Pushdown Automata and Deterministic Parsing

Memory with a Stack

Computer Science Theory

Slide Set 6

Learning Outcomes

By the end of this slide set, you should be able to:

- define a PDA and interpret transition labels and instantaneous descriptions;
- distinguish final-state and empty-stack acceptance;
- design and trace PDAs for matched counts, delimiters, and palindromes;
- explain both directions of the CFG–NPDA equivalence theorem;
- state the restrictions defining a deterministic PDA;
- distinguish regular, deterministic context-free, and context-free languages; and
- perform a simple shift–reduce parse and identify parser conflicts.

Road Map

1. Pushdown Automata: Definition and Design
2. Equivalence of CFGs and NPDAs
3. Deterministic PDAs and DCFLs
4. Deterministic Parsing
5. Review and Exercises

Why Add a Stack?

A finite automaton has only a constant amount of memory. It cannot retain the unbounded exact count needed for a language such as

$$\{0^n 1^n \mid n \geq 0\}.$$

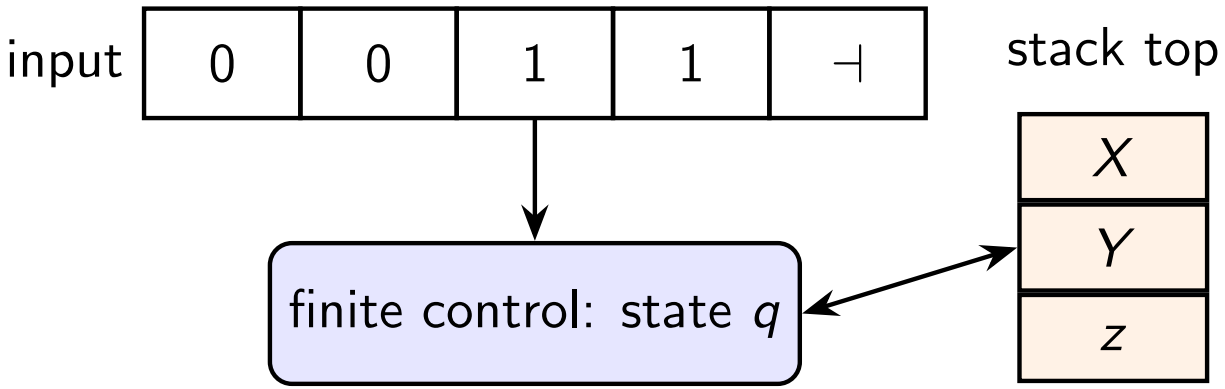
A stack provides unbounded but restricted memory:

- only the top symbol is directly accessible;
- symbols are removed in last-in, first-out order; and
- nesting and reversal naturally match stack behavior.

PDA intuition

A pushdown automaton is an NFA equipped with a stack. Its finite control chooses moves using the current state, the next input symbol (or ε), and the stack top.

PDA Architecture



In one move, a PDA may consume one input symbol or consume nothing, pop the top stack symbol, replace it by a finite string, and change state.

Formal Definition and Textbook Conventions

We use the Linz-style 7-tuple

$$M = (Q, \Sigma, \Gamma, \delta, q_0, z, F),$$

where

- Q, Σ, and Γ are finite state, input, and stack alphabets;
- $q_0 \in Q$ is the start state, $z \in \Gamma$ is the initial stack symbol, and $F \subseteq Q$;
- $$\delta : Q \times (\Sigma \cup \{\varepsilon\}) \times \Gamma \to \mathcal{P}_{\text{fin}}(Q \times \Gamma^*).$$

Sipser commonly uses a six-component convention with an initially empty stack and permits a move not to inspect a stack symbol. These conventions are computationally equivalent.

Convention used below

The stack top is written on the left, and every displayed transition pops one stack symbol before pushing its replacement.

Reading PDA Transition Labels

We abbreviate a move by

$$a, X \to \gamma.$$

It means: read a (or read nothing if $a = \varepsilon$), pop X, and push γ. With the top on the left, the first symbol of γ becomes the new top.

Label	Effect
$0, z \to 0z$	read 0 and place a marker above the bottom symbol
$0, 0 \to 00$	read 0 and add one more marker
$1, 0 \to \varepsilon$	read 1 and remove one marker
$\varepsilon, z \to z$	change state without consuming input or changing the stack

An NPDA may have several enabled moves and accepts if at least one computation branch accepts.

Instantaneous Descriptions

An instantaneous description (ID) is a triple

$$(q, w, u),$$

where q is the current state, w is the unread input, and u is the stack content with top on the left. If $(p, \gamma) \in \delta(q, a, X)$, then

$$(q, aw, X\alpha) \vdash (p, w, \gamma\alpha),$$

where $a \in \Sigma \cup \{\varepsilon\}$ and $aw = w$ when $a = \varepsilon$.

Reflexive transitive closure

$(q, w, u) \vdash^* (p, x, v)$ means that zero or more PDA moves transform the first ID into the second.

Two Acceptance Conventions

A computation starts in (q_0, w, z) and must consume the entire input.

Acceptance by final state

$$(q_0, w, z) \vdash^* (q_f, \varepsilon, u) \quad \text{for some } q_f \in F,\ u \in \Gamma^*.$$

Acceptance by empty stack

$$(q_0, w, z) \vdash^* (q, \varepsilon, \varepsilon) \quad \text{for some } q \in Q.$$

For NPDAs, both conventions define exactly the context-free languages.

Why the Two NPDA Conventions Are Equivalent

Final state $\Rightarrow$ empty stack

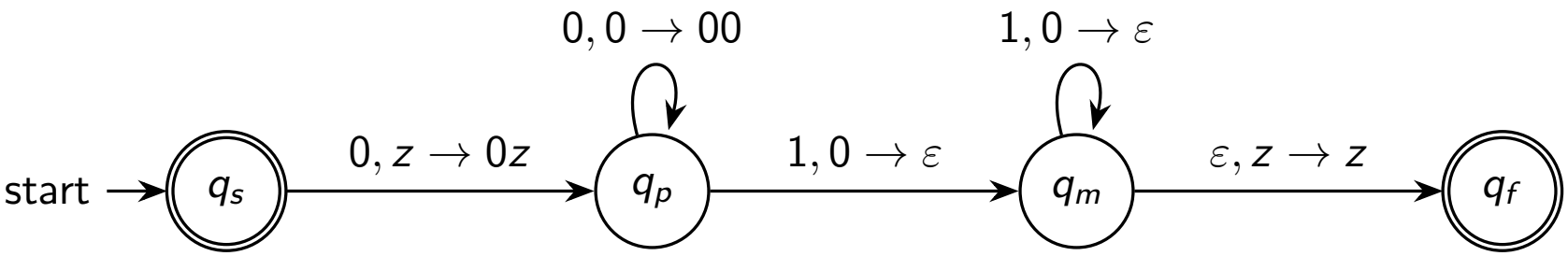
Add a fresh bottom marker and start state. Simulate the original NPDA; whenever it enters an old final state, nondeterministically move to a drain state that pops every remaining stack symbol.

Empty stack $\Rightarrow$ final state

Protect the simulated stack with a fresh bottom marker. When the simulated stack becomes empty, move to a new final state. Final-state acceptance still requires that all input has been consumed.

These constructions preserve the existence of an accepting branch. Later we will see why the same conversions do not preserve determinism in general.

Worked PDA: $\{0^n 1^n \mid n \geq 0\}$



Trace on 0011

$$(q_s, 0011, z) \vdash (q_p, 011, 0z) \vdash (q_p, 11, 00z)$$
$$\vdash (q_m, 1, 0z) \vdash (q_m, \varepsilon, z) \vdash (q_f, \varepsilon, z).$$

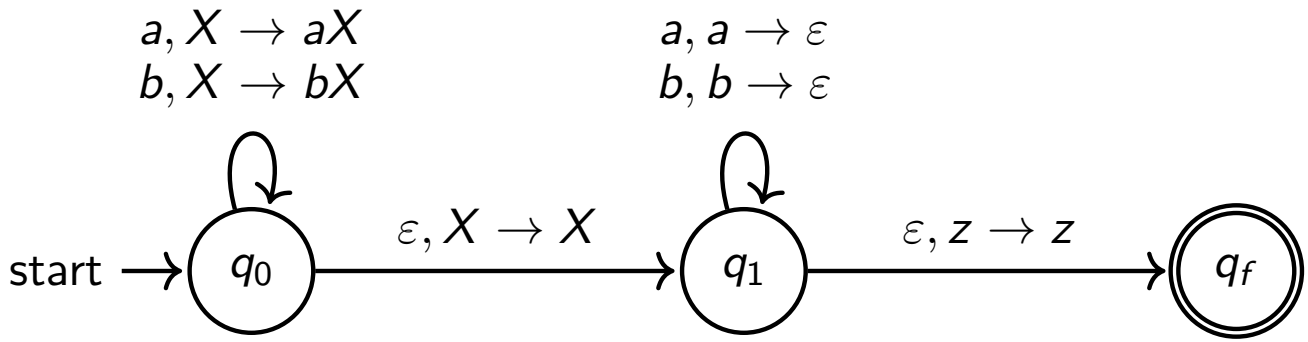
The states enforce the input order 0^*1^*; the stack enforces equal counts.

NPDA for Even-Length Palindromes

The language

$$\{ww^R \mid w \in \{a, b\}^*\}$$

requires the machine to guess the midpoint.



Here X ranges over $\Gamma = \{a, b, z\}$. One branch guesses the correct midpoint and accepts.

Tracing the Palindrome NPDA on *abba*

One accepting branch is

$$
\begin{aligned}
(q_0, abba, z) &\vdash (q_0, bba, az) && \text{push } a \\
&\vdash (q_0, ba, baz) && \text{push } b \\
&\vdash (q_1, ba, baz) && \text{guess the midpoint} \\
&\vdash (q_1, a, az) && \text{match and pop } b \\
&\vdash (q_1, \varepsilon, z) && \text{match and pop } a \\
&\vdash (q_f, \varepsilon, z). &&
\end{aligned}
$$

Guesses made too early or too late eventually block or leave unmatched input/stack symbols. To recognize odd-length palindromes, add midpoint transitions that consume one symbol without changing the stack.

A Deterministic Stack Pattern: Balanced Parentheses

A PDA can recognize balanced parentheses deterministically:

1. on input (, push a marker;
2. on input), pop one marker; reject if no marker is available;
3. at the endmarker, accept exactly when only the bottom marker remains.

Trace of stack contents on (()())

$$z \to (z \to ((z \to (z \to ((z \to (z \to z.$$

This is the machine analogue of the CFG $S \to (S)S \mid \varepsilon$.

The Equivalence Theorem

Theorem

A language is context-free iff some nondeterministic pushdown automaton recognizes it.

The proof has two constructive directions:

$$\text{CFG} \Longrightarrow \text{NPDA} \qquad \text{and} \qquad \text{NPDA} \Longrightarrow \text{CFG}.$$

- A CFG derivation can be simulated as nondeterministic stack expansion.
- A PDA's stack-balanced computations can be named by grammar variables.

Thus grammars and NPDAs are two views of the same language class: generation versus recognition.

CFG $\Rightarrow$ NPDA: Construction

Given a CFG $G = (V, T, S, P)$, build an NPDA that accepts by empty stack:

1. initialize the stack with S above a bottom marker;
2. if the top is a variable A, nondeterministically choose $A \to \beta$ and replace A by β;
3. if the top is terminal a and the next input symbol is a, read it and pop it;
4. when the input and simulated sentential form are exhausted, pop the bottom marker and accept by empty stack.

Invariant

At every stage, the stack above the bottom marker is a sentential form that could still derive the unread suffix of the input.

With our leftmost-stack convention, the symbols of β are pushed so its leftmost symbol is processed first.

Tracing the CFG Simulation

For $S \to 0S1 \mid \varepsilon$ on input 0011, show the unread input before the stack:

$$\begin{array}{ccl} 0011 & ; & Sz \\ 0011 & ; & 0S1z \\ 011 & ; & S1z \\ 011 & ; & 0S11z \\ 11 & ; & S11z \\ 11 & ; & 11z \\ 1 & ; & 1z \\ \varepsilon & ; & z \\ \varepsilon & ; & \varepsilon. \end{array}$$

Variable expansions are ε-moves; terminal matches consume input; the final move pops z. The accepting branch mirrors the derivation $S \Rightarrow 0S1 \Rightarrow 00S11 \Rightarrow 0011$.

NPDA $\Rightarrow$ CFG: State-Pair Variables

First normalize the NPDA so a transition pops one symbol and pushes either two symbols or none. Introduce variables

$$[pAq],$$

intended to generate the strings that take the PDA from state p with A on top to state q just after A has been removed, leaving the lower stack unchanged.

- A pop transition gives a terminal or ε production.
- A transition replacing A by BC gives productions that split the computation at an intermediate state where B has been removed and C remains.
- Start productions connect the initial state/stack symbol to possible accepting states.

The grammar is finite because Q and Γ are finite.

The Key NPDA-to-CFG Production

Suppose

$$(r, BC) \in \delta(p, a, A),$$

meaning that the PDA reads a, changes from p to r, and replaces A by BC. For every intermediate state s and destination state q, add

$$[pAq] \to a\,[rBs][sCq].$$

Interpretation

The first variable generates the input that removes B, ending in state s; the second generates the input that then removes C, ending in q.

If $(q, \varepsilon) \in \delta(p, a, A)$, add $[pAq] \to a$. These productions recursively describe every stack-balanced accepting computation.

What Makes a PDA Deterministic?

A PDA is deterministic when, for every state q and stack top X:

1. for each $a \in \Sigma \cup \{\varepsilon\}$, at most one move is available; and
2. if an ε-move is available, no input-consuming move is available for any next symbol.

Formally,

$$|\delta(q, a, X)| \leq 1,$$

and $\delta(q, \varepsilon, X) \neq \varnothing$ implies $\delta(q, a, X) = \varnothing$ for every $a \in \Sigma$.

Consequence

For a fixed input, a DPDA has at most one computation path. An NPDA may explore many incompatible stack strategies and accepts if one succeeds.

Known Midpoint Versus Guessed Midpoint

Compare two languages:

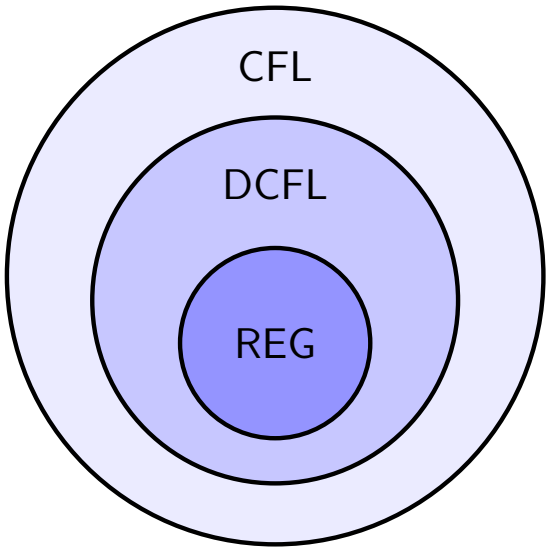
$$L_1 = \{wcw^R \mid w \in \{a, b\}^*\}, \qquad L_2 = \{ww^R \mid w \in \{a, b\}^*\}.$$

- L_1 is a DCFL: push before the marker c, then deterministically pop and compare after c.
- L_2 is context-free but not deterministic context-free; an NPDA guesses where the second half begins.

Proof caution

Failure of the obvious DPDA design is not itself a proof that L_2 is not a DCFL. The non-DCFL claim is a theorem established using deeper closure or structural arguments.

Language-Class Hierarchy



$$\mathrm{REG} \subsetneq \mathrm{DCFL} \subsetneq \mathrm{CFL}.$$

- $\{0^n 1^n\}$ is a nonregular DCFL.
- $\{ww^R\}$ is a CFL but not a DCFL.
- Every DCFL has an unambiguous CFG.
- The converse fails: the usual palindrome grammar is unambiguous, but its language is not a DCFL.

Important Closure Properties of DCFLs

DCFLs are closed under:

- complement, using a normalized endmarked DPDA; and
- intersection with a regular language, using a product with a DFA.

They are not closed under intersection or union. Let

$$L_1 = \{a^i b^j c^k \mid i = j\},$$
$$L_2 = \{a^i b^j c^k \mid j = k\}.$$

Here $i, j, k \geq 0$. Both are DCFLs, but

$$L_1 \cap L_2 = \{a^n b^n c^n \mid n \geq 0\},$$

which is not context-free. Nonclosure under union follows from complement closure and De Morgan's laws.

Acceptance Conventions Under Determinism

Final-state and empty-stack acceptance are equivalent for NPDAs but not interchangeable for DPDAs.

Why empty-stack DPDAs are restricted

If a DPDA accepts x by empty stack, it has no stack content with which to continue processing a proper extension xy. Deterministic empty-stack languages are therefore prefix-free.

General DCFLs are normally defined using final-state acceptance together with an explicit endmarker. The endmarker lets the machine distinguish:

- "the current prefix could be accepted if input ends now," from
- "the input has actually ended."

Why Parsing Looks Like a DPDA

A parser reads tokens left to right while its stack records unfinished syntactic structure.

- The finite control represents a parser state or set of viable grammar situations.
- The stack stores grammar symbols or states for nested constructs.
- The lookahead token and stack top determine the next deterministic action.

Endmarker

A fresh symbol such as $\dashv$ or \$ marks the actual end of the token stream and allows acceptance to be decided deterministically.

Top-Down and Bottom-Up Parsing

Top-down / LL-style	Bottom-up / LR-style
Starts from S and predicts productions.	Starts from the input and recognizes handles.
Constructs a leftmost derivation.	Constructs a rightmost derivation in reverse.
Must choose a production before seeing its full body.	Delays a decision until a right-hand side is recognized.
Simple and natural for hand-written parsers.	Handles a broader range of programming-language grammars.

Both approaches require grammar restrictions or parser tables that make each decision unique.

Handles and Rightmost Derivations

In a right-sentential form, a **handle** is an occurrence of a production right-hand side whose reduction reverses one step of a rightmost derivation. For

$$S \to aSb \mid ab,$$

a rightmost derivation of $aabb$ is

$$S \Rightarrow aSb \Rightarrow aabb.$$

Reversing it gives

$$aabb \Rightarrow aSb \Rightarrow S.$$

First the middle ab is the handle; after that reduction, the entire aSb is the handle.

Shift–Reduce Parser Actions

A bottom-up parser repeatedly chooses one action:

- **Shift:** move the next token onto the stack;
- **Reduce:** replace a recognized handle β by A when $A \to \beta$ is a production;
- **Accept:** the stack represents S and the lookahead is the endmarker;
- **Error:** no valid continuation exists.

An LR(k) parser scans left to right and uses at most k lookahead symbols to construct a rightmost derivation in reverse.

Worked Shift–Reduce Trace for *aabb*

Grammar: $S \to aSb \mid ab$.

Stack	Unread input	Action
ε	$aabb \dashv$	shift a
a	$abb \dashv$	shift a
aa	$bb \dashv$	shift b
aab	$b \dashv$	reduce $ab \to S$
aS	$b \dashv$	shift b
aSb	$\dashv$	reduce $aSb \to S$
S	$\dashv$	accept

Parser states—not only visible grammar symbols—normally determine when a reduction is valid.

Parser Conflicts

Shift–reduce conflict

The parser cannot choose uniquely between shifting the next token and reducing the current stack suffix.

Reduce–reduce conflict

Two different productions appear eligible for reduction.

Conflicts may indicate:

- an ambiguous grammar;
- insufficient lookahead for the chosen parsing method; or
- a grammar that needs restructuring or explicit precedence/associativity declarations.

An ambiguous grammar cannot be LR(k) for any fixed k, because a deterministic parser cannot choose between two genuine parse structures for the same input.

Checkpoint Exercises I: PDA Mechanics

1. Trace the $\{0^n 1^n\}$ PDA on 000111. Where does it block on 00111?
2. Give an accepting ID sequence for the palindrome NPDA on *baab*.
3. Modify the palindrome NPDA to accept odd-length palindromes as well.
4. Design a DPDA for $\{wcw^R \mid w \in \{a, b\}^*\}$.
5. Explain why final-state acceptance requires the input to be exhausted.
6. Convert, at proof-sketch level, an empty-stack NPDA to final-state acceptance.

Checkpoint Exercises II: Theory and Parsing

1. Place each language in the smallest class: regular, DCFL, CFL, or non-CFL.

$$0^*1^*, \quad \{0^n1^n\}, \quad \{ww^R\}, \quad \{a^nb^nc^n\}.$$

2. Why does an unambiguous CFG not necessarily define a DCFL?
3. Use the state-pair meaning to explain $[pAq]$ in one sentence.
4. Identify the two handles in the reduction $aabb \Rightarrow aSb \Rightarrow S$.
5. What is the difference between a shift–reduce conflict and a reduce–reduce conflict?
6. Why is an endmarker especially useful for deterministic acceptance?

Hints and Short Answers I

- On 00111, the stack reaches z while one 1 remains, so no accepting complete-input ID exists.
- For $baab$, push b, a, guess the midpoint, then pop on a, b.
- Add transitions from the push phase to the match phase that consume one a or b without changing the stack.
- For wcw^R, push before c, change phase on c, and then pop matching symbols.
- A final state reached with unread input is not an accepting ID under the language definition.
- Protect the simulated stack with a fresh bottom marker and enter a new final state when that marker is exposed.

Hints and Short Answers II

- The four classes are respectively regular; DCFL but nonregular; CFL but not DCFL; and non-CFL.
- Unambiguity is necessary for deterministic context-freeness but not sufficient; palindromes provide a standard example.
- $[pAq]$ generates strings that remove A while taking the PDA from state p to state q without disturbing the lower stack.
- The handles are the middle ab, followed by the entire aSb.
- Shift–reduce offers one shift and one reduction; reduce–reduce offers two reductions.
- The endmarker distinguishes a completed input from a prefix that might receive more symbols.

Common Mistakes to Avoid

- Forgetting which end of the written stack is the top.
- Treating an ε-move as if it consumed an input symbol.
- Accepting merely because a final state is reached while input remains.
- Assuming all nondeterministic branches must accept; one accepting branch is sufficient.
- Assuming final-state and empty-stack acceptance remain equivalent for DPDAs.
- Claiming a language is non-DCFL only because one deterministic design attempt failed.
- Reducing the wrong substring during a shift–reduce trace instead of the actual handle.

Summary

- A PDA combines finite control with an unbounded LIFO stack.
- NPDAs accept by the existence of a successful computation branch.
- Final-state and empty-stack acceptance are equivalent for NPDAs.
- CFGs and NPDAs define exactly the context-free languages.
- Determinism is strictly weaker for PDAs: $\mathrm{REG} \subsetneq \mathrm{DCFL} \subsetneq \mathrm{CFL}$.
- Endmarkers and deterministic stacks connect DPDAs to practical LL/LR parsing.
- Shift–reduce parsing reconstructs a rightmost derivation in reverse.

References and Suggested Reading

Peter Linz and Susan H. Rodger,

An Introduction to Formal Languages and Automata, 7th ed.,

Jones & Bartlett Learning, 2023.

Michael Sipser,

Introduction to the Theory of Computation, 3rd ed.,

Cengage Learning, 2013.

Recommended emphasis: Linz for PDA instantaneous descriptions, acceptance conventions, DPDAs, and parsing; Sipser for the CFG–PDA equivalence constructions and nondeterministic computation viewpoint.