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;
- justify a PDA design using an invariant and test boundary cases;
- 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.

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.

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

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

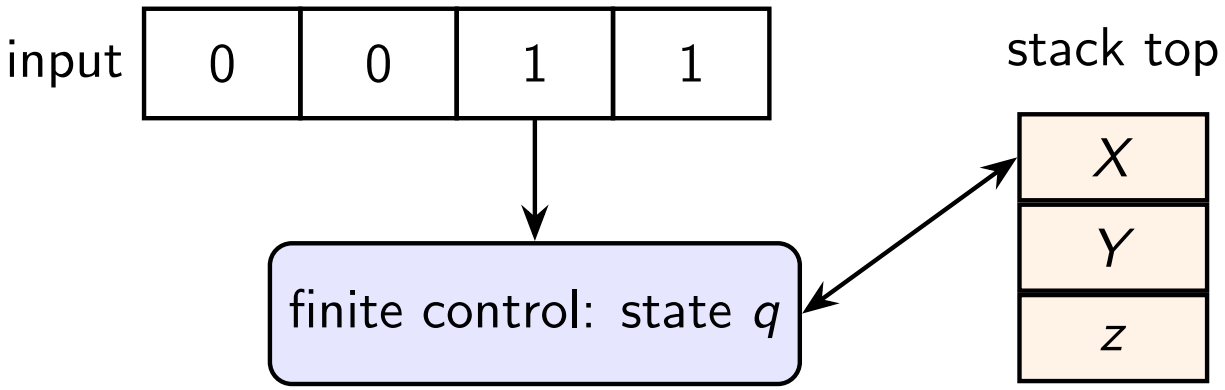
A stack provides unbounded but restricted memory:

- 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

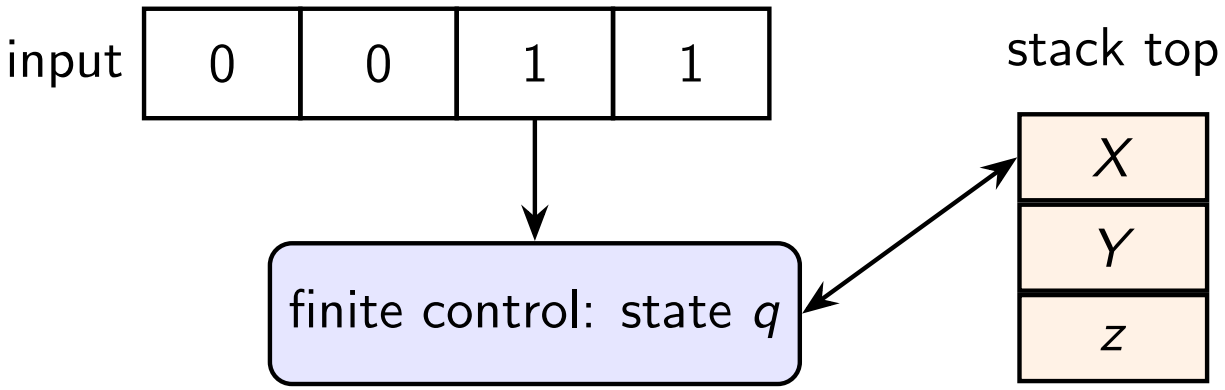
A pushdown automaton is an NFA equipped with a stack. Its finite control chooses moves using the current state, the next input symbol (or ε), and the stack top.

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. The input head moves only to the right when a symbol is consumed. The stack is not random-access memory: even reading below the top requires first removing what lies above it.

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. The input head moves only to the right when a symbol is consumed. The stack is not random-access memory: even reading below the top requires first removing what lies above it.

Formal Definition and Textbook Conventions

We use the Linz-style 7-tuple

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

where

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

Here $\mathcal{P}_{\text{fin}}$ means finite subsets, so the machine has finitely many transition alternatives. ε is not an alphabet symbol. 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.

An empty stack enables no further moves in this convention.

Formal Definition and Textbook Conventions

We use the Linz-style 7-tuple

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

where

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

Here $\mathcal{P}_{\text{fin}}$ means finite subsets, so the machine has finitely many transition alternatives. ε is not an alphabet symbol. 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.

An empty stack enables no further moves in this convention.

Formal Definition and Textbook Conventions

We use the Linz-style 7-tuple

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

where

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

Here $\mathcal{P}_{\text{fin}}$ means finite subsets, so the machine has finitely many transition alternatives. ε is not an alphabet symbol. 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.

An empty stack enables no further moves in this convention.

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. Unlisted transitions mean $\varnothing$ (no move), not an implicit self-loop.

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. Unlisted transitions mean $\varnothing$ (no move), not an implicit self-loop.

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.

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.

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. This means equivalent machines can be constructed, not that the two tests give the same language for an unchanged machine.

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. This means equivalent machines can be constructed, not that the two tests give the same language for an unchanged machine.

Acceptance, Blocking, and Infinite Branches

Acceptance means that a *finite* successful computation exists. A blocked branch is unsuccessful unless its current ID already satisfies the chosen acceptance condition.

Three different situations

- (q_f, ε, Xz) accepts by final state, but not by empty stack.
- (q, ε, z) is not an empty stack: z is a real symbol.
- (q_f, a, ε) does not accept: input a remains, and our convention enables no move without a stack top.

An ε-cycle such as $\delta(q, \varepsilon, z) = \{(q, z)\}$ can produce an infinite branch. It does not consume input or itself establish acceptance.

Do not confuse recognition with a terminating simulation

One infinite branch does not invalidate another accepting branch. Conversely, repeatedly exploring one branch may never settle membership. CFL membership is decidable using a CFG and a terminating parser such as CYK.

Acceptance, Blocking, and Infinite Branches

Acceptance means that a *finite* successful computation exists. A blocked branch is unsuccessful unless its current ID already satisfies the chosen acceptance condition.

Three different situations

- (q_f, ε, Xz) accepts by final state, but not by empty stack.
- (q, ε, z) is not an empty stack: z is a real symbol.
- (q_f, a, ε) does not accept: input a remains, and our convention enables no move without a stack top.

An ε-cycle such as $\delta(q, \varepsilon, z) = \{(q, z)\}$ can produce an infinite branch. It does not consume input or itself establish acceptance.

Do not confuse recognition with a terminating simulation

One infinite branch does not invalidate another accepting branch. Conversely, repeatedly exploring one branch may never settle membership. CFL membership is decidable using a CFG and a terminating parser such as CYK.

A PDA Design Checklist

1. Specify the alphabet, acceptance mode, and any explicit endmarker.
2. Decide what each stack symbol records: a count, delimiter, or symbol awaiting a reversed match.
3. Use states for phases; the stack alone need not enforce input order.
4. Write an invariant relating the consumed prefix, state, and stack.
5. List moves for *every relevant stack top*, including the bottom marker. Leave invalid cases without a move.
6. Test ε, the shortest valid word, a mismatch, and a valid word with an invalid extension; then prove both inclusions.

What nondeterminism adds

A branch can guess a phase boundary or grammar production and then verify it. The guess cannot inspect the future, and an incorrect branch must never accept. Deterministic designs need an observable basis for each choice, such as a delimiter or the first symbol of a new block.

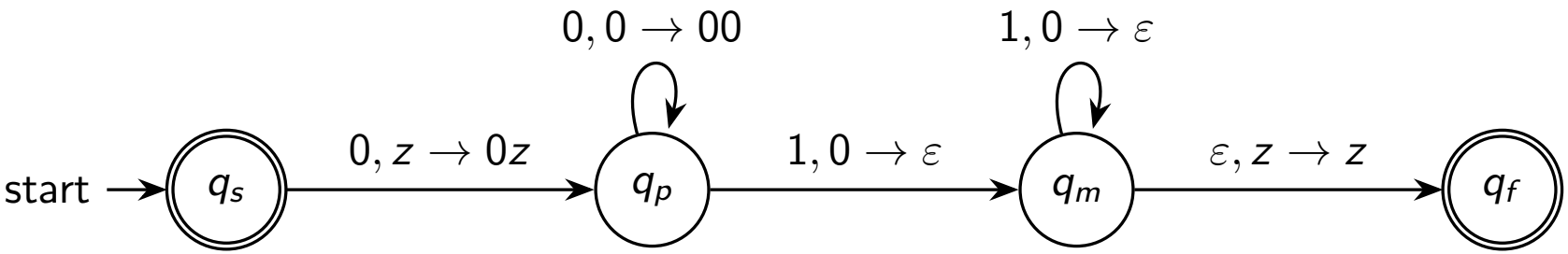
A PDA Design Checklist

1. Specify the alphabet, acceptance mode, and any explicit endmarker.
2. Decide what each stack symbol records: a count, delimiter, or symbol awaiting a reversed match.
3. Use states for phases; the stack alone need not enforce input order.
4. Write an invariant relating the consumed prefix, state, and stack.
5. List moves for *every relevant stack top*, including the bottom marker. Leave invalid cases without a move.
6. Test ε, the shortest valid word, a mismatch, and a valid word with an invalid extension; then prove both inclusions.

What nondeterminism adds

A branch can guess a phase boundary or grammar production and then verify it. The guess cannot inspect the future, and an incorrect branch must never accept. Deterministic designs need an observable basis for each choice, such as a delimiter or the first symbol of a new block.

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

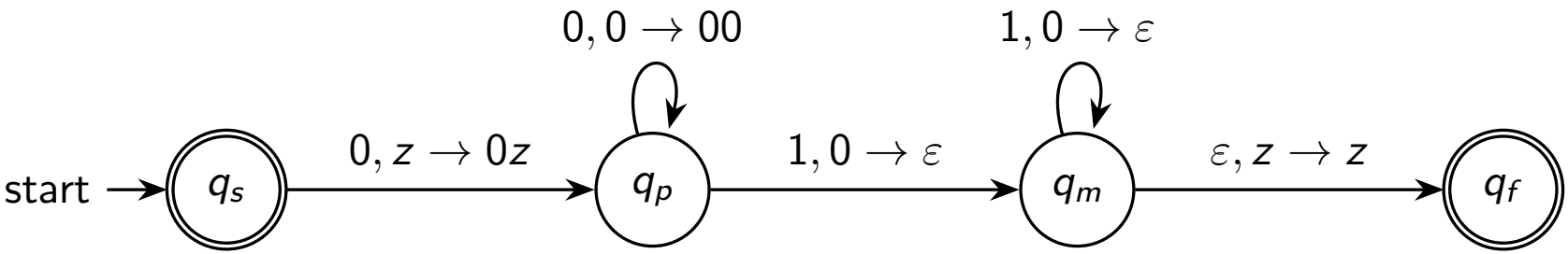


Trace on 0011

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

The states enforce the input order 0^*1^*; the stack enforces equal counts. State q_s accepts ε; q_f has no outgoing moves.

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. State q_s accepts ε; q_f has no outgoing moves.

Why the Matched-Count PDA Is Correct

The control state records the phase; the stack records the difference between the counts read so far.

Invariant along every nonblocked run

In q_p, the consumed prefix is 0^k and the stack is $0^k z$, where $k \geq 1$. In q_m, the prefix is $0^k 1^j$ and the stack is $0^{k-j} z$, where $1 \leq j \leq k$. Each displayed transition preserves these statements.

Soundness. The initial final state accepts only ε. To reach q_f, q_m must expose z, so $j = k$. Acceptance additionally requires exhausted input. Thus only $0^k 1^k$ can be accepted.

Completeness. For any $k \geq 1$, push exactly k markers on the zeros, pop them on the k ones, then enter q_f. The case $k = 0$ accepts in q_s without a move.

Boundary checks

001 leaves a marker; 00111 leaves input even after reaching q_f; 0101 cannot return to the push phase. All three are rejected.

Why the Matched-Count PDA Is Correct

The control state records the phase; the stack records the difference between the counts read so far.

Invariant along every nonblocked run

In q_p, the consumed prefix is 0^k and the stack is $0^k z$, where $k \geq 1$. In q_m, the prefix is $0^k 1^j$ and the stack is $0^{k-j} z$, where $1 \leq j \leq k$. Each displayed transition preserves these statements.

Soundness. The initial final state accepts only ε. To reach q_f, q_m must expose z, so $j = k$. Acceptance additionally requires exhausted input. Thus only $0^k 1^k$ can be accepted.

Completeness. For any $k \geq 1$, push exactly k markers on the zeros, pop them on the k ones, then enter q_f. The case $k = 0$ accepts in q_s without a move.

Boundary checks

001 leaves a marker; 00111 leaves input even after reaching q_f; 0101 cannot return to the push phase. All three are rejected.

Why the Matched-Count PDA Is Correct

The control state records the phase; the stack records the difference between the counts read so far.

Invariant along every nonblocked run

In q_p, the consumed prefix is 0^k and the stack is $0^k z$, where $k \geq 1$. In q_m, the prefix is $0^k 1^j$ and the stack is $0^{k-j} z$, where $1 \leq j \leq k$. Each displayed transition preserves these statements.

Soundness. The initial final state accepts only ε. To reach q_f, q_m must expose z, so $j = k$. Acceptance additionally requires exhausted input. Thus only $0^k 1^k$ can be accepted.

Completeness. For any $k \geq 1$, push exactly k markers on the zeros, pop them on the k ones, then enter q_f. The case $k = 0$ accepts in q_s without a move.

Boundary checks

001 leaves a marker; 00111 leaves input even after reaching q_f; 0101 cannot return to the push phase. All three are rejected.

NPDA for Even-Length Palindromes

The language

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

is recognized by guessing the midpoint and then checking the guess.

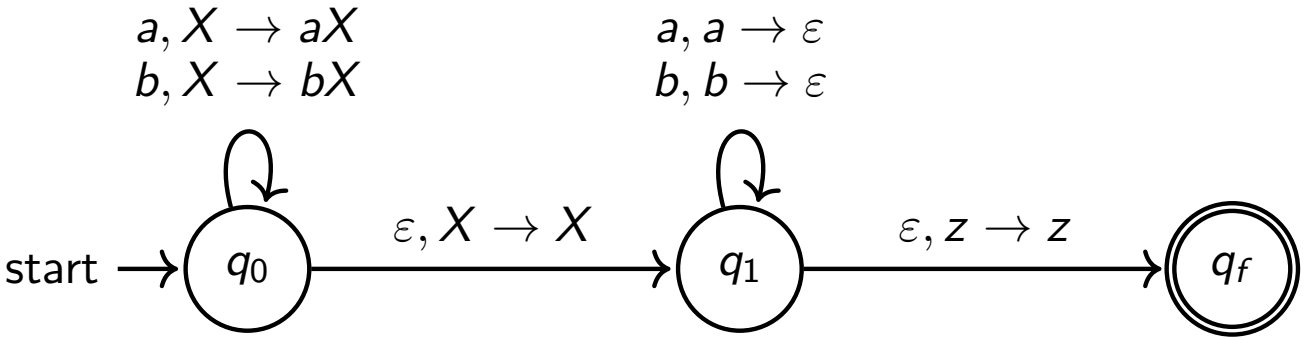
Here X ranges over $\Gamma = \{a, b, z\}$; $z \notin \{a, b\}$. For each word in the language, a branch guesses the midpoint and accepts. A nonmember has no accepting branch.

NPDA for Even-Length Palindromes

The language

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

is recognized by guessing the midpoint and then checking the guess.



Here X ranges over $\Gamma = \{a, b, z\}$; $z \notin \{a, b\}$. For each word in the language, a branch guesses the midpoint and accepts. A nonmember has no accepting branch.

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 accept both parities, add midpoint moves consuming one symbol without changing the stack. For odd lengths only, replace the empty midpoint moves.

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 accept both parities, add midpoint moves consuming one symbol without changing the stack. For odd lengths only, replace the empty midpoint moves.

A Deterministic Stack Pattern: Balanced Parentheses

Here we explicitly supply $w \dashv$, where $\dashv \notin \{(,)\}$ is a fresh right endmarker. It is a real input symbol, unlike ε. 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 on (()()), with X for an unmatched opener

$$z \to Xz \to XXz \to Xz \to XXz \to Xz \to z.$$

In a work state q, use $(, A \to XA$ for $A \in \{X, z\}$, $), X \to \varepsilon$, and $\dashv, z \to z$ to a final state. No other moves are enabled. The bottom marker cannot match a closer. This is the machine analogue of the CFG $S \to (S)S \mid \varepsilon$.

A Deterministic Stack Pattern: Balanced Parentheses

Here we explicitly supply $w\,\dashv$, where $\dashv \notin \{(,)\}$ is a fresh right endmarker. It is a real input symbol, unlike ε. 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 on (()()), with X for an unmatched opener

$$z \to Xz \to XXz \to Xz \to XXz \to Xz \to z.$$

In a work state q, use $(, A \to XA$ for $A \in \{X, z\}$, $), X \to \varepsilon$, and $\dashv, z \to z$ to a final state. No other moves are enabled. The bottom marker cannot match a closer. 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.

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. whenever only the bottom marker remains, allow it to be popped. Empty-stack acceptance additionally requires exhausted input.

Invariant

If u is the consumed prefix and α is the stack above the bottom marker, then $S \Rightarrow^* u\alpha$ by leftmost expansion. It is $u\alpha$, not necessarily α alone, that is a sentential form.

A bad choice may leave α unable to derive the unread suffix. Only one successful branch is needed. With top on the left, replacing A by β puts its leftmost symbol on top.

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. whenever only the bottom marker remains, allow it to be popped. Empty-stack acceptance additionally requires exhausted input.

Invariant

If u is the consumed prefix and α is the stack above the bottom marker, then $S \Rightarrow^* u\alpha$ by leftmost expansion. It is $u\alpha$, not necessarily α alone, that is a sentential form.

A bad choice may leave α unable to derive the unread suffix. Only one successful branch is needed. With top on the left, replacing A by β puts its leftmost symbol on top.

Tracing the CFG Simulation

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

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

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

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 ⇒ CFG: The Core Idea

First convert the NPDA to empty-stack acceptance. The difficulty is that its stack can grow without bound; we cannot make one variable per full stack.

Name a stack-removal task instead

The variable $[pAq]$ describes input consumed while starting in state p with top A, removing A and everything pushed above it, and first exposing the lower stack in state q. The lower stack is untouched.

If a move replaces A by BC, then all of B's work must finish before C's work begins. A grammar production joins those two tasks, guessing the intermediate state. There are only finitely many state/symbol triples.

$$S \to [q_0 z q] \quad \text{for every } q \in Q.$$

These alternatives describe exactly the computations that remove the initial stack. Induction relates derivation trees to nested stack removals. The appendix supplies the precise productions and a worked conversion.

NPDA ⇒ CFG: The Core Idea

First convert the NPDA to empty-stack acceptance. The difficulty is that its stack can grow without bound; we cannot make one variable per full stack.

Name a stack-removal task instead

The variable $[pAq]$ describes input consumed while starting in state p with top A, removing A and everything pushed above it, and first exposing the lower stack in state q. The lower stack is untouched.

If a move replaces A by BC, then all of B's work must finish before C's work begins. A grammar production joins those two tasks, guessing the intermediate state. There are only finitely many state/symbol triples.

$$S \to [q_0 z q] \quad \text{for every } q \in Q.$$

These alternatives describe exactly the computations that remove the initial stack. Induction relates derivation trees to nested stack removals. The appendix supplies the precise productions and a worked conversion.

NPDA $\Rightarrow$ CFG: The Core Idea

First convert the NPDA to empty-stack acceptance. The difficulty is that its stack can grow without bound; we cannot make one variable per full stack.

Name a stack-removal task instead

The variable $[pAq]$ describes input consumed while starting in state p with top A, removing A and everything pushed above it, and first exposing the lower stack in state q. The lower stack is untouched.

If a move replaces A by BC, then all of B's work must finish before C's work begins. A grammar production joins those two tasks, guessing the intermediate state. There are only finitely many state/symbol triples.

$$S \to [q_0 z q] \quad \text{for every } q \in Q.$$

These alternatives describe exactly the computations that remove the initial stack. Induction relates derivation trees to nested stack removals. The appendix supplies the precise productions and a worked conversion.

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.

A **DCFL** is a language accepted by a DPDA by final state. This definition does not require an endmarker; marked inputs will be identified explicitly. Determinism does not guarantee halting.

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.

A **DCFL** is a language accepted by a DPDA by final state. This definition does not require an endmarker; marked inputs will be identified explicitly. Determinism does not guarantee halting.

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.

A **DCFL** is a language accepted by a DPDA by final state. This definition does not require an endmarker; marked inputs will be identified explicitly. Determinism does not guarantee halting.

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.

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.

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.

Worked DPDA: A Marked Midpoint

For $L = \{wcw^R : w \in \{a, b\}^*\}$ use states p (push), m (match), and final state f. Initially the state is p and the stack is z. With $x \in \{a, b\}$ and $X \in \{a, b, z\}$, the complete move list is

$$\delta(p, x, X) = \{(p, xX)\}, \quad \delta(p, c, X) = \{(m, X)\},$$
$$\delta(m, x, x) = \{(m, \varepsilon)\}, \quad \delta(m, \varepsilon, z) = \{(f, z)\}.$$

All other entries are empty; f has no outgoing moves. No endmarker is used.

Why it works, and why it is deterministic

After reading w before c, the stack is $w^R z$. After c, each input symbol must match the top, so acceptance requires exactly w^R. The only empty move is at (m, z), where no consuming move is enabled. There is never a choice between pushing and matching.

Tests: c and $abcba$ accept; ε, $abca$, and cc reject. On cc the run may reach f with unread c, which is not acceptance.

Worked DPDA: A Marked Midpoint

For $L = \{wcw^R : w \in \{a, b\}^*\}$ use states p (push), m (match), and final state f. Initially the state is p and the stack is z. With $x \in \{a, b\}$ and $X \in \{a, b, z\}$, the complete move list is

$$\delta(p, x, X) = \{(p, xX)\}, \quad \delta(p, c, X) = \{(m, X)\},$$
$$\delta(m, x, x) = \{(m, \varepsilon)\}, \quad \delta(m, \varepsilon, z) = \{(f, z)\}.$$

All other entries are empty; f has no outgoing moves. No endmarker is used.

Why it works, and why it is deterministic

After reading w before c, the stack is $w^R z$. After c, each input symbol must match the top, so acceptance requires exactly w^R. The only empty move is at (m, z), where no consuming move is enabled. There is never a choice between pushing and matching.

Tests: c and $abcba$ accept; ε, $abca$, and cc reject. On cc the run may reach f with unread c, which is not acceptance.

Worked DPDA: A Marked Midpoint

For $L = \{wcw^R : w \in \{a, b\}^*\}$ use states p (push), m (match), and final state f. Initially the state is p and the stack is z. With $x \in \{a, b\}$ and $X \in \{a, b, z\}$, the complete move list is

$$\delta(p, x, X) = \{(p, xX)\}, \quad \delta(p, c, X) = \{(m, X)\},$$
$$\delta(m, x, x) = \{(m, \varepsilon)\}, \quad \delta(m, \varepsilon, z) = \{(f, z)\}.$$

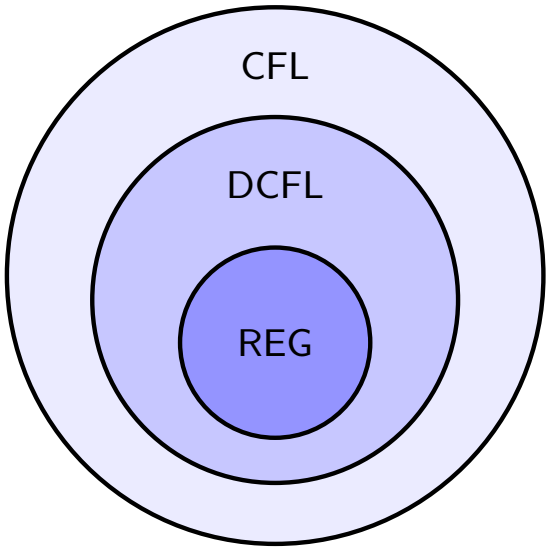
All other entries are empty; f has no outgoing moves. No endmarker is used.

Why it works, and why it is deterministic

After reading w before c, the stack is $w^R z$. After c, each input symbol must match the top, so acceptance requires exactly w^R. The only empty move is at (m, z), where no consuming move is enabled. There is never a choice between pushing and matching.

Tests: c and *abcba* accept; ε, *abca*, and *cc* reject. On *cc* the run may reach f with unread c, which is not acceptance.

Language-Class Hierarchy

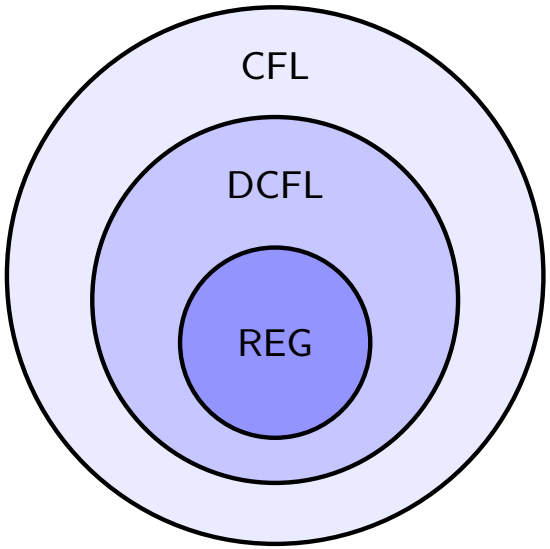


$$\text{REG} \subsetneq \text{DCFL} \subsetneq \text{CFL}.$$

- $\{0^n 1^n : n \geq 0\}$ is a nonregular DCFL.
- $\{ww^R : w \in \{a, b\}^*\}$ is a CFL but not a DCFL.
- Every DCFL has an unambiguous CFG.
- The converse fails: $S \to aSa \mid bSb \mid \varepsilon$ is unambiguous, but its language is not a DCFL.

To simulate any DFA, keep z unchanged on every move and copy the DFA's states, transitions, and final states. Thus every regular language is a DCFL.

Language-Class Hierarchy



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

- $\{0^n 1^n : n \geq 0\}$ is a nonregular DCFL.
- $\{ww^R : w \in \{a, b\}^*\}$ is a CFL but not a DCFL.
- Every DCFL has an unambiguous CFG.
- The converse fails: $S \to aSa \mid bSb \mid \varepsilon$ is unambiguous, but its language is not a DCFL.

To simulate any DFA, keep z unchanged on every move and copy the DFA's states, transitions, and final states. Thus every regular language is a DCFL.

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.

Counterexample: $\{a, aa\}$ is regular and hence a DCFL, but not prefix-free, so no unmarked empty-stack DPDA accepts it. With a fresh endmarker, we instead supply $w \dashv$ and recognize $L \dashv = \{w \dashv : w \in L\}$. A suitably arranged final-state DPDA can consume $\dashv$ at its acceptance decision and then drain the stack. The marker distinguishes end of input from a prefix that might continue.

Do not silently transfer a claim about $L \dashv$ to empty-stack acceptance of L.

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.

Counterexample: $\{a, aa\}$ is regular and hence a DCFL, but not prefix-free, so no unmarked empty-stack DPDA accepts it. With a fresh endmarker, we instead supply $w \dashv$ and recognize $L \dashv = \{w \dashv : w \in L\}$. A suitably arranged final-state DPDA can consume $\dashv$ at its acceptance decision and then drain the stack. The marker distinguishes end of input from a prefix that might continue.

Do not silently transfer a claim about $L \dashv$ to empty-stack acceptance of L.

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.
- A fixed-size lookahead buffer can be stored in finite control; together with the parser state, it determines the next action.

Endmarker

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

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.
- A fixed-size lookahead buffer can be stored in finite control; together with the parser state, it determines the next 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 reductions (write $\rightsquigarrow$ to distinguish them from grammar derivation steps):

$$aabb \rightsquigarrow aSb \rightsquigarrow S.$$

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

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 reductions (write $\rightsquigarrow$ to distinguish them from grammar derivation steps):

$$aabb \rightsquigarrow aSb \rightsquigarrow S.$$

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 handle β by A when $A \to \beta$ is a production, without consuming input;
- **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 is not LR(k) for any fixed k. Precedence rules can select one interpretation, but this adds a disambiguation policy; it does not make the original grammar unambiguous.

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 is not LR(k) for any fixed k. Precedence rules can select one interpretation, but this adds a disambiguation policy; it does not make the original grammar unambiguous.

Checkpoint Exercises I: PDA Mechanics

1. Trace the $\{0^n1^n\}$ 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, and explain the change needed for odd lengths only.
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. Use $n \geq 0$ and $w \in \{a, b\}^*$:

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

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 $aabb \rightsquigarrow aSb \rightsquigarrow S$.
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?

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.

Appendix Guide

The main sequence develops PDA mechanics, both equivalence directions, determinism, and the ideas behind predictive and shift–reduce parsing. The following material is optional enrichment, organized for self-study:

- checkpoint answers, followed by five solved construction challenges;
- acceptance conversions and exact CFG–NPDA proof details;
- DCFL closure and the complement-construction pitfall;
- bounded stacks, two-stack machines, and visibly pushdown automata;
- a predictive parse, concrete conflicts, and a complete LR table.

Suggested use

Attempt each challenge before revealing its solution. The main lecture retains both equivalence directions at proof-sketch level; the detailed conversions and parser machinery can be assigned as follow-up reading. Exercise sources and editions are identified explicitly below.

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 consuming one a or b without changing the stack. Retain the empty midpoint moves for both parities; remove them for odd lengths only.
- 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.

Challenge Sources and How to Use the Solutions

The prompts below are paraphrased; the worked solutions and explanations are written for this course, not reproduced from a solutions manual.

Challenge	Source / status
1. Choose a count comparison	Sipser, 2nd ed., Exercise 2.10 (language from 2.9)
2. Cancel opposite symbols	Sipser, 2nd ed., Exercise 2.7(a), referring to 2.6(a)
3. Find a reversed substring	Sipser, 2nd ed., Exercise 2.7(c), referring to 2.6(c)
4. Combine a PDA and a DFA	Sipser, 2nd ed., Problem 2.18(a)
5. Split a count across blocks	Additional course practice; no textbook exercise number claimed

[S06] supplies these edition-specific references. Linz's PDA/ID viewpoint guides the phase descriptions and invariants.

Before reading an answer

Choose the acceptance convention, state the stack invariant, and test the empty and zero-count cases. Then justify both inclusions.

Challenge 1: Guess Which Equality to Check

Task: build an NPDA for

$$L = \{a^i b^j c^k : i = j \text{ or } j = k, \quad i, j, k \geq 0\}.$$

Solution: make one initial nondeterministic choice

Both branches enforce the order $a^* b^* c^*$ and consume all input.

- **Check $i = j$:** push a marker per a, pop one per b, and permit the c-phase only after all markers are removed. Once in that phase, read any number of c's.
- **Check $j = k$:** read the initial a's without changing the stack, push on b, and pop on c.

Use a protected bottom marker. A branch accepts only in an appropriate final state with no unmatched markers; phases may have length zero.

Why it works: a successful branch certifies one equality. Conversely, if either equality holds, its branch can finish successfully.

Checks: *aabbc* uses the first branch; *abbcc* uses the second; *abbc* fails both; ε succeeds in both. Several accepting branches are permitted—acceptance means logical OR.

Challenge 1: Guess Which Equality to Check

Task: build an NPDA for

$$L = \{a^i b^j c^k : i = j \text{ or } j = k, \quad i, j, k \geq 0\}.$$

Solution: make one initial nondeterministic choice

Both branches enforce the order $a^* b^* c^*$ and consume all input.

- **Check $i = j$:** push a marker per a, pop one per b, and permit the c-phase only after all markers are removed. Once in that phase, read any number of c's.
- **Check $j = k$:** read the initial a's without changing the stack, push on b, and pop on c.

Use a protected bottom marker. A branch accepts only in an appropriate final state with no unmatched markers; phases may have length zero.

Why it works: a successful branch certifies one equality. Conversely, if either equality holds, its branch can finish successfully.

Checks: *aabbc* uses the first branch; *abbcc* uses the second; *abbc* fails both; ε succeeds in both. Several accepting branches are permitted—acceptance means logical OR.

Challenge 1: Guess Which Equality to Check

Task: build an NPDA for

$$L = \{a^i b^j c^k : i = j \text{ or } j = k, \quad i, j, k \geq 0\}.$$

Solution: make one initial nondeterministic choice

Both branches enforce the order $a^* b^* c^*$ and consume all input.

- **Check $i = j$:** push a marker per a, pop one per b, and permit the c-phase only after all markers are removed. Once in that phase, read any number of c's.
- **Check $j = k$:** read the initial a's without changing the stack, push on b, and pop on c.

Use a protected bottom marker. A branch accepts only in an appropriate final state with no unmatched markers; phases may have length zero.

Why it works: a successful branch certifies one equality. Conversely, if either equality holds, its branch can finish successfully.

Checks: *aabbc* uses the first branch; *abbcc* uses the second; *abbc* fails both; ε succeeds in both. Several accepting branches are permitted—acceptance means logical OR.

Challenge 2: More a's Than b's, in Any Order

Task: recognize $L = \{w \in \{a, b\}^* : |w|_a > |w|_b\}$. A push-then-pop design is insufficient: the symbols may interleave. **Solution:** in a work state q, use the following replacements. An entry γ means $\delta(q, \text{input}, X) = \{(q, \gamma)\}$.

input\top X	z	A	B
a	Az	AA	ε
b	Bz	ε	BB

Supply a fresh endmarker $\dashv$. The only move on it is $\delta(q, \dashv, A) = \{(f, A)\}$, where f is final and has no outgoing moves. This DPDA accepts exactly $L\dashv$.

Invariant: cancel opposite symbols

For the consumed prefix u, let $d = |u|_a - |u|_b$. The stack is $A^d z$ if $d > 0$, $B^{-d} z$ if $d < 0$, and z if $d = 0$. Each table entry updates this signed difference by $+1$ or -1.

At the end, top A is equivalent to $d > 0$. Thus *ababa* succeeds, while *abbab* and ε fail. One stack records a difference, not two independent unbounded counters.

Challenge 2: More a's Than b's, in Any Order

Task: recognize $L = \{w \in \{a, b\}^* : |w|_a > |w|_b\}$. A push-then-pop design is insufficient: the symbols may interleave. **Solution:** in a work state q, use the following replacements. An entry γ means $\delta(q, \text{input}, X) = \{(q, \gamma)\}$.

input\top X	z	A	B
a	Az	AA	ε
b	Bz	ε	BB

Supply a fresh endmarker $\dashv$. The only move on it is $\delta(q, \dashv, A) = \{(f, A)\}$, where f is final and has no outgoing moves. This DPDA accepts exactly $L\,\dashv$.

Invariant: cancel opposite symbols

For the consumed prefix u, let $d = |u|_a - |u|_b$. The stack is $A^d z$ if $d > 0$, $B^{-d} z$ if $d < 0$, and z if $d = 0$. Each table entry updates this signed difference by $+1$ or -1.

At the end, top A is equivalent to $d > 0$. Thus *ababa* succeeds, while *abbab* and ε fail. One stack records a difference, not two independent unbounded counters.

Challenge 2: More a's Than b's, in Any Order

Task: recognize $L = \{w \in \{a, b\}^* : |w|_a > |w|_b\}$. A push-then-pop design is insufficient: the symbols may interleave. **Solution:** in a work state q, use the following replacements. An entry γ means $\delta(q, \text{input}, X) = \{(q, \gamma)\}$.

input\top X	z	A	B
a	Az	AA	ε
b	Bz	ε	BB

Supply a fresh endmarker $\dashv$. The only move on it is $\delta(q, \dashv, A) = \{(f, A)\}$, where f is final and has no outgoing moves. This DPDA accepts exactly $L\dashv$.

Invariant: cancel opposite symbols

For the consumed prefix u, let $d = |u|_a - |u|_b$. The stack is $A^d z$ if $d > 0$, $B^{-d} z$ if $d < 0$, and z if $d = 0$. Each table entry updates this signed difference by $+1$ or -1.

At the end, top A is equivalent to $d > 0$. Thus *ababa* succeeds, while *abbab* and ε fail. One stack records a difference, not two independent unbounded counters.

Challenge 3: Guess Where a Reversed Word Occurs

Task: build an NPDA for

$$L = \{w\#x : w, x \in \{0,1\}^*,\ w^R \text{ is a substring of } x\}.$$

1. **Store:** push each bit of w; on the unique $\#$, enter the skip phase. The stack is now $w^R z$.
2. **Skip:** read any prefix of x without changing the stack; use an ε-move to guess the match's starting position.
3. **Match:** consume a bit only when it equals the top bit, then pop. There is no skipping within the match.
4. **Finish:** when z is exposed, enter a final tail state that reads any remaining bits without changing z.

A successful branch splits $x = u\, w^R\, v$; conversely, any such split supplies a successful guess. No phase after the first permits $\#$.

Trace idea: $10\#0011$**.** After $\#$ the stack is $01z$. Skip 0, match and pop 01, then read the last 1: $0011 = 0 \cdot 01 \cdot 1$.

For $w = \varepsilon$, skip $\to$ match $\to$ finish immediately: the empty word occurs in every x.

Challenge 3: Guess Where a Reversed Word Occurs

Task: build an NPDA for

$$L = \{w\#x : w, x \in \{0,1\}^*,\ w^R \text{ is a substring of } x\}.$$

1. **Store:** push each bit of w; on the unique #, enter the skip phase. The stack is now $w^R z$.
2. **Skip:** read any prefix of x without changing the stack; use an ε-move to guess the match's starting position.
3. **Match:** consume a bit only when it equals the top bit, then pop. There is no skipping within the match.
4. **Finish:** when z is exposed, enter a final tail state that reads any remaining bits without changing z.

A successful branch splits $x = u\, w^R\, v$; conversely, any such split supplies a successful guess. No phase after the first permits #.

Trace idea: 10#0011. After # the stack is $01z$. Skip 0, match and pop 01, then read the last 1: $0011 = 0 \cdot 01 \cdot 1$.

For $w = \varepsilon$, skip $\to$ match $\to$ finish immediately: the empty word occurs in every x.

Challenge 3: Guess Where a Reversed Word Occurs

Task: build an NPDA for

$$L = \{w\#x : w, x \in \{0,1\}^*,\ w^R \text{ is a substring of } x\}.$$

1. **Store:** push each bit of w; on the unique $\#$, enter the skip phase. The stack is now $w^R z$.
2. **Skip:** read any prefix of x without changing the stack; use an ε-move to guess the match's starting position.
3. **Match:** consume a bit only when it equals the top bit, then pop. There is no skipping within the match.
4. **Finish:** when z is exposed, enter a final tail state that reads any remaining bits without changing z.

A successful branch splits $x = u\, w^R\, v$; conversely, any such split supplies a successful guess. No phase after the first permits $\#$.

Trace idea: 10#0011**.** After $\#$ the stack is $01z$. Skip 0, match and pop 01, then read the last 1: $0011 = 0 \cdot 01 \cdot 1$.

For $w = \varepsilon$, skip $\rightarrow$ match $\rightarrow$ finish immediately: the empty word occurs in every x.

Challenge 4: Add a Finite-State Constraint to a PDA

Task: prove $C \cap R$ is context-free when C is context-free and R is regular, by an explicit product construction. Let M be a final-state NPDA for C, with states Q and finals F; let D be a DFA for R, with states H, transition η, and finals J. Use states $Q \times H$, the original stack, start (q_0, h_0), and finals $F \times J$.

Update both controls, but keep just one stack

For each $(p, \gamma) \in \delta(q, a, X)$ with $a \in \Sigma$, include

$$((p, \eta(h, a)), \gamma) \in \delta'((q, h), a, X).$$

For each $(p, \gamma) \in \delta(q, \varepsilon, X)$, include

$$((p, h), \gamma) \in \delta'((q, h), \varepsilon, X).$$

Invariant: after prefix u, the first component and stack follow M, while the second component is $\eta^*(h_0, u)$. An accepting branch exists exactly when $u \in C$ and $u \in R$.

If M is deterministic, so is the product. Two arbitrary PDAs cannot be combined this way: their two independent stacks do not fit in one PDA.

Challenge 4: Add a Finite-State Constraint to a PDA

Task: prove $C \cap R$ is context-free when C is context-free and R is regular, by an explicit product construction. Let M be a final-state NPDA for C, with states Q and finals F; let D be a DFA for R, with states H, transition η, and finals J. Use states $Q \times H$, the original stack, start (q_0, h_0), and finals $F \times J$.

Update both controls, but keep just one stack

For each $(p, \gamma) \in \delta(q, a, X)$ with $a \in \Sigma$, include

$$((p, \eta(h, a)), \gamma) \in \delta'((q, h), a, X).$$

For each $(p, \gamma) \in \delta(q, \varepsilon, X)$, include

$$((p, h), \gamma) \in \delta'((q, h), \varepsilon, X).$$

Invariant: after prefix u, the first component and stack follow M, while the second component is $\eta^*(h_0, u)$. An accepting branch exists exactly when $u \in C$ and $u \in R$.

If M is deterministic, so is the product. Two arbitrary PDAs cannot be combined this way: their two independent stacks do not fit in one PDA.

Challenge 4: Add a Finite-State Constraint to a PDA

Task: prove $C \cap R$ is context-free when C is context-free and R is regular, by an explicit product construction. Let M be a final-state NPDA for C, with states Q and finals F; let D be a DFA for R, with states H, transition η, and finals J. Use states $Q \times H$, the original stack, start (q_0, h_0), and finals $F \times J$.

Update both controls, but keep just one stack

For each $(p, \gamma) \in \delta(q, a, X)$ with $a \in \Sigma$, include

$$((p, \eta(h, a)), \gamma) \in \delta'((q, h), a, X).$$

For each $(p, \gamma) \in \delta(q, \varepsilon, X)$, include

$$((p, h), \gamma) \in \delta'((q, h), \varepsilon, X).$$

Invariant: after prefix u, the first component and stack follow M, while the second component is $\eta^*(h_0, u)$. An accepting branch exists exactly when $u \in C$ and $u \in R$.

If M is deterministic, so is the product. Two arbitrary PDAs cannot be combined this way: their two independent stacks do not fit in one PDA.

Challenge 5: One Block Pays Two Separate Debts

Additional practice: construct a CFG and a deterministic stack recognizer for

$$L = \{a^i b^{i+j} c^j : i, j \geq 0\}.$$

Grammar: $S \to AC$, $A \to aAb \mid \varepsilon$, $C \to bCc \mid \varepsilon$. The two variables generate $a^i b^i$ and $b^j c^j$, so concatenation proves both inclusions immediately.

Deterministic solution on input $w \dashv$

Enforce $a^* b^* c^*$ with states. Push A for each a. Each b first cancels an A if one remains; otherwise it pushes a B. Each c must pop a B. Accept on the endmarker exactly when only z remains. There are no guesses or competing empty moves.

After $a^i b^r$, the stack is $A^{i-r} z$ if $r < i$, and $B^{r-i} z$ otherwise. The c's must number exactly $r - i$.

For $aabbbbbccc$, the first two b's cancel the a-debt; the next three create the c-debt. Zero-length phases and ε work too. **Lesson:** sequential obligations can reuse one stack; they need not be stored as independent counters at the same time.

Challenge 5: One Block Pays Two Separate Debts

Additional practice: construct a CFG and a deterministic stack recognizer for

$$L = \{a^i b^{i+j} c^j : i, j \geq 0\}.$$

Grammar: $S \to AC$, $A \to aAb \mid \varepsilon$, $C \to bCc \mid \varepsilon$. The two variables generate $a^i b^i$ and $b^j c^j$, so concatenation proves both inclusions immediately.

Deterministic solution on input $w \dashv$

Enforce $a^* b^* c^*$ with states. Push A for each a. Each b first cancels an A if one remains; otherwise it pushes a B. Each c must pop a B. Accept on the endmarker exactly when only z remains. There are no guesses or competing empty moves.

After $a^i b^r$, the stack is $A^{i-r} z$ if $r < i$, and $B^{r-i} z$ otherwise. The c's must number exactly $r - i$.

For $aabbbbbccc$, the first two b's cancel the a-debt; the next three create the c-debt. Zero-length phases and ε work too. **Lesson:** sequential obligations can reuse one stack; they need not be stored as independent counters at the same time.

Challenge 5: One Block Pays Two Separate Debts

Additional practice: construct a CFG and a deterministic stack recognizer for

$$L = \{a^i b^{i+j} c^j : i, j \geq 0\}.$$

Grammar: $S \to AC$, $A \to aAb \mid \varepsilon$, $C \to bCc \mid \varepsilon$. The two variables generate $a^i b^i$ and $b^j c^j$, so concatenation proves both inclusions immediately.

Deterministic solution on input $w \dashv$

Enforce $a^* b^* c^*$ with states. Push A for each a. Each b first cancels an A if one remains; otherwise it pushes a B. Each c must pop a B. Accept on the endmarker exactly when only z remains. There are no guesses or competing empty moves.

After $a^i b^r$, the stack is $A^{i-r} z$ if $r < i$, and $B^{r-i} z$ otherwise. The c's must number exactly $r - i$.

For $aabbbbbccc$, the first two b's cancel the a-debt; the next three create the c-debt. Zero-length phases and ε work too. **Lesson:** sequential obligations can reuse one stack; they need not be stored as independent counters at the same time.

Challenge 5: One Block Pays Two Separate Debts

Additional practice: construct a CFG and a deterministic stack recognizer for

$$L = \{a^i b^{i+j} c^j : i, j \geq 0\}.$$

Grammar: $S \to AC$, $A \to aAb \mid \varepsilon$, $C \to bCc \mid \varepsilon$. The two variables generate $a^i b^i$ and $b^j c^j$, so concatenation proves both inclusions immediately.

Deterministic solution on input $w \dashv$

Enforce $a^* b^* c^*$ with states. Push A for each a. Each b first cancels an A if one remains; otherwise it pushes a B. Each c must pop a B. Accept on the endmarker exactly when only z remains. There are no guesses or competing empty moves.

After $a^i b^r$, the stack is $A^{i-r}z$ if $r < i$, and $B^{r-i}z$ otherwise. The c's must number exactly $r - i$.

For $aabbbbbccc$, the first two b's cancel the a-debt; the next three create the c-debt. Zero-length phases and ε work too. **Lesson:** sequential obligations can reuse one stack; they need not be stored as independent counters at the same time.

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, allow a move to a drain state that pops every remaining symbol, including the fresh marker, using only ε-moves.

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 with no outgoing moves. Final-state acceptance still requires that all input has been consumed.

These constructions preserve the existence of an accepting branch, but do not preserve determinism in general (recall the prefix-free restriction). Starting the drain too early cannot accept unread input: the drain consumes none.

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, allow a move to a drain state that pops every remaining symbol, including the fresh marker, using only ε-moves.

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 with no outgoing moves. Final-state acceptance still requires that all input has been consumed.

These constructions preserve the existence of an accepting branch, but do not preserve determinism in general (recall the prefix-free restriction). Starting the drain too early cannot accept unread input: the drain consumes none.

Exact CFG-to-NPDA Rules and Correctness

For $G = (V, T, S, P)$, choose fresh $z \notin V \cup T$ and states s, q. Start in (s, w, z) and accept by empty stack. Use

$$\begin{aligned}
\delta(s, \varepsilon, z) &= \{(q, Sz)\}, \\
\delta(q, \varepsilon, A) &= \{(q, \beta) : A \to \beta \in P\} \quad && (A \in V), \\
\delta(q, a, a) &= \{(q, \varepsilon)\} && (a \in T), \\
\delta(q, \varepsilon, z) &= \{(q, \varepsilon)\}.
\end{aligned}$$

All other entries are empty. The last rule does *not* test for end of input. **Soundness:** before the final bottom-marker pop, the invariant $S \Rightarrow^* u\alpha$ holds for consumed prefix u and stack αz. An accepting run has consumed all of w with $\alpha = \varepsilon$, so $S \Rightarrow^* w$.

Completeness: follow a leftmost derivation of w, expanding each variable at the top and consuming exposed terminals. After matching w, pop z. Every generated word therefore has an accepting computation. A branch making a different production choice may fail or loop.

Exact CFG-to-NPDA Rules and Correctness

For $G = (V, T, S, P)$, choose fresh $z \notin V \cup T$ and states s, q. Start in (s, w, z) and accept by empty stack. Use

$$\begin{aligned}
\delta(s, \varepsilon, z) &= \{(q, Sz)\}, \\
\delta(q, \varepsilon, A) &= \{(q, \beta) : A \to \beta \in P\} \quad && (A \in V), \\
\delta(q, a, a) &= \{(q, \varepsilon)\} && (a \in T), \\
\delta(q, \varepsilon, z) &= \{(q, \varepsilon)\}.
\end{aligned}$$

All other entries are empty. The last rule does *not* test for end of input. **Soundness:** before the final bottom-marker pop, the invariant $S \Rightarrow^* u\alpha$ holds for consumed prefix u and stack αz. An accepting run has consumed all of w with $\alpha = \varepsilon$, so $S \Rightarrow^* w$.

Completeness: follow a leftmost derivation of w, expanding each variable at the top and consuming exposed terminals. After matching w, pop z. Every generated word therefore has an accepting computation. A branch making a different production choice may fail or loop.

Exact CFG-to-NPDA Rules and Correctness

For $G = (V, T, S, P)$, choose fresh $z \notin V \cup T$ and states s, q. Start in (s, w, z) and accept by empty stack. Use

$$\begin{aligned}
\delta(s, \varepsilon, z) &= \{(q, Sz)\}, \\
\delta(q, \varepsilon, A) &= \{(q, \beta) : A \to \beta \in P\} \quad && (A \in V), \\
\delta(q, a, a) &= \{(q, \varepsilon)\} && (a \in T), \\
\delta(q, \varepsilon, z) &= \{(q, \varepsilon)\}.
\end{aligned}$$

All other entries are empty. The last rule does *not* test for end of input. **Soundness:** before the final bottom-marker pop, the invariant $S \Rightarrow^* u\alpha$ holds for consumed prefix u and stack αz. An accepting run has consumed all of w with $\alpha = \varepsilon$, so $S \Rightarrow^* w$.

Completeness: follow a leftmost derivation of w, expanding each variable at the top and consuming exposed terminals. After matching w, pop z. Every generated word therefore has an accepting computation. A branch making a different production choice may fail or loop.

NPDA $\Rightarrow$ CFG: State-Pair Variables

First convert to **empty-stack acceptance**. Call the resulting start state q_0 and initial stack symbol z. Introduce variables

$$[pAq],$$

for computations that start in p with A on top and end in q when A and all symbols pushed above its lower stack have been removed. The lower stack is neither read nor changed during this segment.

- 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.
- For a fresh start variable, add $S \to [q_0 z q]$ for *every* $q \in Q$, because the terminal state is irrelevant to empty-stack acceptance.

The next slide gives replacements of lengths $0, 1, 2$; a later slide gives the general finite replacement. No restriction on push length is assumed.

NPDA $\Rightarrow$ CFG: State-Pair Variables

First convert to **empty-stack acceptance**. Call the resulting start state q_0 and initial stack symbol z. Introduce variables

$$[pAq],$$

for computations that start in p with A on top and end in q when A and all symbols pushed above its lower stack have been removed. The lower stack is neither read nor changed during this segment.

- 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.
- For a fresh start variable, add $S \to [q_0 z q]$ for *every* $q \in Q$, because the terminal state is irrelevant to empty-stack acceptance.

The next slide gives replacements of lengths $0, 1, 2$; a later slide gives the general finite replacement. No restriction on push length is assumed.

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.

The other two basic cases are

$$\begin{aligned} (q, \varepsilon) \in \delta(p, a, A) \quad &\Rightarrow \quad [pAq] \to a, \\ (r, B) \in \delta(p, a, A) \quad &\Rightarrow \quad [pAq] \to a[rBq] \quad (q \in Q). \end{aligned}$$

Here $a \in \Sigma \cup \{\varepsilon\}$; if $a = \varepsilon$, no terminal is written. These cases explain how stack lifetimes become grammar structure.

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.

The other two basic cases are

$$\begin{aligned} (q, \varepsilon) \in \delta(p, a, A) \quad &\Rightarrow \quad [pAq] \to a, \\ (r, B) \in \delta(p, a, A) \quad &\Rightarrow \quad [pAq] \to a[rBq] \quad (q \in Q). \end{aligned}$$

Here $a \in \Sigma \cup \{\varepsilon\}$; if $a = \varepsilon$, no terminal is written. These cases explain how stack lifetimes become grammar structure.

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.

The other two basic cases are

$$\begin{aligned} (q, \varepsilon) \in \delta(p, a, A) \quad &\Rightarrow \quad [pAq] \to a, \\ (r, B) \in \delta(p, a, A) \quad &\Rightarrow \quad [pAq] \to a[rBq] \quad (q \in Q). \end{aligned}$$

Here $a \in \Sigma \cup \{\varepsilon\}$; if $a = \varepsilon$, no terminal is written. These cases explain how stack lifetimes become grammar structure.

The General State-Pair Construction

Let M accept by empty stack. For a transition $(r, B_1 \cdots B_k) \in \delta(p, a, A)$ with $k \geq 1$, add

$$[pAq_k] \to a[rB_1q_1][q_1B_2q_2] \cdots [q_{k-1}B_kq_k]$$

for every $q_1, \ldots, q_k \in Q$. For $k = 0$, add only $[pAr] \to a$. Omit a when it is ε. Start with $S \to [q_0zq]$ for all $q \in Q$.

Why the construction is finite

The NPDA has finitely many rules and a finite maximum replacement length. Each rule uses finitely many choices of intermediate states. There are $|Q|^2|\Gamma|$ state-pair variables, plus the new start.

Why it is correct: the stack discipline forces all descendants of B_1 to disappear before B_2 is exposed, and so on. Split an accepting computation at these first exposures; its input splits into exactly the factors represented on the right-hand side. Conversely, concatenating those stack-removal computations implements the production without touching the lower stack. Induction on the computation length / derivation tree proves the two directions.

The General State-Pair Construction

Let M accept by empty stack. For a transition $(r, B_1 \cdots B_k) \in \delta(p, a, A)$ with $k \geq 1$, add

$$[pAq_k] \to a[rB_1q_1][q_1B_2q_2] \cdots [q_{k-1}B_kq_k]$$

for every $q_1, \ldots, q_k \in Q$. For $k = 0$, add only $[pAr] \to a$. Omit a when it is ε. Start with $S \to [q_0zq]$ for all $q \in Q$.

Why the construction is finite

The NPDA has finitely many rules and a finite maximum replacement length. Each rule uses finitely many choices of intermediate states. There are $|Q|^2|\Gamma|$ state-pair variables, plus the new start.

Why it is correct: the stack discipline forces all descendants of B_1 to disappear before B_2 is exposed, and so on. Split an accepting computation at these first exposures; its input splits into exactly the factors represented on the right-hand side. Conversely, concatenating those stack-removal computations implements the production without touching the lower stack. Induction on the computation length / derivation tree proves the two directions.

The General State-Pair Construction

Let M accept by empty stack. For a transition $(r, B_1 \cdots B_k) \in \delta(p, a, A)$ with $k \geq 1$, add

$$[pAq_k] \to a[rB_1q_1][q_1B_2q_2] \cdots [q_{k-1}B_kq_k]$$

for every $q_1, \ldots, q_k \in Q$. For $k = 0$, add only $[pAr] \to a$. Omit a when it is ε. Start with $S \to [q_0zq]$ for all $q \in Q$.

Why the construction is finite

The NPDA has finitely many rules and a finite maximum replacement length. Each rule uses finitely many choices of intermediate states. There are $|Q|^2|\Gamma|$ state-pair variables, plus the new start.

Why it is correct: the stack discipline forces all descendants of B_1 to disappear before B_2 is exposed, and so on. Split an accepting computation at these first exposures; its input splits into exactly the factors represented on the right-hand side. Conversely, concatenating those stack-removal computations implements the production without touching the lower stack. Induction on the computation length / derivation tree proves the two directions.

A Small State-Pair Example

Use states p, q, initial stack symbol Z, and empty-stack acceptance. The only transitions are

$$\delta(p, a, Z) = \{(p, XZ)\}, \qquad \delta(p, b, X) = \{(q, \varepsilon)\}, \qquad \delta(q, \varepsilon, Z) = \{(q, \varepsilon)\}.$$

The only accepted word is ab: $(p, ab, Z) \vdash (p, b, XZ) \vdash (q, \varepsilon, Z) \vdash (q, \varepsilon, \varepsilon)$.

The productive part of the constructed grammar

$$S \to [pZq], \qquad [pZq] \to a[pXq][qZq], \qquad [pXq] \to b, \qquad [qZq] \to \varepsilon.$$

Thus $S \Rightarrow [pZq] \Rightarrow a[pXq][qZq] \Rightarrow ab[qZq] \Rightarrow ab$.

The full construction also adds $S \to [pZp]$ and productions for other intermediate-state choices. They do not yield terminal words here and are removed by grammar simplification. The state guesses are checked by whether their variables can actually derive a word.

A Small State-Pair Example

Use states p, q, initial stack symbol Z, and empty-stack acceptance. The only transitions are

$$\delta(p, a, Z) = \{(p, XZ)\}, \qquad \delta(p, b, X) = \{(q, \varepsilon)\}, \qquad \delta(q, \varepsilon, Z) = \{(q, \varepsilon)\}.$$

The only accepted word is ab: $(p, ab, Z) \vdash (p, b, XZ) \vdash (q, \varepsilon, Z) \vdash (q, \varepsilon, \varepsilon)$.

The productive part of the constructed grammar

$$S \to [pZq], \qquad [pZq] \to a[pXq][qZq], \qquad [pXq] \to b, \qquad [qZq] \to \varepsilon.$$

Thus $S \Rightarrow [pZq] \Rightarrow a[pXq][qZq] \Rightarrow ab[qZq] \Rightarrow ab$.

The full construction also adds $S \to [pZp]$ and productions for other intermediate-state choices. They do not yield terminal words here and are removed by grammar simplification. The state guesses are checked by whether their variables can actually derive a word.

A Small State-Pair Example

Use states p, q, initial stack symbol Z, and empty-stack acceptance. The only transitions are

$$\delta(p, a, Z) = \{(p, XZ)\}, \qquad \delta(p, b, X) = \{(q, \varepsilon)\}, \qquad \delta(q, \varepsilon, Z) = \{(q, \varepsilon)\}.$$

The only accepted word is ab: $(p, ab, Z) \vdash (p, b, XZ) \vdash (q, \varepsilon, Z) \vdash (q, \varepsilon, \varepsilon)$.

The productive part of the constructed grammar

$$S \to [pZq], \qquad [pZq] \to a[pXq][qZq], \qquad [pXq] \to b, \qquad [qZq] \to \varepsilon.$$

Thus $S \Rightarrow [pZq] \Rightarrow a[pXq][qZq] \Rightarrow ab[qZq] \Rightarrow ab$.

The full construction also adds $S \to [pZp]$ and productions for other intermediate-state choices. They do not yield terminal words here and are removed by grammar simplification. The state guesses are checked by whether their variables can actually derive a word.

Important Closure Properties of DCFLs

DCFLs are closed under:

- complement relative to a fixed Σ^*, after appropriate DPDA normalization (not simply swapping final states); 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.

Important Closure Properties of DCFLs

DCFLs are closed under:

- complement relative to a fixed Σ^*, after appropriate DPDA normalization (not simply swapping final states); 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.

Why Complementing a DPDA Is Not Just Flipping Finals

Consider alphabet $\{a\}$, one nonfinal state q, and the only rule $\delta(q, \varepsilon, z) = \{(q, z)\}$. Its language is empty.

The failed shortcut

Making q final accepts ε, but it still does not accept a: the machine loops without consuming that input. The complement should be $\{a\}^*$, not merely $\{\varepsilon\}$.

The theorem remains true: DCFLs are closed under complement. Its proof requires normalization that handles blocking and infinite empty-move behavior and arranges a definitive acceptance/rejection decision after processing the input. One then interchanges these outcomes. An endmarker is useful in this construction, but adding it alone does not cure an infinite empty loop.

A recognizer is not automatically a decider

Determinism gives a unique possible continuation, not progress or termination. The analogous DFA shortcut works only because a complete DFA consumes exactly one symbol on each step and always finishes.

Why Complementing a DPDA Is Not Just Flipping Finals

Consider alphabet $\{a\}$, one nonfinal state q, and the only rule $\delta(q, \varepsilon, z) = \{(q, z)\}$. Its language is empty.

The failed shortcut

Making q final accepts ε, but it still does not accept a: the machine loops without consuming that input. The complement should be $\{a\}^*$, not merely $\{\varepsilon\}$.

The theorem remains true: DCFLs are closed under complement. Its proof requires normalization that handles blocking and infinite empty-move behavior and arranges a definitive acceptance/rejection decision after processing the input. One then interchanges these outcomes. An endmarker is useful in this construction, but adding it alone does not cure an infinite empty loop.

A recognizer is not automatically a decider

Determinism gives a unique possible continuation, not progress or termination. The analogous DFA shortcut works only because a complete DFA consumes exactly one symbol on each step and always finishes.

Why Complementing a DPDA Is Not Just Flipping Finals

Consider alphabet $\{a\}$, one nonfinal state q, and the only rule $\delta(q, \varepsilon, z) = \{(q, z)\}$. Its language is empty.

The failed shortcut

Making q final accepts ε, but it still does not accept a: the machine loops without consuming that input. The complement should be $\{a\}^*$, not merely $\{\varepsilon\}$.

The theorem remains true: DCFLs are closed under complement. Its proof requires normalization that handles blocking and infinite empty-move behavior and arranges a definitive acceptance/rejection decision after processing the input. One then interchanges these outcomes. An endmarker is useful in this construction, but adding it alone does not cure an infinite empty loop.

A recognizer is not automatically a decider

Determinism gives a unique possible continuation, not progress or termination. The analogous DFA shortcut works only because a complete DFA consumes exactly one symbol on each step and always finishes.

A Fixed Stack-Height Bound Gives a Regular Language

Theorem

Fix a constant $K \geq 1$, independent of the input. Restrict a PDA to computations whose stack height never exceeds K (including the bottom marker). The resulting language is regular.

Proof: encode each pair (q, α), with $|\alpha| \leq K$, as a state of an ε-NFA. There are at most

$$|Q| \sum_{i=0}^{K} |\Gamma|^i$$

such states. Copy the PDA's input/empty moves between these pairs; discard moves exceeding the bound. Mark acceptance according to the original final-state or empty-stack test. Runs correspond exactly. This is an NFA state bound, not necessarily a DFA state bound.

Do not overgeneralize

A bound depending on input length is not a fixed bound. Also, a PDA may use unbounded stack space while accepting a regular language: a final work state that pushes X on every input symbol accepts Σ^* even though it never pops.

A Fixed Stack-Height Bound Gives a Regular Language

Theorem

Fix a constant $K \geq 1$, independent of the input. Restrict a PDA to computations whose stack height never exceeds K (including the bottom marker). The resulting language is regular.

Proof: encode each pair (q, α), with $|\alpha| \leq K$, as a state of an ε-NFA. There are at most

$$|Q| \sum_{i=0}^{K} |\Gamma|^i$$

such states. Copy the PDA's input/empty moves between these pairs; discard moves exceeding the bound. Mark acceptance according to the original final-state or empty-stack test. Runs correspond exactly. This is an NFA state bound, not necessarily a DFA state bound.

Do not overgeneralize

A bound depending on input length is not a fixed bound. Also, a PDA may use unbounded stack space while accepting a regular language: a final work state that pushes X on every input symbol accepts Σ^* even though it never pops.

A Fixed Stack-Height Bound Gives a Regular Language

Theorem

Fix a constant $K \geq 1$, independent of the input. Restrict a PDA to computations whose stack height never exceeds K (including the bottom marker). The resulting language is regular.

Proof: encode each pair (q, α), with $|\alpha| \leq K$, as a state of an ε-NFA. There are at most

$$|Q| \sum_{i=0}^{K} |\Gamma|^i$$

such states. Copy the PDA's input/empty moves between these pairs; discard moves exceeding the bound. Mark acceptance according to the original final-state or empty-stack test. Runs correspond exactly. This is an NFA state bound, not necessarily a DFA state bound.

Do not overgeneralize

A bound depending on input length is not a fixed bound. Also, a PDA may use unbounded stack space while accepting a regular language: a final work state that pushes X on every input symbol accepts Σ^* even though it never pops.

Two Stacks Can Simulate a Turing Tape

Further challenge: Sipser's MIT Problem Set 2 (2025), Problem 4 asks about the power of k-stack PDAs [S25]. Keep the current tape symbol in finite control. The left stack stores cells left of the head, nearest first; the right stack does the same on the right. Both have protected bottom markers $\perp$.

$$\cdots a\ b\ \boxed{c}\ d\ e \cdots \quad \longleftrightarrow \quad (\text{left}, \text{current}, \text{right}) = (ba\perp, c, de\perp).$$

Simulate a write and a head move

Write x, then move right: push x on the left stack and pop the right stack into finite control. The example becomes $(xba\perp, d, e\perp)$. Moving left is symmetric. At a bottom marker, supply a blank instead of popping the marker.

Load the input using a stack transfer; then simulate tape steps with empty-input moves. Conversely, a TM can simulate any fixed finite number of stacks. Thus two stacks already give Turing-recognizable languages; a third adds no language power. For example, $\{a^n b^n c^n : n \geq 0\}$ separates two stacks from one.

This is a computability claim, not a real-time simulation claim.

Two Stacks Can Simulate a Turing Tape

Further challenge: Sipser's MIT Problem Set 2 (2025), Problem 4 asks about the power of k-stack PDAs [S25]. Keep the current tape symbol in finite control. The left stack stores cells left of the head, nearest first; the right stack does the same on the right. Both have protected bottom markers $\perp$.

$$\cdots a\ b\ \boxed{c}\ d\ e \cdots \quad \longleftrightarrow \quad (\text{left}, \text{current}, \text{right}) = (ba\perp, c, de\perp).$$

Simulate a write and a head move

Write x, then move right: push x on the left stack and pop the right stack into finite control. The example becomes $(xba\perp, d, e\perp)$. Moving left is symmetric. At a bottom marker, supply a blank instead of popping the marker.

Load the input using a stack transfer; then simulate tape steps with empty-input moves. Conversely, a TM can simulate any fixed finite number of stacks. Thus two stacks already give Turing-recognizable languages; a third adds no language power. For example, $\{a^n b^n c^n : n \geq 0\}$ separates two stacks from one.

This is a computability claim, not a real-time simulation claim.

Two Stacks Can Simulate a Turing Tape

Further challenge: Sipser's MIT Problem Set 2 (2025), Problem 4 asks about the power of k-stack PDAs [S25]. Keep the current tape symbol in finite control. The left stack stores cells left of the head, nearest first; the right stack does the same on the right. Both have protected bottom markers $\perp$.

$$\cdots a\ b\ \boxed{c}\ d\ e \cdots \quad \longleftrightarrow \quad (\text{left}, \text{current}, \text{right}) = (ba\perp, c, de\perp).$$

Simulate a write and a head move

Write x, then move right: push x on the left stack and pop the right stack into finite control. The example becomes $(xba\perp, d, e\perp)$. Moving left is symmetric. At a bottom marker, supply a blank instead of popping the marker.

Load the input using a stack transfer; then simulate tape steps with empty-input moves. Conversely, a TM can simulate any fixed finite number of stacks. Thus two stacks already give Turing-recognizable languages; a third adds no language power. For example, $\{a^n b^n c^n : n \geq 0\}$ separates two stacks from one.

This is a computability claim, not a real-time simulation claim.

Visibly Pushdown Automata: A Useful Middle Ground

Ordinary NPDAs cannot always be determinized. A useful restriction makes the kind of stack action visible in the next input symbol. Fix a partition of the alphabet:

$$\Sigma = \Sigma_{\text{call}} \dot{\cup} \Sigma_{\text{return}} \dot{\cup} \Sigma_{\text{internal}}.$$

Calls push one symbol; returns pop one (with a specified bottom-marker convention); internal symbols leave the stack unchanged. There are no arbitrary empty-input stack actions. Control and stack-symbol choices may still be nondeterministic.

Theorem for finite words [AM04]

Every nondeterministic visibly pushdown automaton has an equivalent deterministic one. Over the same fixed alphabet partition, these languages are closed under union, intersection, and complement.

Procedure-call/return traces provide a natural application: matching a return to its call is nested structure. This models finite abstractions of program behavior, not every detail of an arbitrary program.

Determinization needs summaries of matched stack activity, not merely a subset of control states as in the NFA construction.

Visibly Pushdown Automata: A Useful Middle Ground

Ordinary NPDAs cannot always be determinized. A useful restriction makes the kind of stack action visible in the next input symbol. Fix a partition of the alphabet:

$$\Sigma = \Sigma_{\text{call}} \dot{\cup} \Sigma_{\text{return}} \dot{\cup} \Sigma_{\text{internal}}.$$

Calls push one symbol; returns pop one (with a specified bottom-marker convention); internal symbols leave the stack unchanged. There are no arbitrary empty-input stack actions. Control and stack-symbol choices may still be nondeterministic.

Theorem for finite words [AM04]

Every nondeterministic visibly pushdown automaton has an equivalent deterministic one. Over the same fixed alphabet partition, these languages are closed under union, intersection, and complement.

Procedure-call/return traces provide a natural application: matching a return to its call is nested structure. This models finite abstractions of program behavior, not every detail of an arbitrary program.

Determinization needs summaries of matched stack activity, not merely a subset of control states as in the NFA construction.

Visibly Pushdown Automata: A Useful Middle Ground

Ordinary NPDAs cannot always be determinized. A useful restriction makes the kind of stack action visible in the next input symbol. Fix a partition of the alphabet:

$$\Sigma = \Sigma_{\text{call}} \dot{\cup} \Sigma_{\text{return}} \dot{\cup} \Sigma_{\text{internal}}.$$

Calls push one symbol; returns pop one (with a specified bottom-marker convention); internal symbols leave the stack unchanged. There are no arbitrary empty-input stack actions. Control and stack-symbol choices may still be nondeterministic.

Theorem for finite words [AM04]

Every nondeterministic visibly pushdown automaton has an equivalent deterministic one. Over the same fixed alphabet partition, these languages are closed under union, intersection, and complement.

Procedure-call/return traces provide a natural application: matching a return to its call is nested structure. This models finite abstractions of program behavior, not every detail of an arbitrary program.

Determinization needs summaries of matched stack activity, not merely a subset of control states as in the NFA construction.

A Predictive Parser: Choosing with One Token

For $S \to aSb \mid \varepsilon$, store $S \dashv$ with the top on the left. Unlike the generic CFG-simulation NPDA, a predictive parser chooses expansions using the next input token (lookahead).

Stack top and lookahead	Action
S, lookahead a	replace S by aSb
S, lookahead b or $\dashv$	replace S by ε
terminal a or b, same lookahead	pop and consume that token
$\dashv$, lookahead $\dashv$	accept
any other combination	error

Why these choices? The nonempty alternative begins with a; an S may be followed by b or by end of input. These are its *FIRST* and *FOLLOW* clues. This is an LL(1) grammar: left-to-right scan, leftmost derivation, one-token lookahead.

An empty expansion does not mean acceptance. For input $b \dashv$, the parser removes S, then reports a mismatch between $\dashv$ and b. A PDA implementation buffers lookahead in its finite control.

A Predictive Parser: Choosing with One Token

For $S \to aSb \mid \varepsilon$, store $S \dashv$ with the top on the left. Unlike the generic CFG-simulation NPDA, a predictive parser chooses expansions using the next input token (lookahead).

Stack top and lookahead	Action
S, lookahead a	replace S by aSb
S, lookahead b or $\dashv$	replace S by ε
terminal a or b, same lookahead	pop and consume that token
$\dashv$, lookahead $\dashv$	accept
any other combination	error

Why these choices? The nonempty alternative begins with a; an S may be followed by b or by end of input. These are its *FIRST* and *FOLLOW* clues. This is an LL(1) grammar: left-to-right scan, leftmost derivation, one-token lookahead.

An empty expansion does not mean acceptance. For input $b \dashv$, the parser removes S, then reports a mismatch between $\dashv$ and b. A PDA implementation buffers lookahead in its finite control.

A Predictive Parser: Choosing with One Token

For $S \to aSb \mid \varepsilon$, store $S \dashv$ with the top on the left. Unlike the generic CFG-simulation NPDA, a predictive parser chooses expansions using the next input token (lookahead).

Stack top and lookahead	Action
S, lookahead a	replace S by aSb
S, lookahead b or $\dashv$	replace S by ε
terminal a or b, same lookahead	pop and consume that token
$\dashv$, lookahead $\dashv$	accept
any other combination	error

Why these choices? The nonempty alternative begins with a; an S may be followed by b or by end of input. These are its *FIRST* and *FOLLOW* clues. This is an LL(1) grammar: left-to-right scan, leftmost derivation, one-token lookahead.

An empty expansion does not mean acceptance. For input $b \dashv$, the parser removes S, then reports a mismatch between $\dashv$ and b. A PDA implementation buffers lookahead in its finite control.

Concrete Conflicts: Which Structure Should Be Built?

Shift–reduce: $E \to E + E \mid E \times E \mid \mathrm{id}$

With stack suffix $E + E$ and lookahead $\times$, reducing first groups $(\mathrm{id} + \mathrm{id}) \times \mathrm{id}$; shifting allows $\mathrm{id} + (\mathrm{id} \times \mathrm{id})$. Choosing the usual precedence calls for shifting here.

Reduce–reduce: $S \to A \mid B$, $A \to a$, $B \to a$

After reading a at end of input, reducing to A or to B leads to distinct parses. Here the ambiguity is explicit: $S \Rightarrow A \Rightarrow a$ and $S \Rightarrow B \Rightarrow a$.

Method failure is not always grammar ambiguity

$S \to aSb \mid ab$ is unambiguous but not LL(1) as written: both alternatives start with a. Factoring as $S \to aR$, $R \to Sb \mid b$ permits a one-token choice. A parsing conflict alone is therefore not a proof of ambiguity.

Concrete Conflicts: Which Structure Should Be Built?

Shift–reduce: $E \to E + E \mid E \times E \mid \mathrm{id}$

With stack suffix $E + E$ and lookahead $\times$, reducing first groups $(\mathrm{id} + \mathrm{id}) \times \mathrm{id}$; shifting allows $\mathrm{id} + (\mathrm{id} \times \mathrm{id})$. Choosing the usual precedence calls for shifting here.

Reduce–reduce: $S \to A \mid B$, $A \to a$, $B \to a$

After reading a at end of input, reducing to A or to B leads to distinct parses. Here the ambiguity is explicit: $S \Rightarrow A \Rightarrow a$ and $S \Rightarrow B \Rightarrow a$.

Method failure is not always grammar ambiguity

$S \to aSb \mid ab$ is unambiguous but not LL(1) as written: both alternatives start with a. Factoring as $S \to aR$, $R \to Sb \mid b$ permits a one-token choice. A parsing conflict alone is therefore not a proof of ambiguity.

Concrete Conflicts: Which Structure Should Be Built?

Shift–reduce: $E \to E + E \mid E \times E \mid \mathrm{id}$

With stack suffix $E + E$ and lookahead $\times$, reducing first groups $(\mathrm{id} + \mathrm{id}) \times \mathrm{id}$; shifting allows $\mathrm{id} + (\mathrm{id} \times \mathrm{id})$. Choosing the usual precedence calls for shifting here.

Reduce–reduce: $S \to A \mid B$, $A \to a$, $B \to a$

After reading a at end of input, reducing to A or to B leads to distinct parses. Here the ambiguity is explicit: $S \Rightarrow A \Rightarrow a$ and $S \Rightarrow B \Rightarrow a$.

Method failure is not always grammar ambiguity

$S \to aSb \mid ab$ is unambiguous but not LL(1) as written: both alternatives start with a. Factoring as $S \to aR$, $R \to Sb \mid b$ permits a one-token choice. A parsing conflict alone is therefore not a proof of ambiguity.

From Grammar Symbols to LR States

Augment $S \to aSb \mid ab$ with $S' \to S$. An **item** is a production with a dot marking how much has been recognized. Closure adds $S \to \cdot aSb$ and $S \to \cdot ab$ whenever the dot is before S.

State	LR(0) items
0	$S' \to \cdot S$, $S \to \cdot aSb$, $S \to \cdot ab$
1	$S \to a \cdot Sb$, $S \to a \cdot b$, $S \to \cdot aSb$, $S \to \cdot ab$
2	$S' \to S \cdot$
3	$S \to ab \cdot$
4	$S \to aS \cdot b$
5	$S \to aSb \cdot$

Moving the dot over a symbol gives a transition to another item set: $0 \xrightarrow{a} 1$, $0 \xrightarrow{S} 2$, $1 \xrightarrow{a} 1$, $1 \xrightarrow{b} 3$, $1 \xrightarrow{S} 4$, $4 \xrightarrow{b} 5$. States 3 and 5 specify reductions; state 2 specifies acceptance at end of input.

From Grammar Symbols to LR States

Augment $S \to aSb \mid ab$ with $S' \to S$. An **item** is a production with a dot marking how much has been recognized. Closure adds $S \to \cdot aSb$ and $S \to \cdot ab$ whenever the dot is before S.

State	LR(0) items
0	$S' \to \cdot S$, $S \to \cdot aSb$, $S \to \cdot ab$
1	$S \to a \cdot Sb$, $S \to a \cdot b$, $S \to \cdot aSb$, $S \to \cdot ab$
2	$S' \to S \cdot$
3	$S \to ab \cdot$
4	$S \to aS \cdot b$
5	$S \to aSb \cdot$

Moving the dot over a symbol gives a transition to another item set: $0 \xrightarrow{a} 1$, $0 \xrightarrow{S} 2$, $1 \xrightarrow{a} 1$, $1 \xrightarrow{b} 3$, $1 \xrightarrow{S} 4$, $4 \xrightarrow{b} 5$. States 3 and 5 specify reductions; state 2 specifies acceptance at end of input.

A Complete Small LR Action/Goto Table

Number the rules $r_1 : S \to aSb$ and $r_2 : S \to ab$. In the table, s_j means shift and push state j; a blank means error.

State	ACTION (lookahead) a	b	$\dashv$	GOTO S
0	s_1			2
1	s_1	s_3		4
2			accept	
3	r_2	r_2	r_2	
4		s_5		
5	r_1	r_1	r_1	

Keep states interleaved with symbols. On a reduction $A \to \beta$, pop $|\beta|$ symbol/state pairs, expose state i, then push A and GOTO(i, A). Do not consume the lookahead.

The earlier *aabb* trace, now including states

$$0 \to 0a1 \to 0a1a1 \to 0a1a1b3$$
$$\to 0a1S4 \to 0a1S4b5 \to 0S2.$$

The successive actions are shift, shift, shift, r_2, shift, r_1, then accept on $\dashv$.

A Complete Small LR Action/Goto Table

Number the rules $r_1 : S \to aSb$ and $r_2 : S \to ab$. In the table, s_j means shift and push state j; a blank means error.

State	ACTION (lookahead) a	b	$\dashv$	GOTO S
0	s_1			2
1	s_1	s_3		4
2			accept	
3	r_2	r_2	r_2	
4		s_5		
5	r_1	r_1	r_1	

Keep states interleaved with symbols. On a reduction $A \to \beta$, pop $|\beta|$ symbol/state pairs, expose state i, then push A and GOTO(i, A). Do not consume the lookahead.

The earlier *aabb* trace, now including states

$$0 \to 0a1 \to 0a1a1 \to 0a1a1b3$$
$$\to 0a1S4 \to 0a1S4b5 \to 0S2.$$

The successive actions are shift, shift, shift, r_2, shift, r_1, then accept on $\dashv$.

A Complete Small LR Action/Goto Table

Number the rules $r_1 : S \to aSb$ and $r_2 : S \to ab$. In the table, s_j means shift and push state j; a blank means error.

State	ACTION (lookahead) a	b	$\dashv$	GOTO S
0	s_1			2
1	s_1	s_3		4
2			accept	
3	r_2	r_2	r_2	
4		s_5		
5	r_1	r_1	r_1	

Keep states interleaved with symbols. On a reduction $A \to \beta$, pop $|\beta|$ symbol/state pairs, expose state i, then push A and GOTO(i, A). Do not consume the lookahead.

The earlier *aabb* trace, now including states

$$0 \to 0a1 \to 0a1a1 \to 0a1a1b3$$
$$\to 0a1S4 \to 0a1S4b5 \to 0S2.$$

The successive actions are shift, shift, shift, r_2, shift, r_1, then accept on $\dashv$.

Additional Practice: Explain, Then Repair

1. At the same (q, X), a machine has $\varepsilon, X \to X$ and $a, X \to XX$, each with just one target. Is it deterministic? **Answer:** no. With next symbol a, both moves are enabled. A design must separate the conditions, for example into different phases.
2. In the CFG simulator for $S \to 0S1 \mid \varepsilon$, choose $S \to \varepsilon$ immediately on input 01. What happens? **Answer:** the stack can empty, but unread 01 prevents acceptance. The invariant still holds: $S \Rightarrow^* \varepsilon$ for this bad branch.
3. Why can a DFA recognize $\{a, aa\}$, but an unmarked empty-stack DPDA cannot? **Answer:** final-state acceptance can accept a prefix and continue; after empty-stack acceptance, the unique run cannot process an extension.

These are newly phrased practice questions, not numbered textbook exercises.

Additional Practice: Explain, Then Repair

1. At the same (q, X), a machine has $\varepsilon, X \to X$ and $a, X \to XX$, each with just one target. Is it deterministic? **Answer:** no. With next symbol a, both moves are enabled. A design must separate the conditions, for example into different phases.
2. In the CFG simulator for $S \to 0S1 \mid \varepsilon$, choose $S \to \varepsilon$ immediately on input 01. What happens? **Answer:** the stack can empty, but unread 01 prevents acceptance. The invariant still holds: $S \Rightarrow^* \varepsilon$ for this bad branch.
3. Why can a DFA recognize $\{a, aa\}$, but an unmarked empty-stack DPDA cannot? **Answer:** final-state acceptance can accept a prefix and continue; after empty-stack acceptance, the unique run cannot process an extension.

These are newly phrased practice questions, not numbered textbook exercises.

Additional Practice: Explain, Then Repair

1. At the same (q, X), a machine has $\varepsilon, X \to X$ and $a, X \to XX$, each with just one target. Is it deterministic? **Answer:** no. With next symbol a, both moves are enabled. A design must separate the conditions, for example into different phases.
2. In the CFG simulator for $S \to 0S1 \mid \varepsilon$, choose $S \to \varepsilon$ immediately on input 01. What happens? **Answer:** the stack can empty, but unread 01 prevents acceptance. The invariant still holds: $S \Rightarrow^* \varepsilon$ for this bad branch.
3. Why can a DFA recognize $\{a, aa\}$, but an unmarked empty-stack DPDA cannot? **Answer:** final-state acceptance can accept a prefix and continue; after empty-stack acceptance, the unique run cannot process an extension.

These are newly phrased practice questions, not numbered textbook exercises.

Additional Practice: Explain, Then Repair

1. At the same (q, X), a machine has $\varepsilon, X \to X$ and $a, X \to XX$, each with just one target. Is it deterministic? **Answer:** no. With next symbol a, both moves are enabled. A design must separate the conditions, for example into different phases.
2. In the CFG simulator for $S \to 0S1 \mid \varepsilon$, choose $S \to \varepsilon$ immediately on input 01. What happens? **Answer:** the stack can empty, but unread 01 prevents acceptance. The invariant still holds: $S \Rightarrow^* \varepsilon$ for this bad branch.
3. Why can a DFA recognize $\{a, aa\}$, but an unmarked empty-stack DPDA cannot? **Answer:** final-state acceptance can accept a prefix and continue; after empty-stack acceptance, the unique run cannot process an extension.

These are newly phrased practice questions, not numbered textbook exercises.

Additional Practice: Explain, Then Repair

1. At the same (q, X), a machine has $\varepsilon, X \to X$ and $a, X \to XX$, each with just one target. Is it deterministic? **Answer:** no. With next symbol a, both moves are enabled. A design must separate the conditions, for example into different phases.
2. In the CFG simulator for $S \to 0S1 \mid \varepsilon$, choose $S \to \varepsilon$ immediately on input 01. What happens? **Answer:** the stack can empty, but unread 01 prevents acceptance. The invariant still holds: $S \Rightarrow^* \varepsilon$ for this bad branch.
3. Why can a DFA recognize $\{a, aa\}$, but an unmarked empty-stack DPDA cannot? **Answer:** final-state acceptance can accept a prefix and continue; after empty-stack acceptance, the unique run cannot process an extension.

These are newly phrased practice questions, not numbered textbook exercises.

Additional Practice: Explain, Then Repair

1. At the same (q, X), a machine has $\varepsilon, X \to X$ and $a, X \to XX$, each with just one target. Is it deterministic? **Answer:** no. With next symbol a, both moves are enabled. A design must separate the conditions, for example into different phases.
2. In the CFG simulator for $S \to 0S1 \mid \varepsilon$, choose $S \to \varepsilon$ immediately on input 01. What happens? **Answer:** the stack can empty, but unread 01 prevents acceptance. The invariant still holds: $S \Rightarrow^* \varepsilon$ for this bad branch.
3. Why can a DFA recognize $\{a, aa\}$, but an unmarked empty-stack DPDA cannot? **Answer:** final-state acceptance can accept a prefix and continue; after empty-stack acceptance, the unique run cannot process an extension.

These are newly phrased practice questions, not numbered textbook exercises.

References: Appendix Challenges and Further Results

[S06] Michael Sipser, *Introduction to the Theory of Computation*, 2nd ed., Thomson Course Technology, 2006. Exercises 2.7(a,c), 2.10 and Problem 2.18(a). These numbers refer specifically to the second edition.

[S25] Michael Sipser, MIT 18.404/6.5400, Problem Set 2, Fall 2025, Problem 4. Two stacks versus one, and simulation of more stacks.

[AM04] Rajeev Alur and P. Madhusudan, *Visibly Pushdown Languages*, STOC 2004, pp. 202–211; linked expanded manuscript dated August 22, 2005. Determinization and Boolean closure for finite-word languages.

The split-count challenge and the bounded-stack proof are additional course material, not attributed to numbered textbook exercises.

References: Constructions and Practical Parsing


Michael Sipser, MIT 18.404J, Fall 2020, Lecture 4: Pushdown Automata, CFG to PDA and Reverse Conversion.


Alfred V. Aho, Columbia COMS W3261, Lecture 8: Pushdown Automata. IDs, acceptance modes, and determinism.


Jean Gallier, University of Pennsylvania CIS 511, Pushdown Automata. Formal grammar–PDA constructions.


Cornell CS 4120, Spring 2023, LR parsing notes. Items, parser states, action/goto tables, and conflicts.


GNU Bison manual, Shift/Reduce Conflicts. Practical disambiguation policies.

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 CFG–PDA equivalence and nondeterministic computation. Teach the core sequence first; select appendix challenges and proof details to suit the course depth.