we build a PDA for $\text{ARITH}$?
A PDA for arithmetic

A PDA for arithmetic

A PDA for arithmetic

A PDA for arithmetic
A PDA for arithmetic

\[ (, \varepsilon \rightarrow (, \varepsilon \rightarrow \varepsilon), ( \rightarrow \varepsilon) \]

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

```
\begin{align*}
&\text{Input} \quad \text{Stack} \\
&\text{int + int } \times \text{ int} \\
&\text{\uparrow} \\
\end{align*}
```

```
\begin{align*}
&\text{Input} \quad \text{Stack} \\
&\text{int + int } \times \text{ int} \\
&\text{\uparrow} \\
\end{align*}
```

```
\begin{align*}
&\text{Input} \quad \text{Stack} \\
&\text{int + int } \times \text{ int} \\
&\text{\uparrow} \\
\end{align*}
```

```
\begin{align*}
&\text{Input} \quad \text{Stack} \\
&\text{int + int } \times \text{ int} \\
&\text{\uparrow} \\
\end{align*}
```
A PDA for arithmetic

\[ (, \epsilon \rightarrow ( ) , \epsilon \rightarrow \epsilon ) \]
\[ \text{int, } \epsilon \rightarrow \epsilon \]
\[ +, \epsilon \rightarrow \epsilon \]
\[ \times, \epsilon \rightarrow \epsilon \]

Input: \( \text{int + int \times int} \)
Stack: \( \$ \)

Accept!
A PDA for arithmetic

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

Start: $, $ → ε, ε → $, ε, ε → ε → ε, ε, ε → ε → ε, ε, ε → ε → ε

Input: int + ( ( int × int ) + int )
Stack: ( ( $ ) )

A PDA for arithmetic

Start: $, $ → ε, ε → $, ε, ε → ε → ε, ε, ε → ε → ε, ε, ε → ε → ε

Input: int + ( ( int × int ) + int )
Stack: ( ( $ ) )
A PDA for arithmetic

Input
int + ( ( int \times int ) + int )

Stack
$ \uparrow$

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 PDA for arithmetic

Input
int + ( ( int \times int ) + int )

Stack
$ \uparrow$

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.

\[ q_0 \xrightarrow{0, 1} q_0, q_1 \xrightarrow{1} q_2 \]

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.

The PDAs we are dealing with are almost invariably nondeterministic.

But what about deterministic PDAs?

A *nondeterministic PDA (NPDA)* allows nondeterministic transitions (e.g., defines multiple possible moves for a given configuration)

Nondeterministic PDAs are strictly stronger then deterministic PDAs

In this respect, the situation is *not* similar to the situation of DFAs and NFAs

Nondeterministic PDAs are equivalent to CFGs
A **deterministic PDA (DPDA)** is a PDA with this extra property:

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’s **at most** one legal sequence of transitions that can be followed for any input.

Being deterministic doesn’t preclude ε-transitions – as long as there’s never a conflict between following the ε-transition or another transition.

There can be at most one ε-transition that could be followed at a time!

Being deterministic also doesn’t preclude a DPDA “dying” from having no transitions defined.

*Note:* Sipser’s formulation of DPDAs requires every transition be defined. For CMPU 240 we’ll allow them to die when the transition’s undefined – but we also won’t use DPDAs very much!

**Is this a DPDA?**

```
start  ε, ε → $  ε, $ → ε
0, $ → 0$  0, 0 → 00 1, 0 → ε
```

No – there’s a choice of what to do when there’s a 0 input and a $ on the stack.
Is this a DPDA? Yes!

This ε-transition is allowed because no other transitions in this state use the input symbol 0.

0, 0 → 00
1, 0 → ε

0, ε → ε

ε, $ → ε

ε, $ → $

This ε-transition is allowed because no other transitions in this state use the stack symbol $.

Why should we care about deterministic PDAs if they’re less powerful?

Parsers are DPDAs!

Thus, the question of what languages a DPDA can accept is really the question of what programming language syntax can be parsed conveniently.

The power of nondeterminism

An example of a context-free language that can be recognized by an NPDA but not by a DPDA is the language of all palindromes (without a dividing marker).

Intuition: Without nondeterminism, how would you know when you’ve read half the string?

However, it’s extremely difficult to actually prove that a given context-free language (CFL) is not a deterministic context-free language (DCFL)?

Unambiguous, deterministic CFGs are complicated and too restricted, so real parsers cheat by looking ahead one token.

Stay tuned: Learn about LL(1) and LR(1) grammars in CMPU 331. Some ambiguities (e.g., dangling else) are easily handled with one token lookahead.
Acknowledgments

This lecture incorporates material from:

- Nancy Ide
- Keith Schwarz
- Jeffrey Ullman
- Jennifer Walter