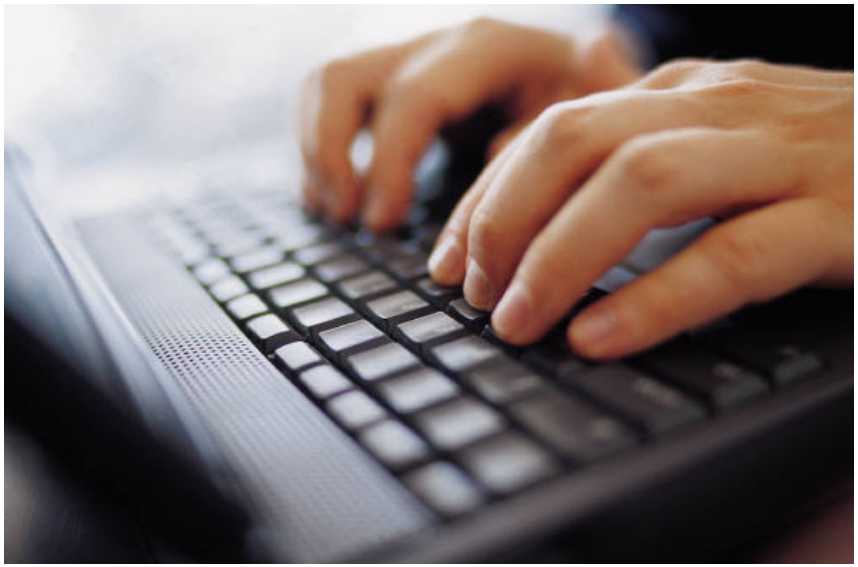
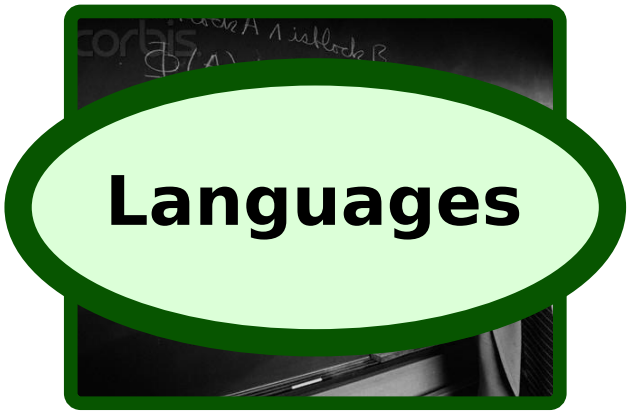
# 02141 Computer Science Modelling
# Context-Free Languages

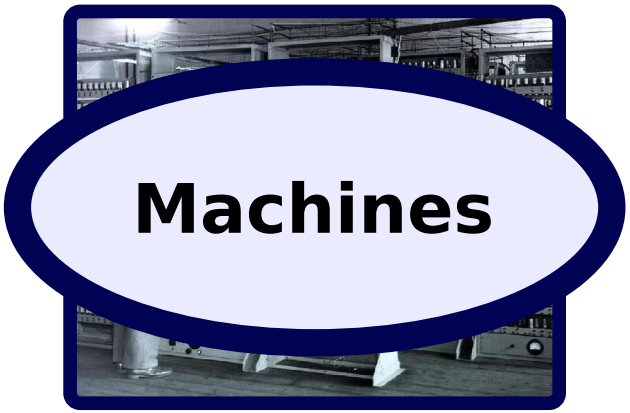
## CFL4: Pushdown Automata



# Automata for Context-Free Languages




**Mathematics**
**(theory)**

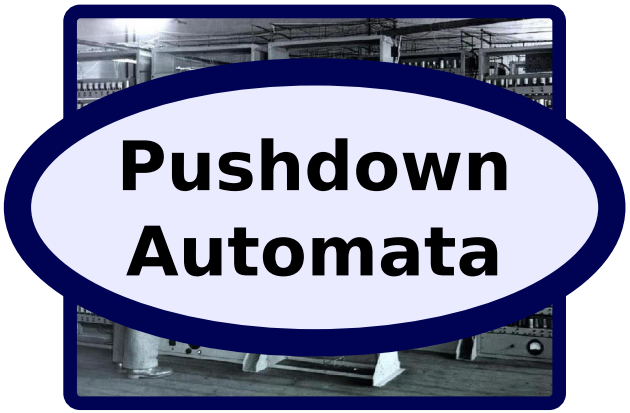



**Engineering**
**(implementation)**


**Linguistics**
**(description / instruction)**

# Automata for Context-Free Languages


**Mathematics**
**(theory)**




**Engineering**
**(implementation)**


**Linguistics**
**(description / instruction)**

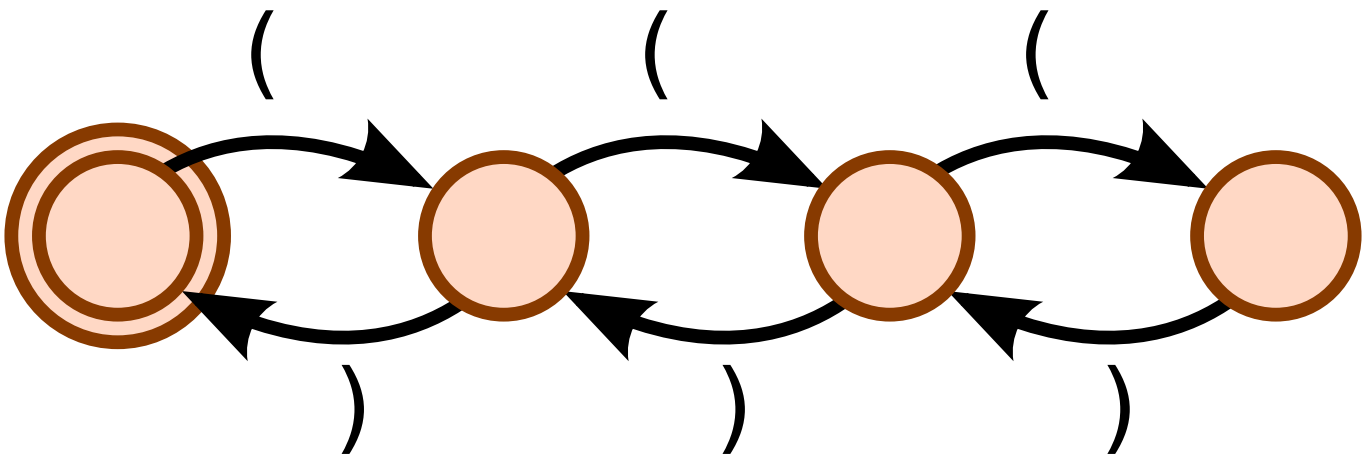
# Automata for Context-Free Languages

We have seen already that **finite automata** are not powerful enough to recognize **context-free languages**.

# Automata for Context-Free Languages

We have seen already that **finite automata** are not powerful enough to recognize **context-free languages**.

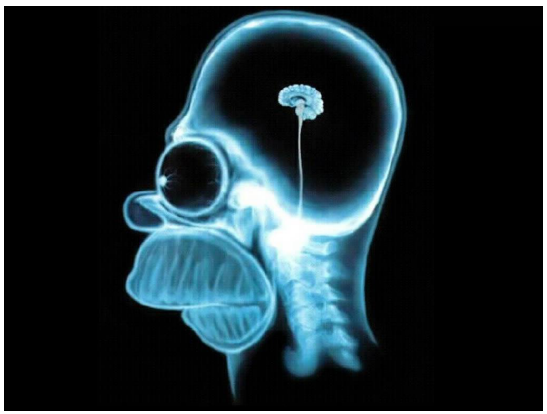
**Example**: The language of well-matched parentheses:

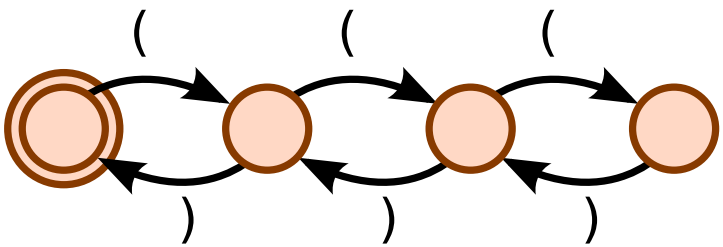


# Automata for Context-Free Languages

We have seen already that **finite automata** are not powerful enough to recognize **context-free languages**.

**Example**: The language of well-matched parentheses:



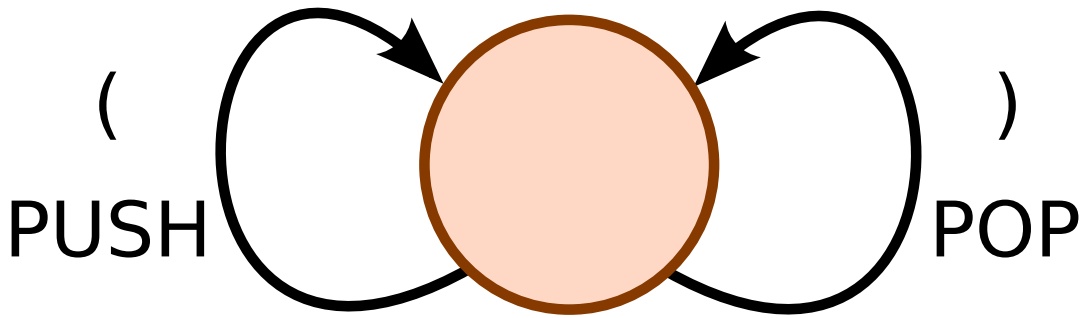



There are always some inputs where we **run out of memory**...

# Automata for Context-Free Languages

We have seen already that **finite automata** are not powerful enough to recognize **context-free languages**.

**Example**: The language of well-matched parentheses:

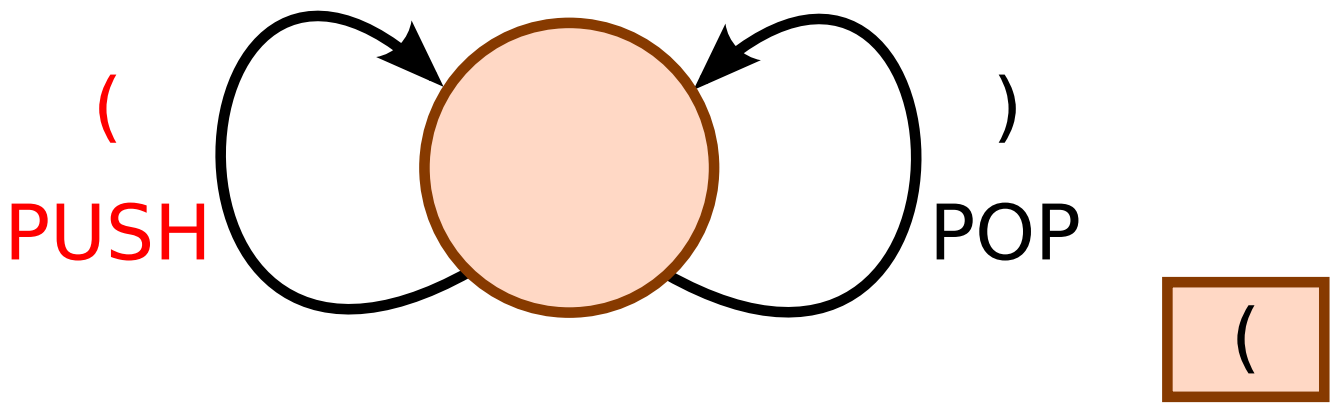



We can fix the problem by **introducing a stack**!

# Automata for Context-Free Languages

We have seen already that **finite automata** are not powerful enough to recognize **context-free languages**.

**Example**: The language of well-matched parentheses:

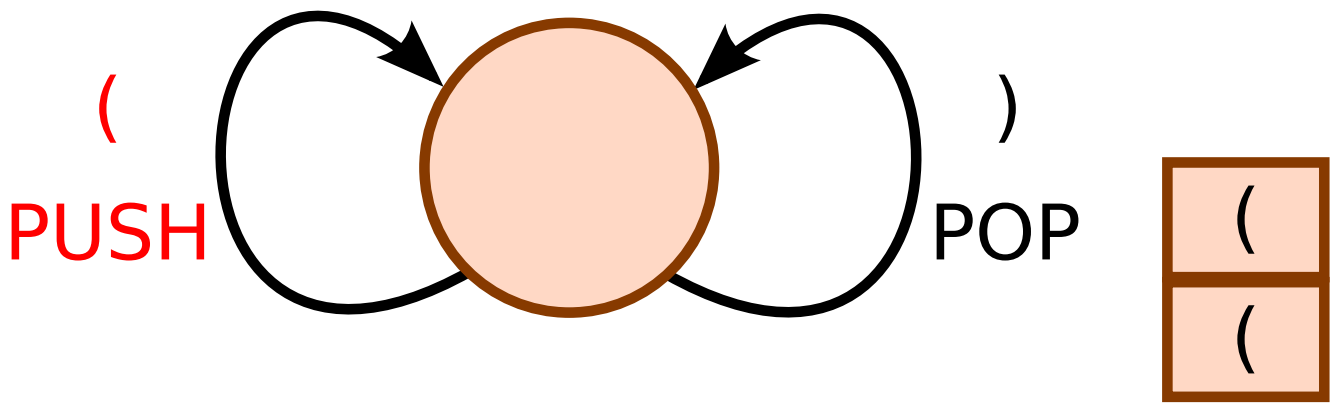



We can fix the problem by **introducing a stack**!

# Automata for Context-Free Languages

We have seen already that **finite automata** are not powerful enough to recognize **context-free languages**.

**Example**: The language of well-matched parentheses:

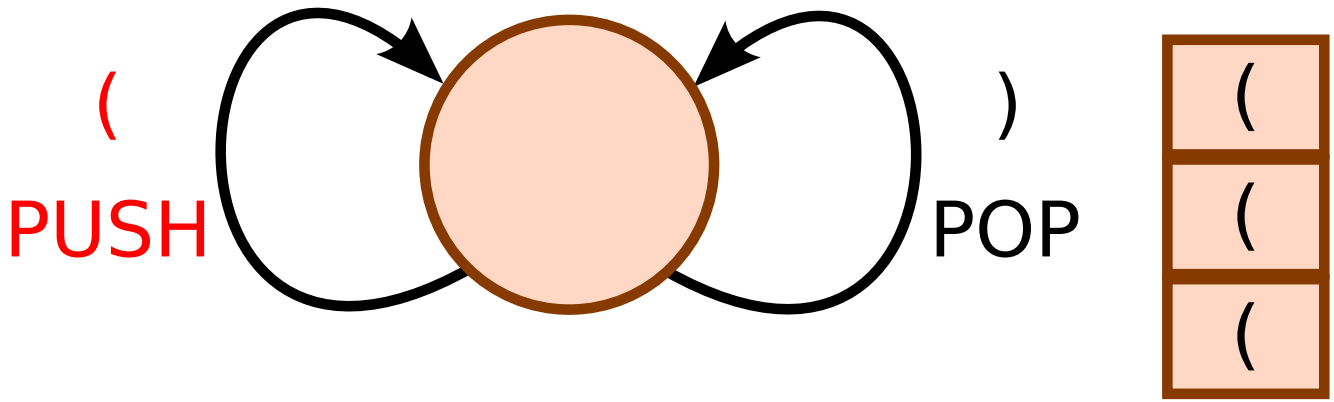



We can fix the problem by **introducing a stack**!

# Automata for Context-Free Languages

We have seen already that **finite automata** are not powerful enough to recognize **context-free languages**.

**Example**: The language of well-matched parentheses:

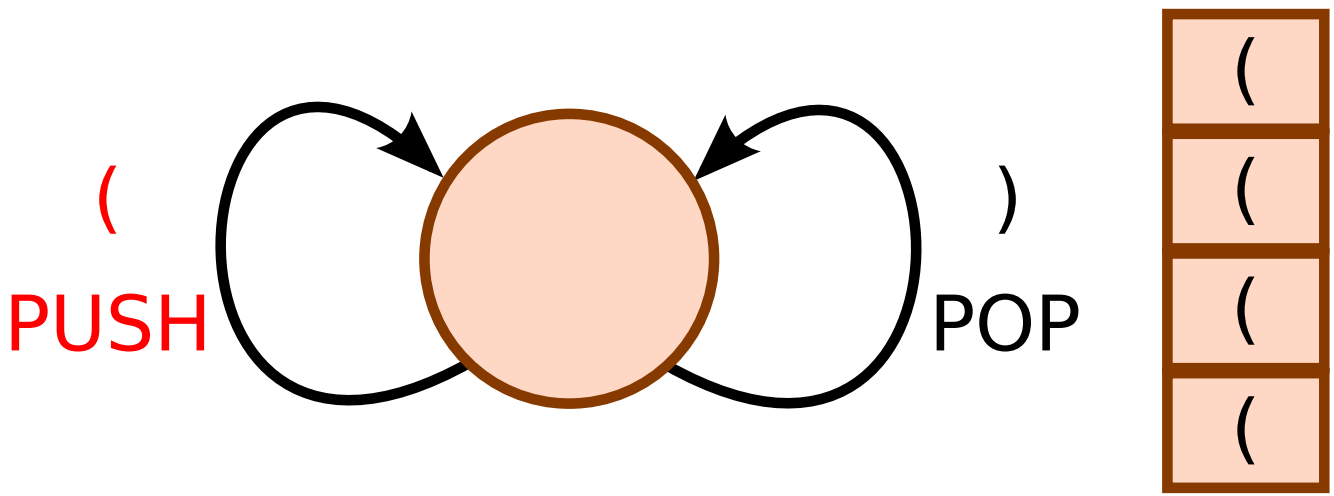



We can fix the problem by **introducing a stack**!

# Automata for Context-Free Languages

We have seen already that **finite automata** are not powerful enough to recognize **context-free languages**.

**Example**: The language of well-matched parentheses:

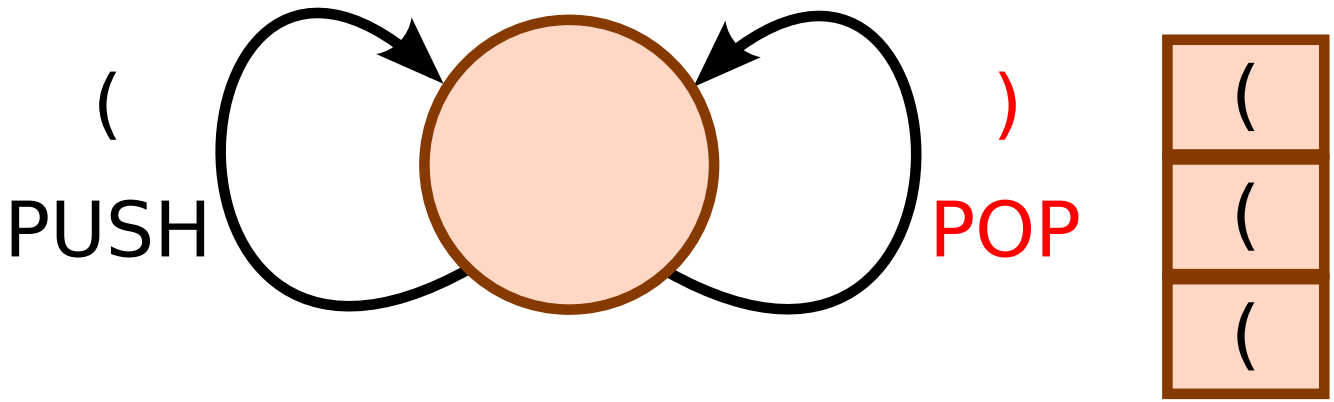



We can fix the problem by **introducing a stack**!

# Automata for Context-Free Languages

We have seen already that **finite automata** are not powerful enough to recognize **context-free languages**.

**Example**: The language of well-matched parentheses:

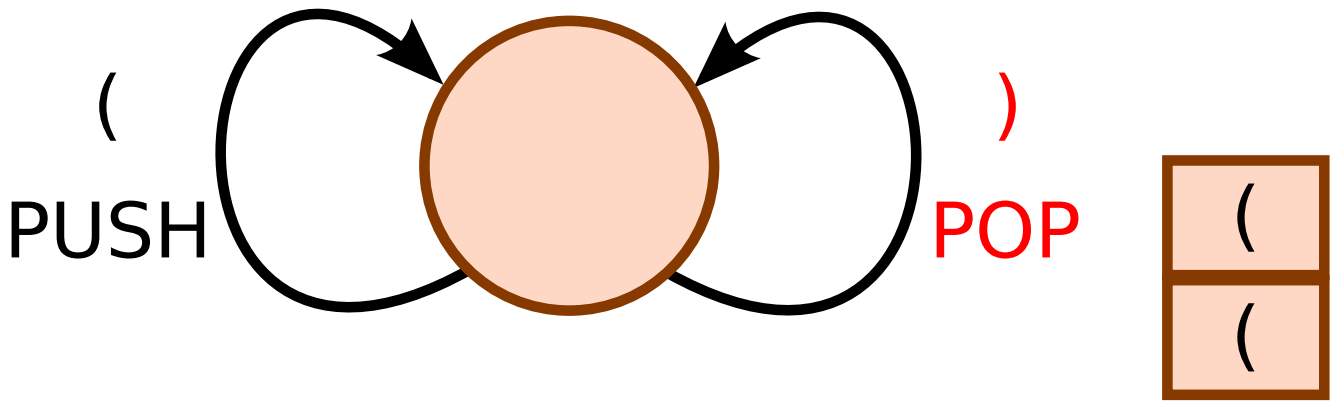



We can fix the problem by **introducing a stack**!

# Automata for Context-Free Languages

We have seen already that **finite automata** are not powerful enough to recognize **context-free languages**.

**Example**: The language of well-matched parentheses:

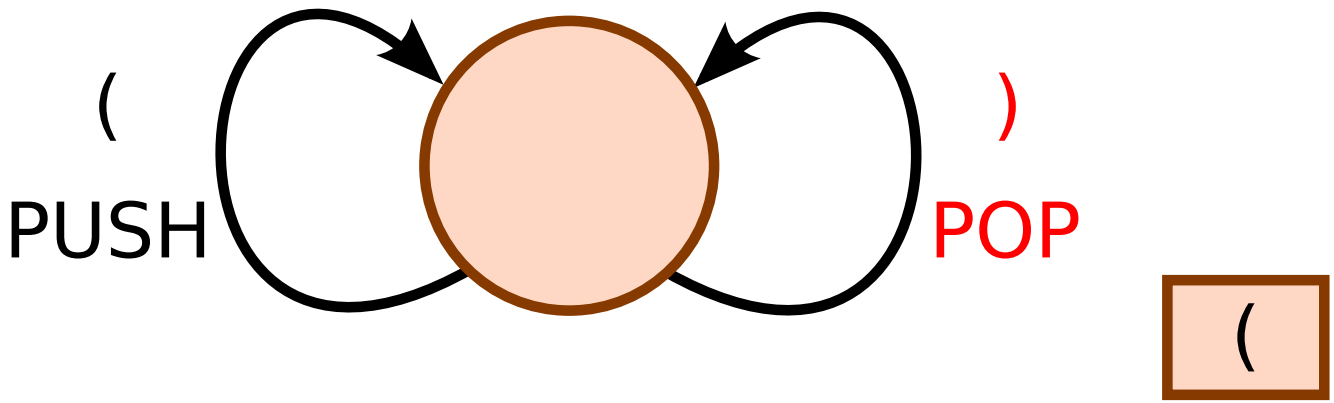



We can fix the problem by **introducing a stack**!

# Automata for Context-Free Languages

We have seen already that **finite automata** are not powerful enough to recognize **context-free languages**.

**Example**: The language of well-matched parentheses:

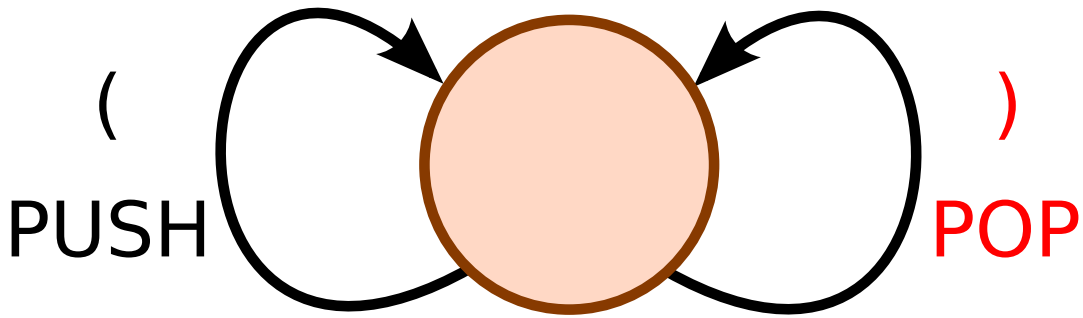



We can fix the problem by **introducing a stack**!

# Automata for Context-Free Languages

We have seen already that **finite automata** are not powerful enough to recognize **context-free languages**.

**Example**: The language of well-matched parentheses:

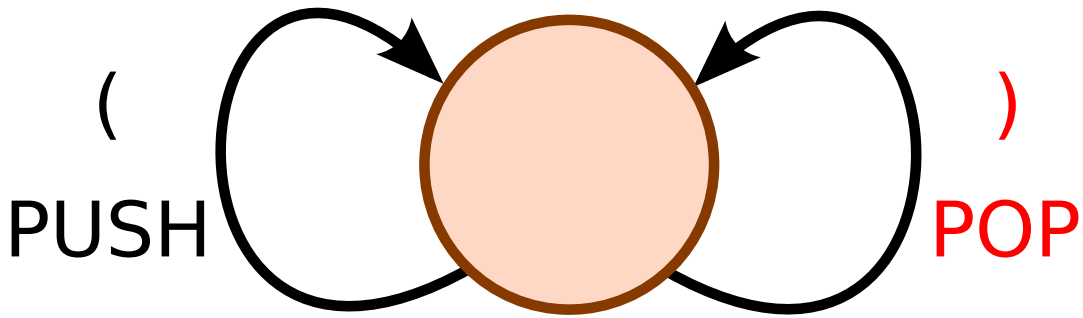



We can fix the problem by **introducing a stack**!

# Automata for Context-Free Languages

We have seen already that **finite automata** are not powerful enough to recognize **context-free languages**.

**Example**: The language of well-matched parentheses:




We can fix the problem by **introducing a stack**!

The stack can grow **as large as we need it to**!

# Pushdown Automata

# Pushdown Automata

A **pushdown automaton** is a seven-tuple $P = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$ where:

# Pushdown Automata

A **pushdown automaton** is a seven-tuple $P = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$ where:

- $Q$ is a finite set of **states**.

# Pushdown Automata

A **pushdown automaton** is a seven-tuple $P = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$ where:

- $Q$ is a finite set of **states**.
- $\Sigma$ is a finite set of **input symbols** (the alphabet).

# Pushdown Automata

A **pushdown automaton** is a seven-tuple $P = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$ where:

- $Q$ is a finite set of **states**.
- $\Sigma$ is a finite set of **input symbols** (the alphabet).
- $\Gamma$ is a finite set of **stack symbols** (things we can store on the stack).

# Pushdown Automata

A **pushdown automaton** is a seven-tuple $P = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$ where:

- $Q$ is a finite set of **states**.
- $\Sigma$ is a finite set of **input symbols** (the alphabet).
- $\Gamma$ is a finite set of **stack symbols** (things we can store on the stack).
- $\delta$ is the **transition relation**.

# Pushdown Automata

A **pushdown automaton** is a seven-tuple $P = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$ where:

- $Q$ is a finite set of **states**.
- $\Sigma$ is a finite set of **input symbols** (the alphabet).
- $\Gamma$ is a finite set of **stack symbols** (things we can store on the stack).
- $\delta$ is the **transition relation**.
- $q_0 \in Q$ is the **start state**.

# Pushdown Automata

A **pushdown automaton** is a seven-tuple $P = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$ where:

- $Q$ is a finite set of **states**.
- $\Sigma$ is a finite set of **input symbols** (the alphabet).
- $\Gamma$ is a finite set of **stack symbols** (things we can store on the stack).
- $\delta$ is the **transition relation**.
- $q_0 \in Q$ is the **start state**.
- $Z_0 \in \Gamma$ is the **start symbol** on the stack.

# Pushdown Automata

A **pushdown automaton** is a seven-tuple $P = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$ where:

- $Q$ is a finite set of **states**.
- $\Sigma$ is a finite set of **input symbols** (the alphabet).
- $\Gamma$ is a finite set of **stack symbols** (things we can store on the stack).
- $\delta$ is the **transition relation**.
- $q_0 \in Q$ is the **start state**.
- $Z_0 \in \Gamma$ is the **start symbol** on the stack.
- $F \subseteq Q$ are the **final states** (accepting states).

# Pushdown Automata

The transition relation $\delta \subset Q \times (\Sigma \cup \{\epsilon\}) \times \Gamma \times Q \times \Gamma^*$:

# Pushdown Automata

The transition relation $\delta \subset Q \times (\Sigma \cup \{\epsilon\}) \times \Gamma \times Q \times \Gamma^*$:

- Relates a triple $(q, a, X)$ of a **state**, an **input symbol** and a **stack symbol**...

# Pushdown Automata

The transition relation $\delta \subset Q \times (\Sigma \cup \{\epsilon\}) \times \Gamma \times Q \times \Gamma^*$:

- Relates a triple $(q, a, X)$ of a **state**, an **input symbol** and a **stack symbol**...
- with a pair $(p, \gamma)$: a **new state** and a **sequence of stack symbols**.

# Pushdown Automata

The transition relation $\delta \subset Q \times (\Sigma \cup \{\epsilon\}) \times \Gamma \times Q \times \Gamma^*$:

- Relates a triple $(q, a, X)$ of a **state**, an **input symbol** and a **stack symbol**...
- with a pair $(p, \gamma)$: a **new state** and a **sequence of stack symbols**.

This means that if we are in **state** $q$ and we read an **input** $a$ and $X$ **is at the top of the stack**,

# Pushdown Automata

The transition relation $\delta \subset Q \times (\Sigma \cup \{\epsilon\}) \times \Gamma \times Q \times \Gamma^*$:

- Relates a triple $(q, a, X)$ of a **state**, an **input symbol** and a **stack symbol**...
- with a pair $(p, \gamma)$: a **new state** and a **sequence of stack symbols**.

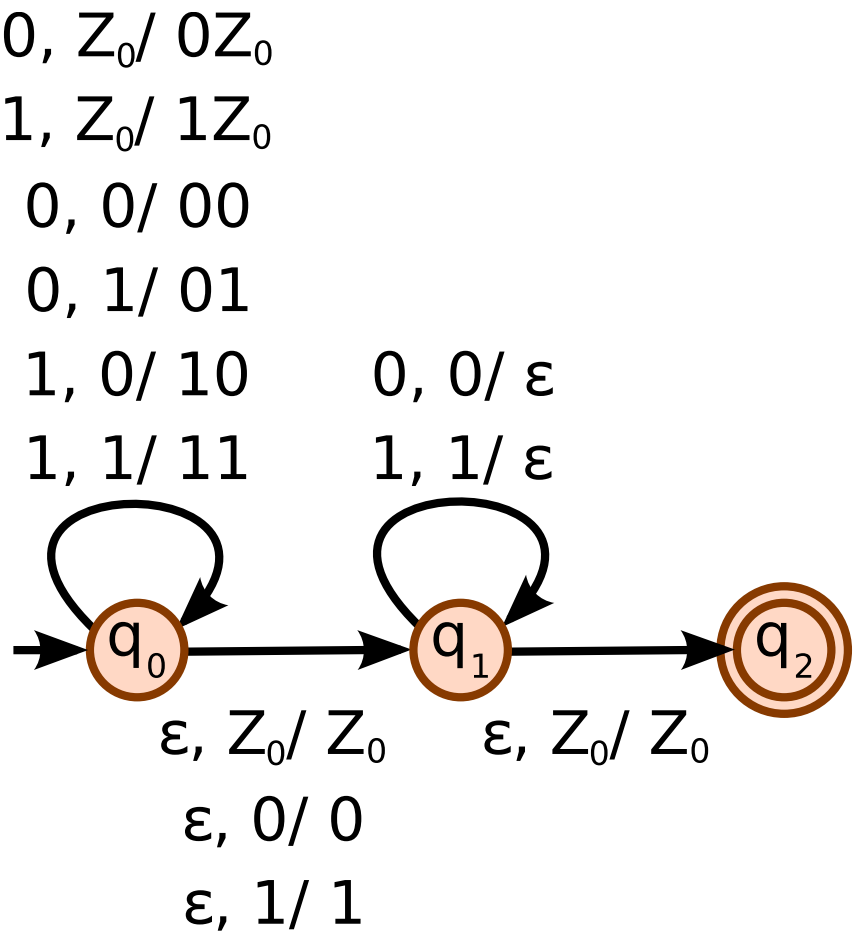
This means that if we are in **state** $q$ and we read an **input** $a$ and $X$ **is at the top of the stack**, then we **move to state** $p$ and **replace** $X$ **with** $\gamma$.

# Pushdown Automata

**Example**: strings $ww^R$ over the alphabet $\Sigma = \{0, 1\}$ (a subset of palindromes over $\Sigma$).

# Pushdown Automata

**Example**: strings $ww^R$ over the alphabet $\Sigma = \{0, 1\}$ (a subset of palindromes over $\Sigma$).

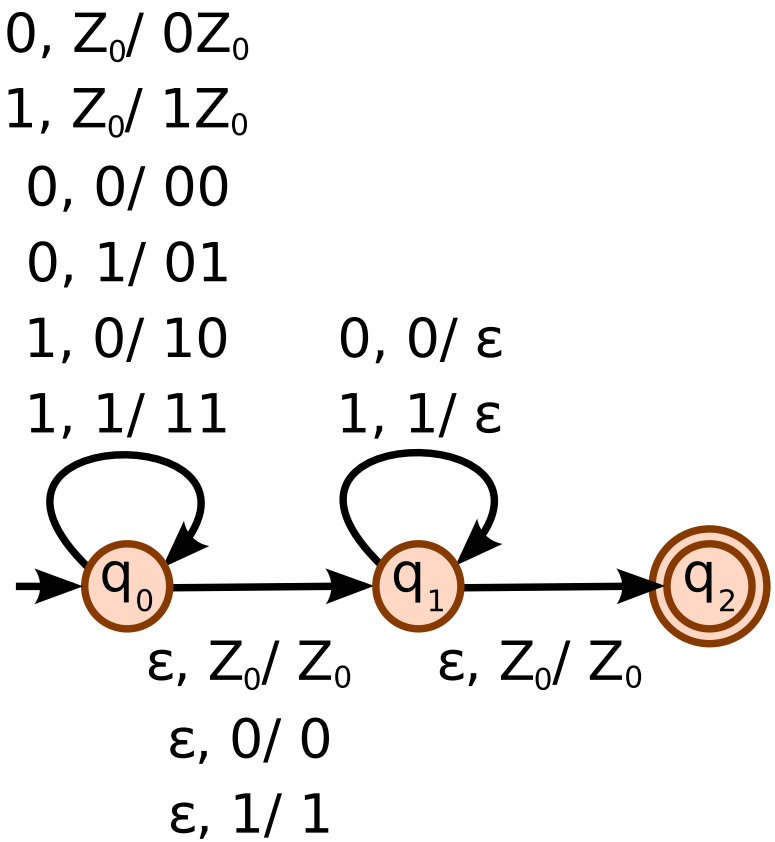


# Pushdown Automata

**Example**: strings $ww^R$ over the alphabet $\Sigma = \{0, 1\}$ (a subset of palindromes over $\Sigma$).

# Pushdown Automata

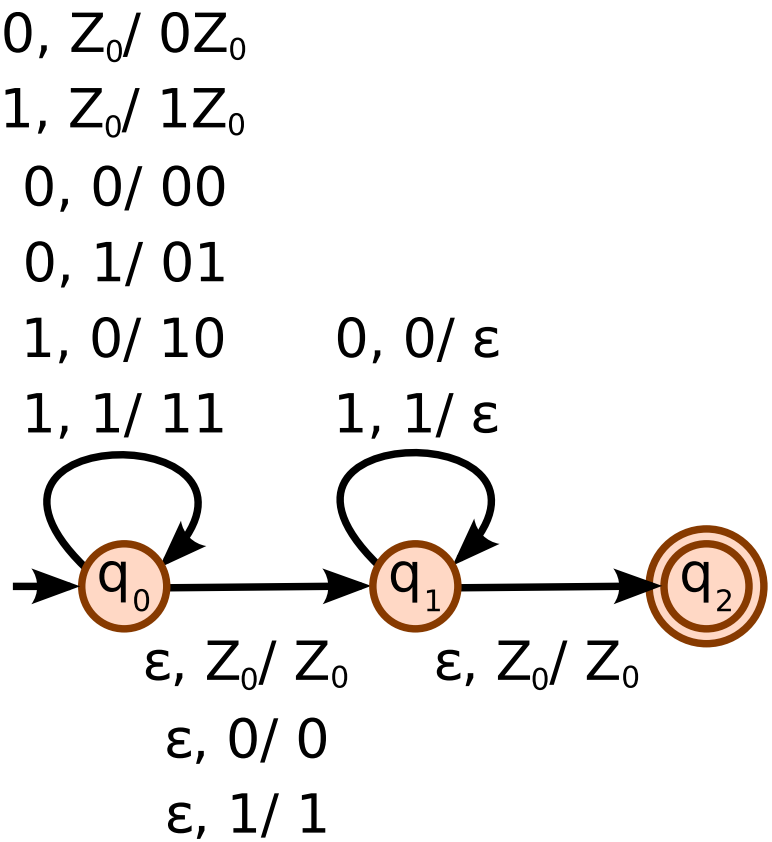
**Example**: strings $ww^R$ over the alphabet $\Sigma = \{0, 1\}$ (a subset of palindromes over $\Sigma$).




The notation $a, X/\gamma$ means we can make the transition if **we read an input** $a$,

# Pushdown Automata

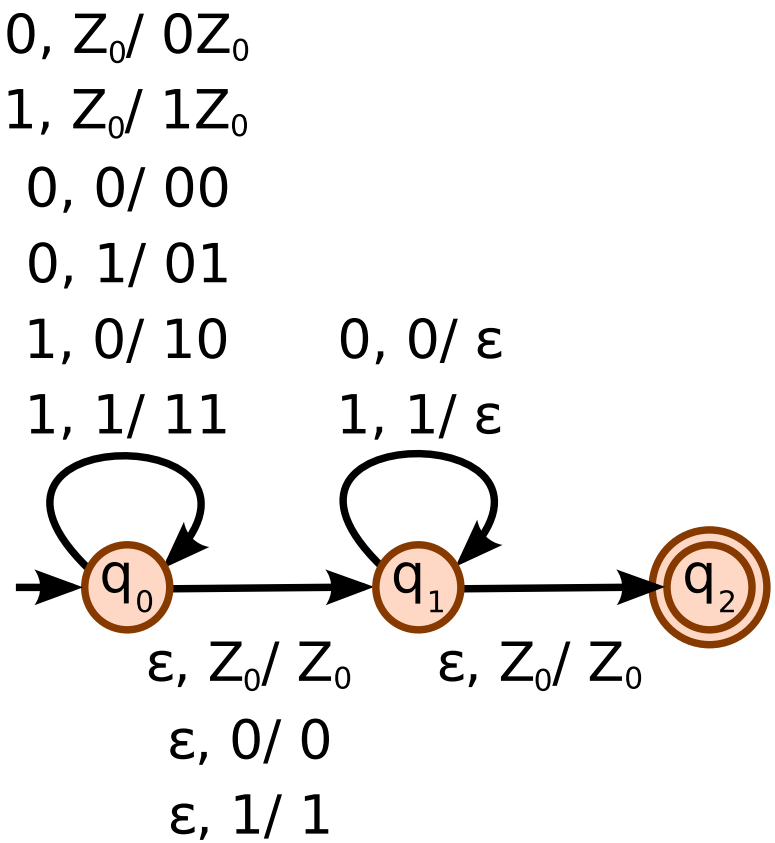
**Example**: strings $ww^R$ over the alphabet $\Sigma = \{0, 1\}$ (a subset of palindromes over $\Sigma$).




The notation $a, X/\gamma$ means we can make the transition if **we read an input** $a$, and $X$ **is the symbol at the top of the stack**.

# Pushdown Automata

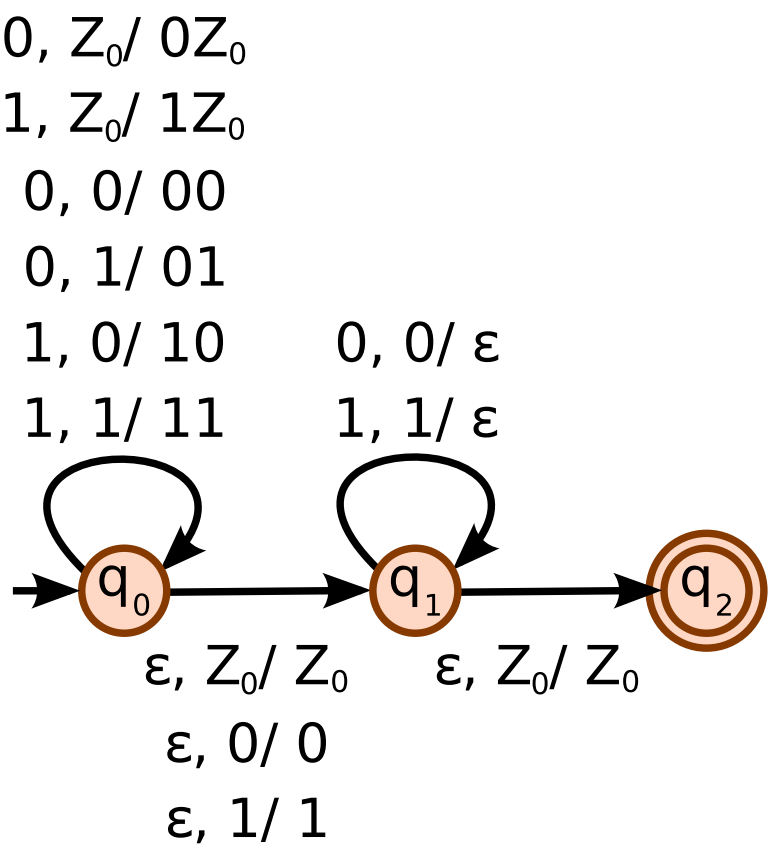
**Example**: strings $ww^R$ over the alphabet $\Sigma = \{0, 1\}$ (a subset of palindromes over $\Sigma$).




The notation $a, X/\gamma$ means we can make the transition if **we read an input** $a$, and $X$ **is the symbol at the top of the stack**. If we make the transition, **we replace** $X$ **with** $\gamma$**.**

# Pushdown Automata

**Example**: strings $ww^R$ over the alphabet $\Sigma = \{0, 1\}$ (a subset of palindromes over $\Sigma$).




$$
\begin{aligned}
\delta(q_0, 0, Z_0) &= \{(q_0, 0Z_0)\} \\
\delta(q_0, 1, Z_0) &= \{(q_0, 1Z_0)\} \\
\delta(q_0, 0, 0) &= \{(q_0, 00)\} \\
\delta(q_0, 0, 1) &= \{(q_0, 01)\} \\
\delta(q_0, 1, 0) &= \{(q_0, 10)\} \\
\delta(q_0, 1, 1) &= \{(q_0, 11)\} \\
\delta(q_0, \epsilon, Z_0) &= \{(q_1, Z_0)\} \\
\delta(q_0, \epsilon, 0) &= \{(q_1, 0)\} \\
\delta(q_0, \epsilon, 1) &= \{(q_1, 1)\} \\
\delta(q_1, 0, 0) &= \{(q_1, \epsilon)\} \\
\delta(q_1, 1, 1) &= \{(q_1, \epsilon)\} \\
\delta(q_1, \epsilon, Z_0) &= \{(q_2, Z_0)\}
\end{aligned}
$$

The notation $a, X/\gamma$ means we can make the transition if **we read an input $a$**, and **$X$ is the symbol at the top of the stack**. If we make the transition, **we replace $X$ with $\gamma$.**

# Pushdown Automata

There are different **types of stack operation**:

# Pushdown Automata

There are different **types of stack operation**:

- $X/X$ — **no change**.

# Pushdown Automata

There are different **types of stack operation**:

- $X/X$ — **no change**.
- $X/YX$ — **push** $Y$ onto the stack.

# Pushdown Automata

There are different **types of stack operation**:

- $X/X$ — **no change**.
- $X/YX$ — **push** $Y$ onto the stack.
- $X/Y$ — **replace** $X$ with $Y$.

# Pushdown Automata

There are different **types of stack operation**:

- $X/X$ — **no change**.
- $X/YX$ — **push** $Y$ onto the stack.
- $X/Y$ — **replace** $X$ with $Y$.
- $X/\epsilon$ — **pop** $X$ from the stack.

# Pushdown Automata

There are different **types of stack operation**:

- $X/X$ — **no change**.
- $X/YX$ — **push** $Y$ onto the stack.
- $X/Y$ — **replace** $X$ with $Y$.
- $X/\epsilon$ — **pop** $X$ from the stack.

To know the **configuration** of a PDA we need to know both the **current state** and the **contents of the stack**.

# Pushdown Automata

There are different **types of stack operation**:

- $X/X$ — **no change**.
- $X/YX$ — **push** $Y$ onto the stack.
- $X/Y$ — **replace** $X$ with $Y$.
- $X/\epsilon$ — **pop** $X$ from the stack.

To know the **configuration** of a PDA we need to know both the **current state** and the **contents of the stack**.

We often use the **instantaneous description (ID)**, which is a triple $(q, w, \gamma)$:

- $q$ is the current state.
- $w$ is the remaining input.
- $\gamma$ is the content of the stack.

# Pushdown Automata

**Example**

0, $Z_0$/ 0$Z_0$

1, $Z_0$/ 1$Z_0$

0, 0/ 00

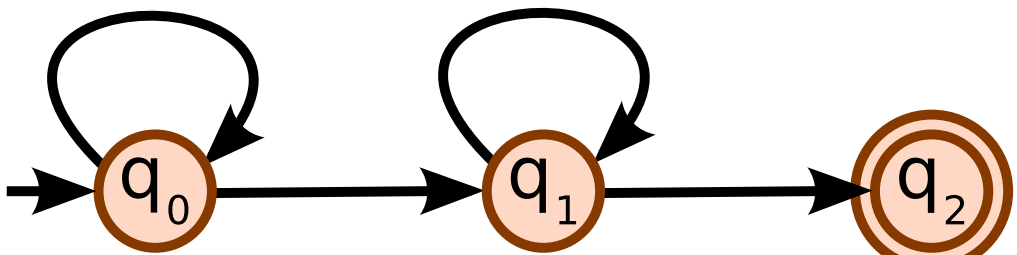
0, 1/ 01

1, 0/ 10

1, 1/ 11

0, 0/ ε

1, 1/ ε




ε, $Z_0$/ $Z_0$

ε, $Z_0$/ $Z_0$

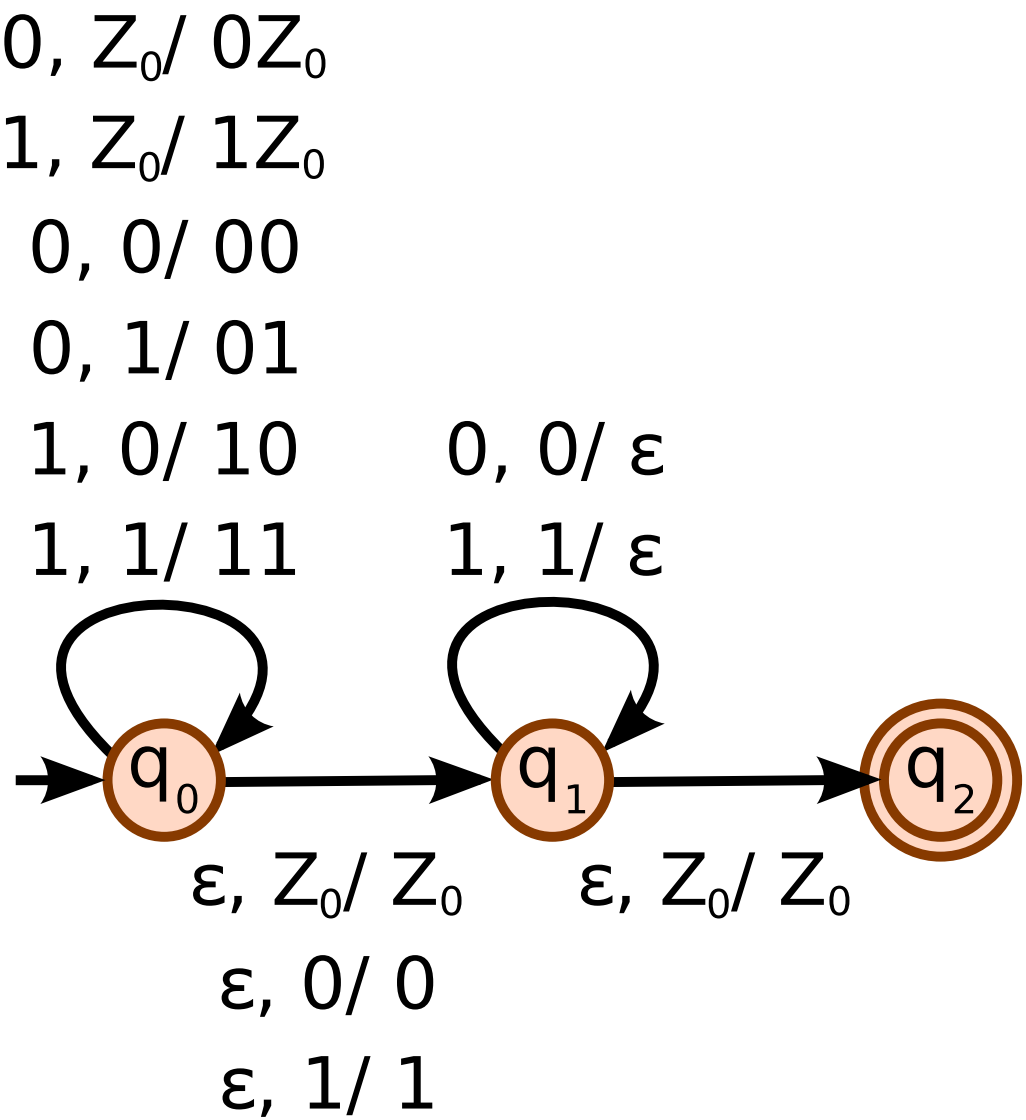
ε, 0/ 0

ε, 1/ 1

$(q_0, 0110, Z_0)$

# Pushdown Automata

**Example**




$$\begin{aligned} & (q_0, 0110, Z_0) \\ \vdash \quad & (q_0, 110, 0Z_0) \end{aligned}$$

# Pushdown Automata

**Example**

0, $Z_0$/ 0$Z_0$

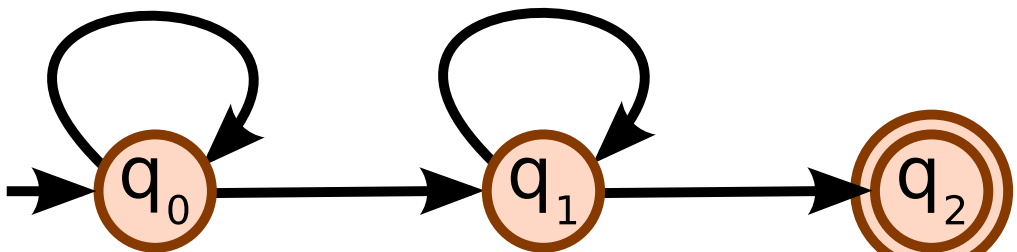
1, $Z_0$/ 1$Z_0$

0, 0/ 00

0, 1/ 01

1, 0/ 10

1, 1/ 11




0, 0/ ε

1, 1/ ε

ε, $Z_0$/ $Z_0$

ε, $Z_0$/ $Z_0$

ε, 0/ 0

ε, 1/ 1

$$\begin{aligned} & (q_0, 0110, Z_0) \\ \vdash\ & (q_0, 110, 0Z_0) \\ \vdash\ & (q_0, 10, 10Z_0) \end{aligned}$$

# Pushdown Automata

**Example**

0, $Z_0$/ 0$Z_0$

1, $Z_0$/ 1$Z_0$

0, 0/ 00

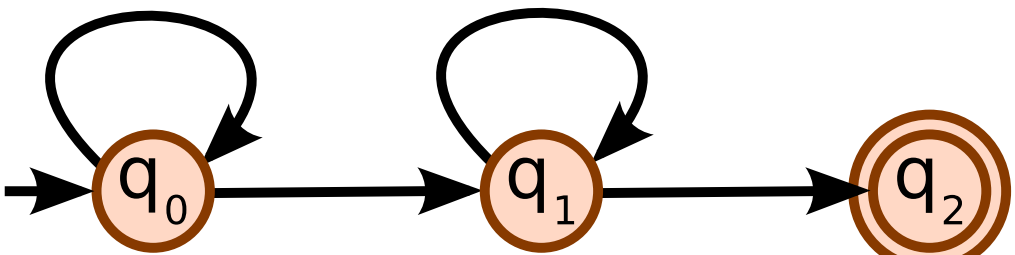
0, 1/ 01

1, 0/ 10

1, 1/ 11

0, 0/ ε

1, 1/ ε




ε, $Z_0$/ $Z_0$

ε, $Z_0$/ $Z_0$

ε, 0/ 0

ε, 1/ 1

$$
\begin{aligned}
& (q_0, 0110, Z_0) \\
\vdash\ & (q_0, 110, 0Z_0) \\
\vdash\ & (q_0, 10, 10Z_0) \\
\vdash\ & (q_0, 0, 110Z_0)
\end{aligned}
$$

# Pushdown Automata

**Example**

0, $Z_0$/ 0$Z_0$

1, $Z_0$/ 1$Z_0$

0, 0/ 00

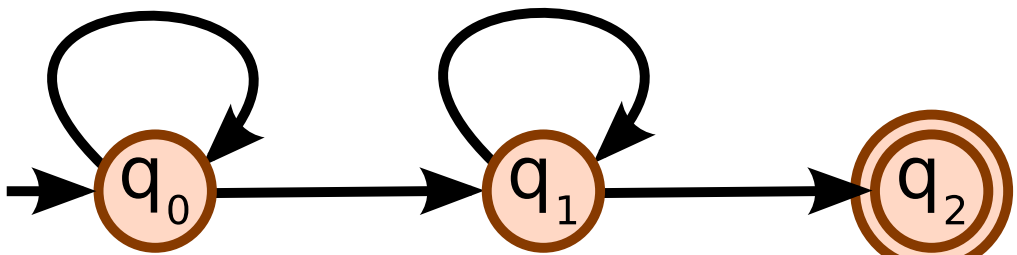
0, 1/ 01

1, 0/ 10

1, 1/ 11

0, 0/ ε

1, 1/ ε




ε, $Z_0$/ $Z_0$

ε, $Z_0$/ $Z_0$

ε, 0/ 0

ε, 1/ 1

$$
\begin{aligned}
& (q_0, 0110, Z_0) \\
\vdash\ & (q_0, 110, 0Z_0) \\
\vdash\ & (q_0, 10, 10Z_0) \\
\vdash\ & (q_0, 0, 110Z_0) \\
\vdash\ & (q_0, \epsilon, 0110Z_0)
\end{aligned}
$$

# Pushdown Automata

**Example**

0, $Z_0$/ 0$Z_0$

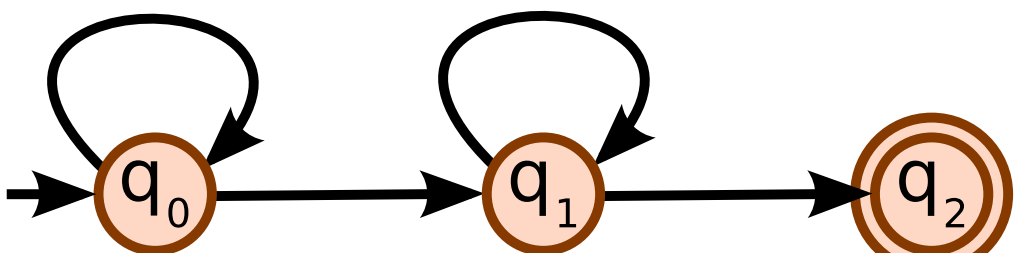
1, $Z_0$/ 1$Z_0$

0, 0/ 00

0, 1/ 01

1, 0/ 10

1, 1/ 11




0, 0/ ε

1, 1/ ε

ε, $Z_0$/ $Z_0$

ε, $Z_0$/ $Z_0$

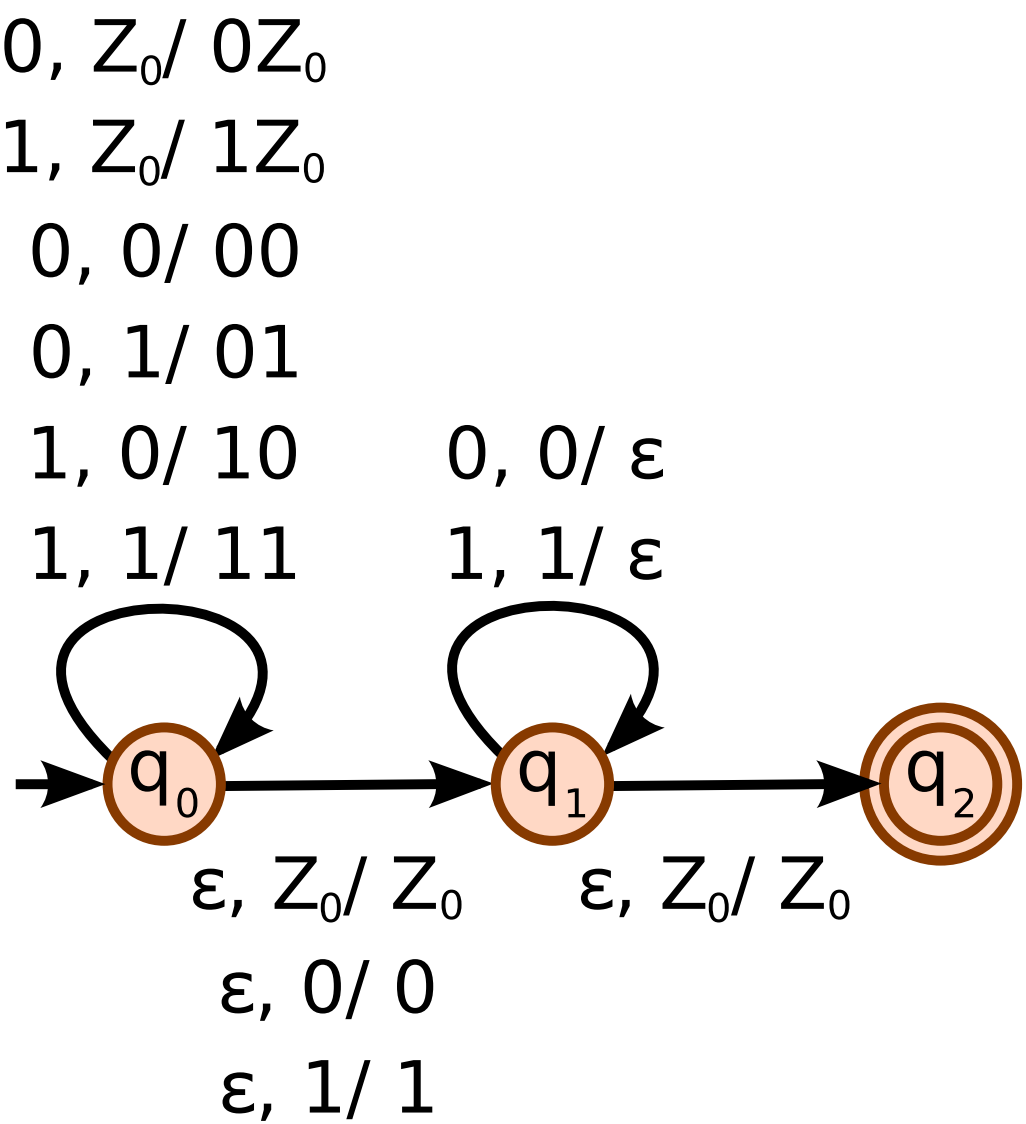
ε, 0/ 0

ε, 1/ 1

$$
\begin{aligned}
& (q_0, 0110, Z_0) \\
\vdash\ & (q_0, 110, 0Z_0) \\
\vdash\ & (q_0, 10, 10Z_0) \\
\vdash\ & (q_0, 0, 110Z_0) \\
\vdash\ & (q_0, \epsilon, 0110Z_0) \\
\vdash\ & (q_1, \epsilon, 0110Z_0)
\end{aligned}
$$

# Pushdown Automata

**Example**




$$
\begin{aligned}
& (q_0, 0110, Z_0) \\
\vdash\ & (q_0, 110, 0Z_0) \\
\vdash\ & (q_0, 10, 10Z_0) \\
\vdash\ & (q_0, 0, 110Z_0) \\
\vdash\ & (q_0, \epsilon, 0110Z_0) \\
\vdash\ & (q_1, \epsilon, 0110Z_0)
\end{aligned}
$$

**We are stuck!**

# Pushdown Automata

**Example**

0, $Z_0$/ 0$Z_0$

1, $Z_0$/ 1$Z_0$

0, 0/ 00

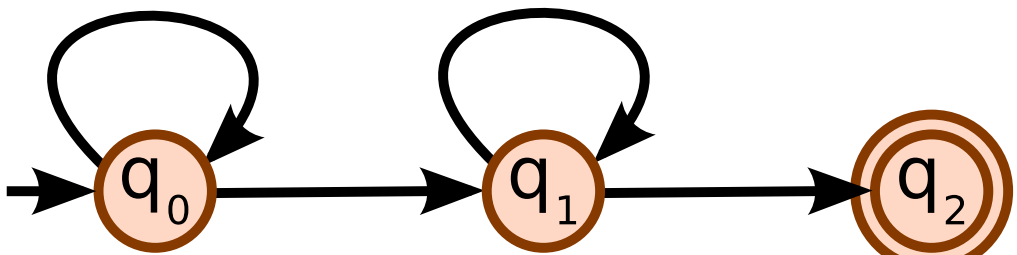
0, 1/ 01

1, 0/ 10

1, 1/ 11

0, 0/ ε

1, 1/ ε




ε, $Z_0$/ $Z_0$

ε, $Z_0$/ $Z_0$

ε, 0/ 0

ε, 1/ 1

$(q_0, 0110, Z_0)$

# Pushdown Automata

**Example**

0, $Z_0$/ 0$Z_0$

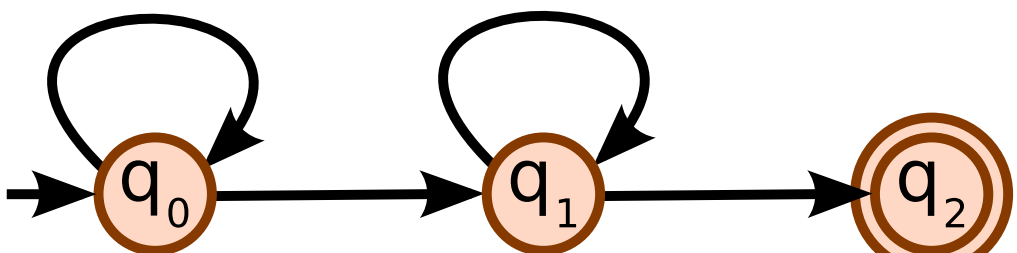
1, $Z_0$/ 1$Z_0$

0, 0/ 00

0, 1/ 01

1, 0/ 10

1, 1/ 11




0, 0/ ε

1, 1/ ε

ε, $Z_0$/ $Z_0$

ε, $Z_0$/ $Z_0$

ε, 0/ 0

ε, 1/ 1

$$\vdash \quad \begin{array}{l} (q_0, 0110, Z_0) \\ (q_1, 0110, Z_0) \end{array}$$

# Pushdown Automata

**Example**

0, $Z_0$/ 0$Z_0$

1, $Z_0$/ 1$Z_0$

0, 0/ 00

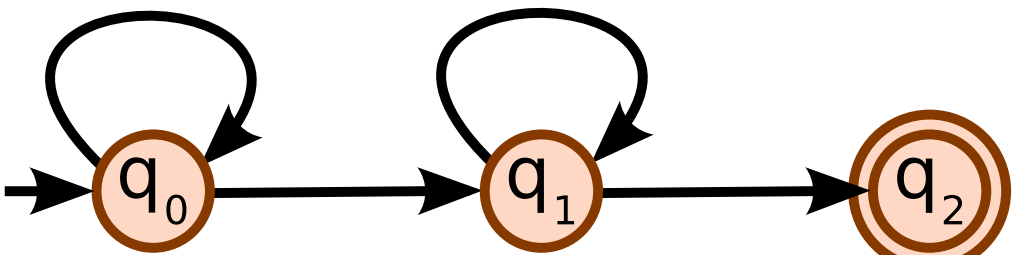
0, 1/ 01

1, 0/ 10

1, 1/ 11

0, 0/ ε

1, 1/ ε




ε, $Z_0$/ $Z_0$

ε, $Z_0$/ $Z_0$

ε, 0/ 0

ε, 1/ 1

$$\begin{aligned} & (q_0, 0110, Z_0) \\ \vdash\ & (q_1, 0110, Z_0) \\ \vdash\ & (q_2, 0110, Z_0) \end{aligned}$$

# Pushdown Automata

**Example**

$0, Z_0/\ 0Z_0$

$1, Z_0/\ 1Z_0$

$0, 0/\ 00$

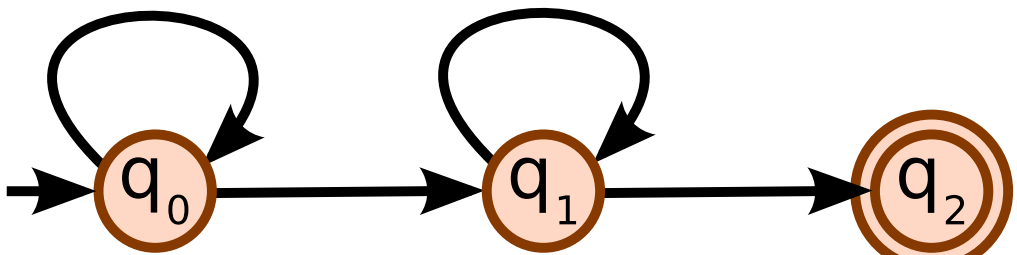
$0, 1/\ 01$

$1, 0/\ 10$

$1, 1/\ 11$

$0, 0/\ \varepsilon$

$1, 1/\ \varepsilon$




$\varepsilon, Z_0/\ Z_0$

$\varepsilon, Z_0/\ Z_0$

$\varepsilon, 0/\ 0$

$\varepsilon, 1/\ 1$

$$\begin{aligned} & (q_0, 0110, Z_0) \\ \vdash\ & (q_1, 0110, Z_0) \\ \vdash\ & (q_2, 0110, Z_0) \end{aligned}$$

**We are stuck!**

# Pushdown Automata

**Example**:

0, $Z_0$/ 0$Z_0$

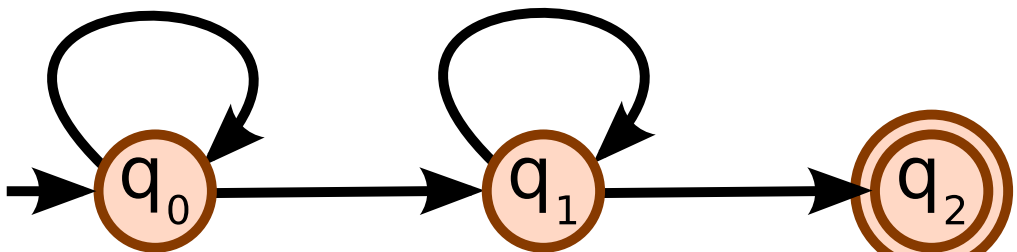
1, $Z_0$/ 1$Z_0$

0, 0/ 00

0, 1/ 01

1, 0/ 10

1, 1/ 11




0, 0/ ε

1, 1/ ε

ε, $Z_0$/ $Z_0$

ε, $Z_0$/ $Z_0$

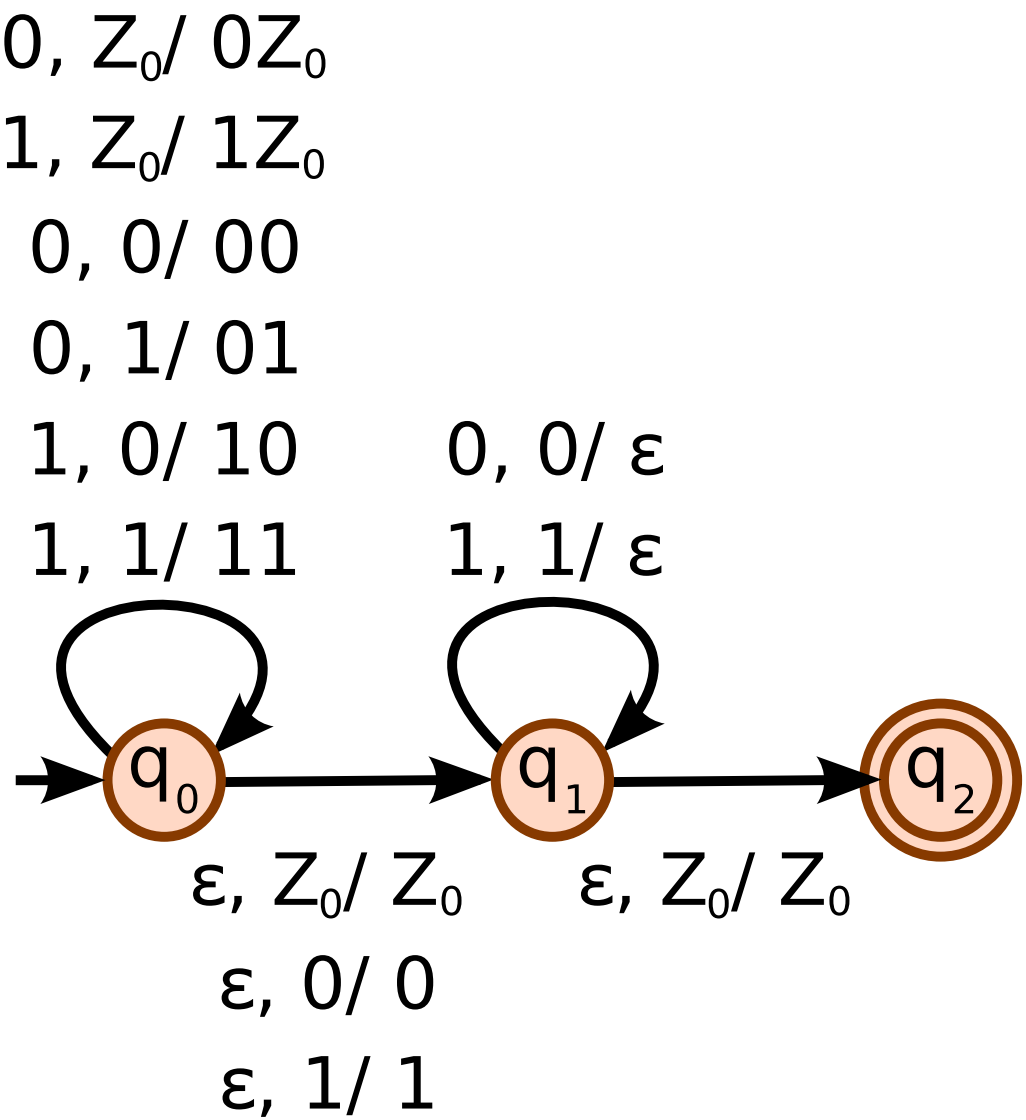
ε, 0/ 0

ε, 1/ 1

$(q_0, 0110, Z_0)$

# Pushdown Automata

**Example**:




$$\begin{aligned} & (q_0, 0110, Z_0) \\ \vdash \quad & (q_0, 110, 0Z_0) \end{aligned}$$

# Pushdown Automata

**Example**:

0, $Z_0$/ 0$Z_0$

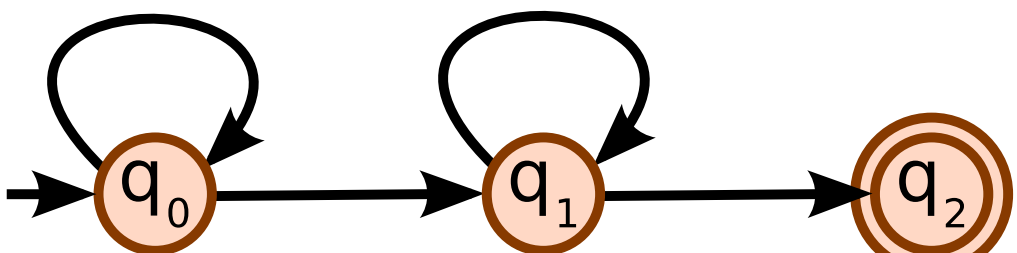
1, $Z_0$/ 1$Z_0$

0, 0/ 00

0, 1/ 01

1, 0/ 10

1, 1/ 11




0, 0/ ε

1, 1/ ε

ε, $Z_0$/ $Z_0$

ε, $Z_0$/ $Z_0$

ε, 0/ 0

ε, 1/ 1

$$
\begin{aligned}
& (q_0, 0110, Z_0) \\
\vdash\ & (q_0, 110, 0Z_0) \\
\vdash\ & (q_0, 10, 10Z_0)
\end{aligned}
$$

# Pushdown Automata

**Example**:

0, $Z_0$/ 0$Z_0$

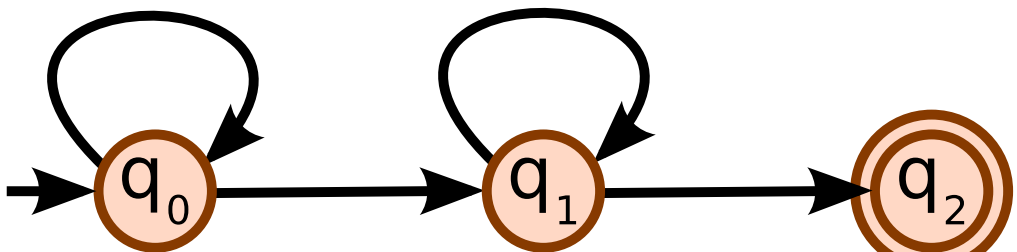
1, $Z_0$/ 1$Z_0$

0, 0/ 00

0, 1/ 01

1, 0/ 10

1, 1/ 11




0, 0/ ε

1, 1/ ε

ε, $Z_0$/ $Z_0$

ε, $Z_0$/ $Z_0$

ε, 0/ 0

ε, 1/ 1

$$
\begin{aligned}
& (q_0, 0110, Z_0) \\
\vdash\ & (q_0, 110, 0Z_0) \\
\vdash\ & (q_0, 10, 10Z_0) \\
\vdash\ & (q_1, 10, 10Z_0)
\end{aligned}
$$

# Pushdown Automata

**Example**:

0, $Z_0$/ 0$Z_0$

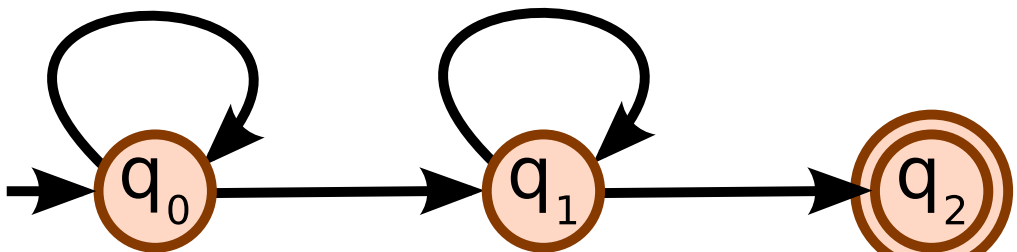
1, $Z_0$/ 1$Z_0$

0, 0/ 00

0, 1/ 01

1, 0/ 10

1, 1/ 11




0, 0/ ε

1, 1/ ε

ε, $Z_0$/ $Z_0$

ε, $Z_0$/ $Z_0$

ε, 0/ 0

ε, 1/ 1

$$
\begin{aligned}
& (q_0, 0110, Z_0) \\
\vdash\ & (q_0, 110, 0Z_0) \\
\vdash\ & (q_0, 10, 10Z_0) \\
\vdash\ & (q_1, 10, 10Z_0) \\
\vdash\ & (q_1, 0, 0Z_0)
\end{aligned}
$$

# Pushdown Automata

**Example**:

0, $Z_0$/ 0$Z_0$

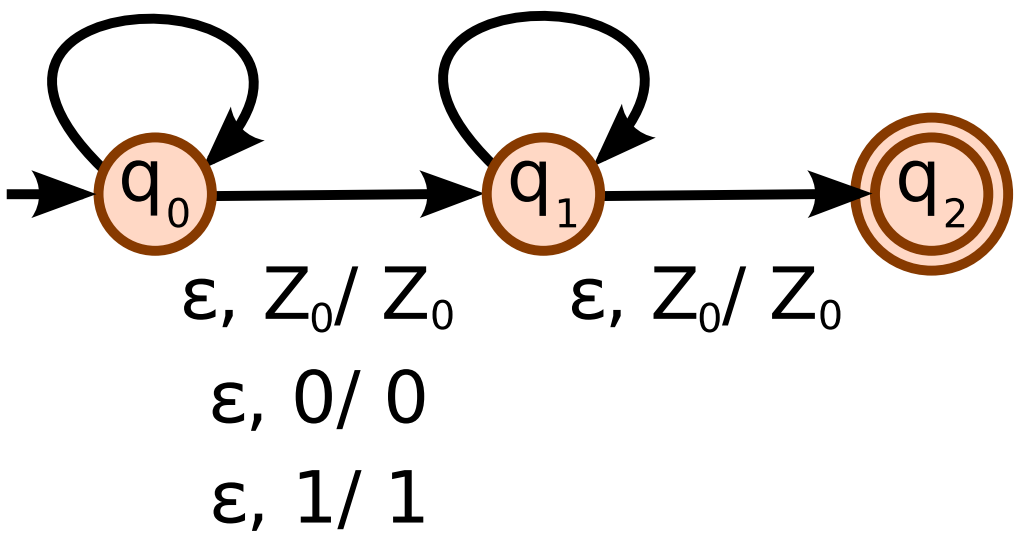
1, $Z_0$/ 1$Z_0$

0, 0/ 00

0, 1/ 01

1, 0/ 10

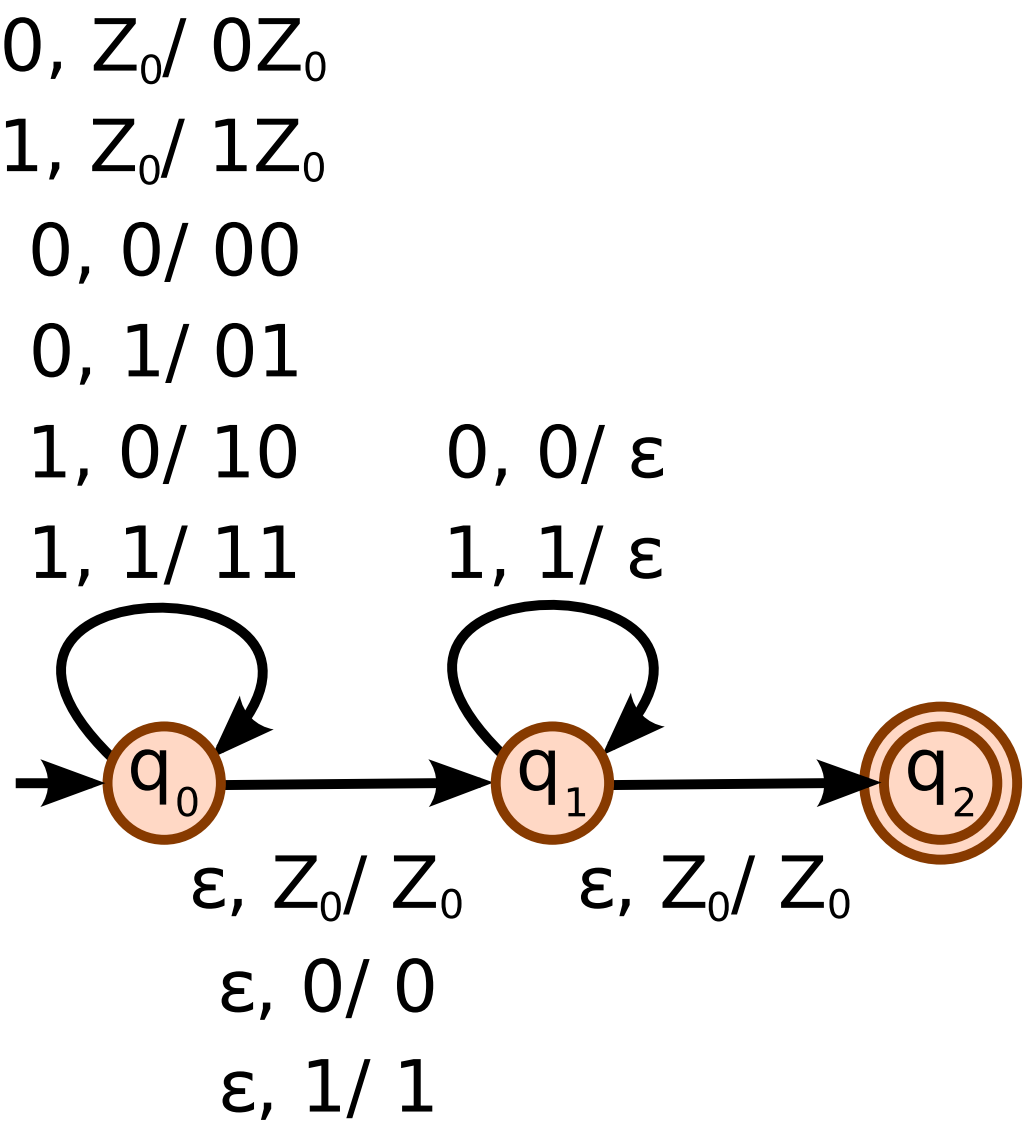
1, 1/ 11




$$
\begin{aligned}
& (q_0, 0110, Z_0) \\
\vdash\ & (q_0, 110, 0Z_0) \\
\vdash\ & (q_0, 10, 10Z_0) \\
\vdash\ & (q_1, 10, 10Z_0) \\
\vdash\ & (q_1, 0, 0Z_0) \\
\vdash\ & (q_1, \epsilon, Z_0)
\end{aligned}
$$

# Pushdown Automata

**Example**:




$$
\begin{aligned}
& (q_0, 0110, Z_0) \\
\vdash\ & (q_0, 110, 0Z_0) \\
\vdash\ & (q_0, 10, 10Z_0) \\
\vdash\ & (q_1, 10, 10Z_0) \\
\vdash\ & (q_1, 0, 0Z_0) \\
\vdash\ & (q_1, \epsilon, Z_0) \\
\vdash\ & (q_2, \epsilon, Z_0)
\end{aligned}
$$

# Pushdown Automata

**Example**:

0, $Z_0$/ 0$Z_0$

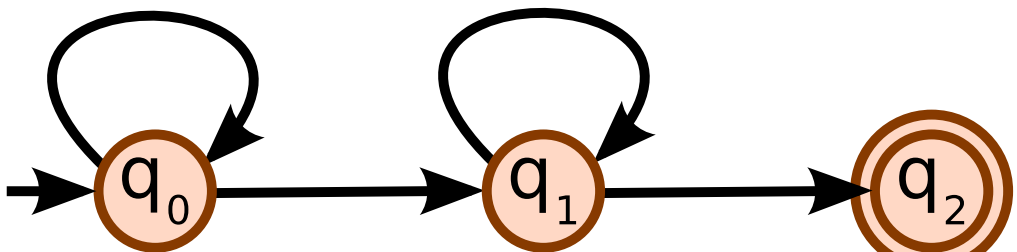
1, $Z_0$/ 1$Z_0$

0, 0/ 00

0, 1/ 01

1, 0/ 10

1, 1/ 11




0, 0/ ε

1, 1/ ε

ε, $Z_0$/ $Z_0$

ε, $Z_0$/ $Z_0$

ε, 0/ 0

ε, 1/ 1

$$
\begin{aligned}
& (q_0, 0110, Z_0) \\
\vdash\ & (q_0, 110, 0Z_0) \\
\vdash\ & (q_0, 10, 10Z_0) \\
\vdash\ & (q_1, 10, 10Z_0) \\
\vdash\ & (q_1, 0, 0Z_0) \\
\vdash\ & (q_1, \epsilon, Z_0) \\
\vdash\ & (q_2, \epsilon, Z_0)
\end{aligned}
$$

**We accept!**

# Exercise 4.1

Consider the PDA
$P = (\{start, guess, check, right!\}, \{0, 1\}, \{0, 1, S, Z_0\}, \delta, start, Z_0, \{right!\})$
where

$$\begin{aligned}
\delta(start, \epsilon, Z_0) &= \{(guess, SZ_0)\} \\
\delta(guess, \epsilon, S) &= \{(guess, S0), (guess, S1), (check, \epsilon)\} \\
\delta(check, 0, 0) &= \{(check, \epsilon)\} \\
\delta(check, 1, 1) &= \{(check, \epsilon)\} \\
\delta(check, \epsilon, Z_0) &= \{(right!, \epsilon)\}
\end{aligned}$$

1. Draw the PDA;
2. Run the PDA with input string 010 and make it get stuck at state *check*.
3. Run the PDA with input string 010 and make it get stuck at state *right*!.
4. Run the PDA with input string 010 and make it get stuck at state *right*! after having read the entire input string.

# Pushdown Automata: Acceptance Criteria

There are **two ways** of accepting strings in a PDA $P = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$:

# Pushdown Automata: Acceptance Criteria

There are **two ways** of accepting strings in a PDA $P = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$:

1. If **we can end up in a final state** after consuming it:

$$L(P) = \{ w \mid (q_0, w, Z_0) \vdash_P^* (q, \epsilon, \gamma) \}$$

for $q \in F$ and any $\gamma \in \Gamma^*$.

# Pushdown Automata: Acceptance Criteria

There are **two ways** of accepting strings in a PDA $P = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$:

1. If **we can end up in a final state** after consuming it:

$$L(P) = \{ w \mid (q_0, w, Z_0) \vdash_P^* (q, \epsilon, \gamma) \}$$

for $q \in F$ and any $\gamma \in \Gamma^*$.

2. If **we can end up with an empty stack** after consuming it:

$$N(P) = \{ w \mid (q_0, w, Z_0) \vdash_P^* (q, \epsilon, \epsilon) \}$$

for any $q \in Q$.

# Pushdown Automata: Acceptance Criteria

There are **two ways** of accepting strings in a PDA $P = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$:

1. If **we can end up in a final state** after consuming it:

$$L(P) = \{ w \mid (q_0, w, Z_0) \vdash_P^* (q, \epsilon, \gamma) \}$$

for $q \in F$ and any $\gamma \in \Gamma^*$.

2. If **we can end up with an empty stack** after consuming it:

$$N(P) = \{ w \mid (q_0, w, Z_0) \vdash_P^* (q, \epsilon, \epsilon) \}$$

for any $q \in Q$.

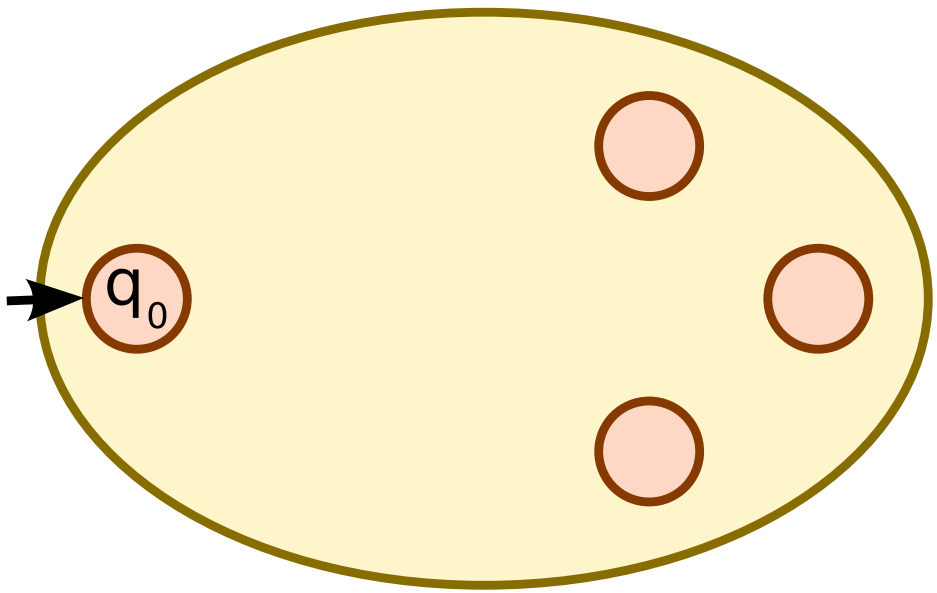
In fact, we can always **switch from one to the other**!

# Pushdown Automata: Acceptance Criteria

How do we go from **acceptance by empty stack** to **acceptance by final state**?

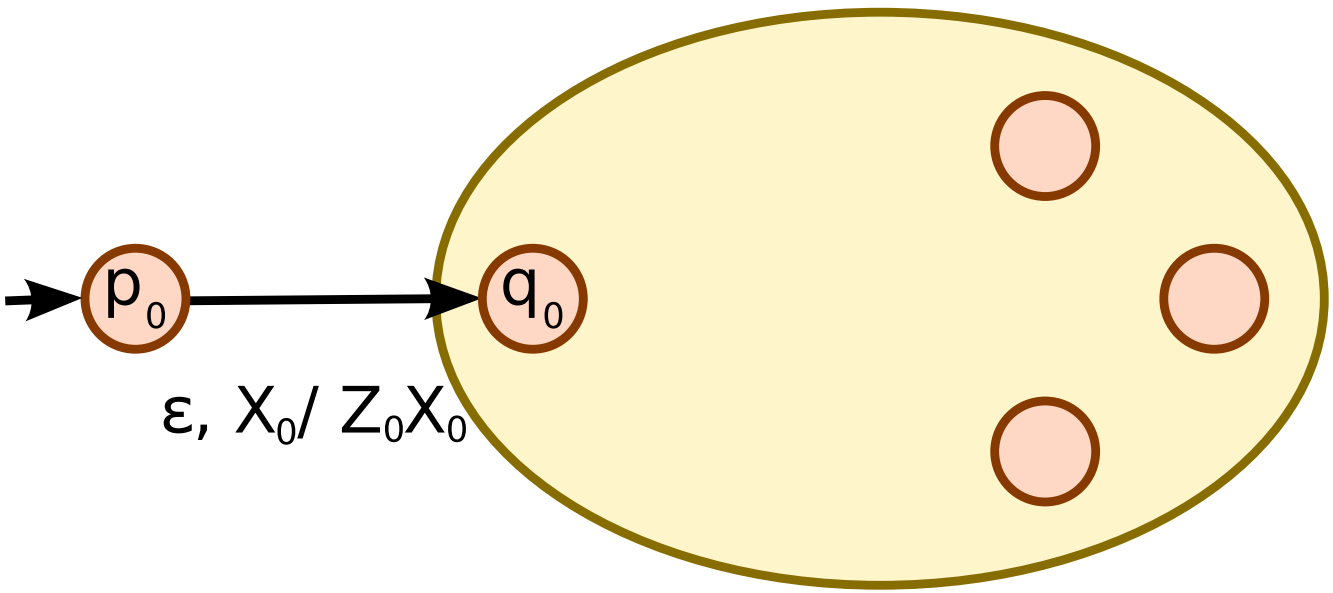
# Pushdown Automata: Acceptance Criteria

How do we go from **acceptance by empty stack** to **acceptance by final state**?



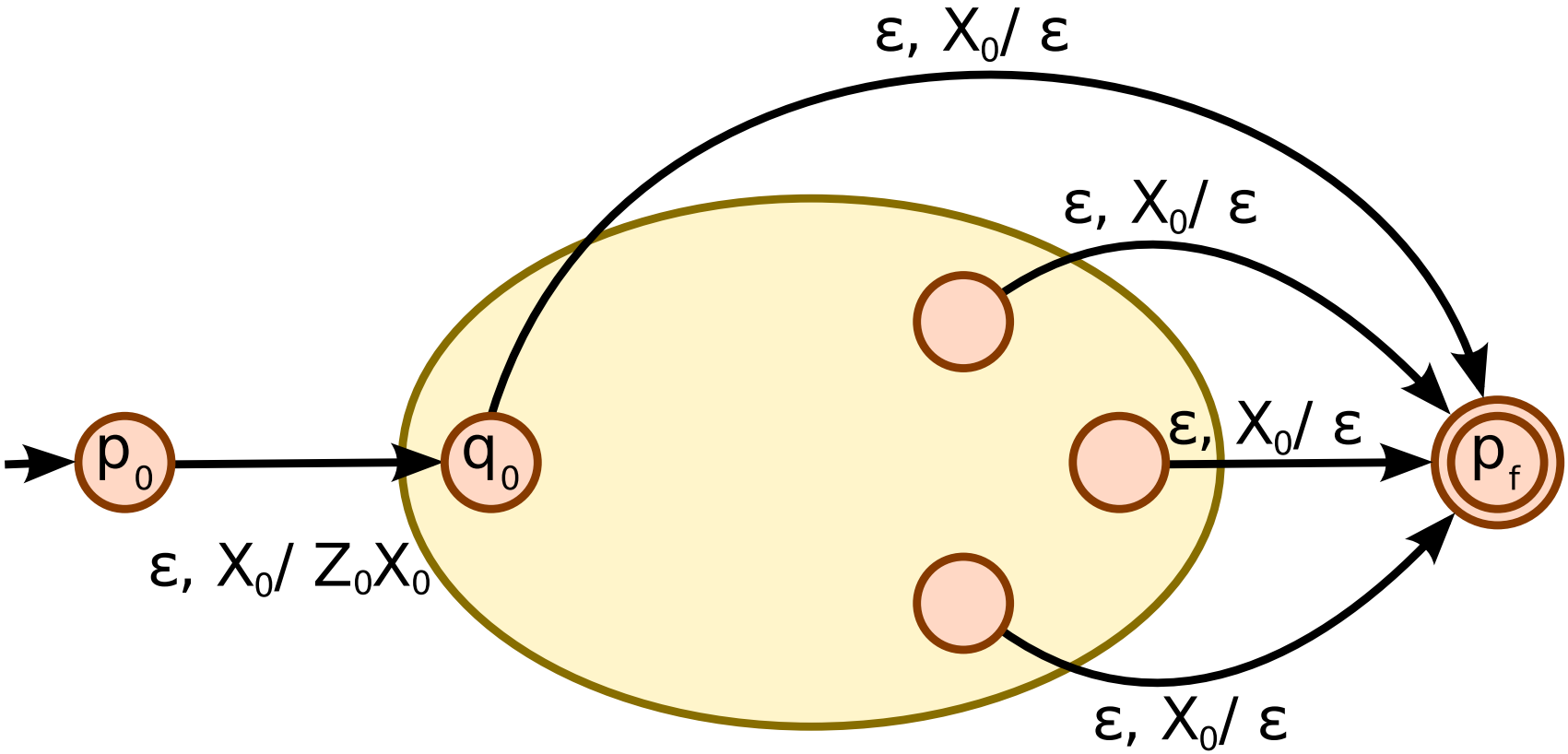
# Pushdown Automata: Acceptance Criteria

How do we go from **acceptance by empty stack** to **acceptance by final state**?



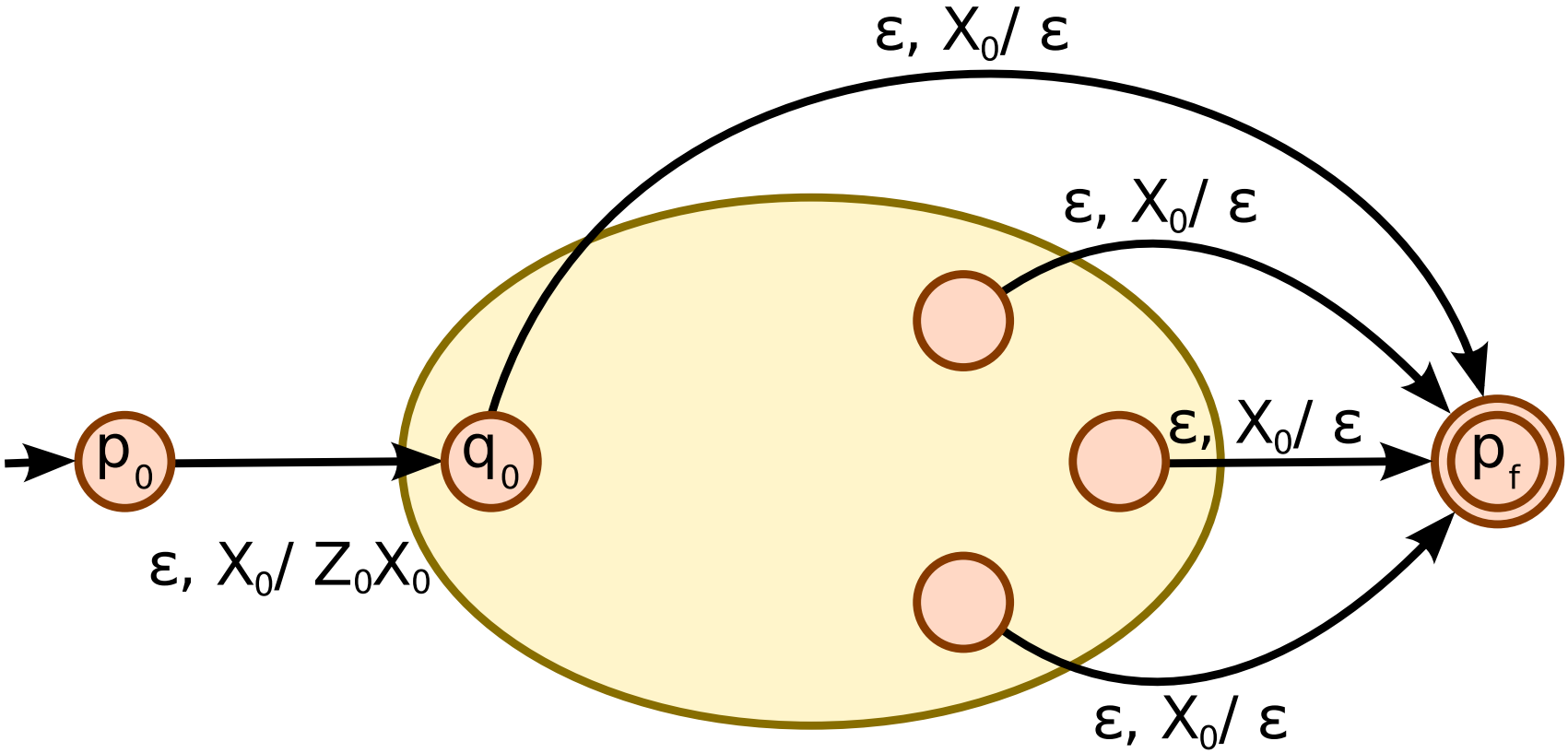
# Pushdown Automata: Acceptance Criteria

How do we go from **acceptance by empty stack** to **acceptance by final state**?



# Pushdown Automata: Acceptance Criteria

How do we go from **acceptance by empty stack** to **acceptance by final state**?



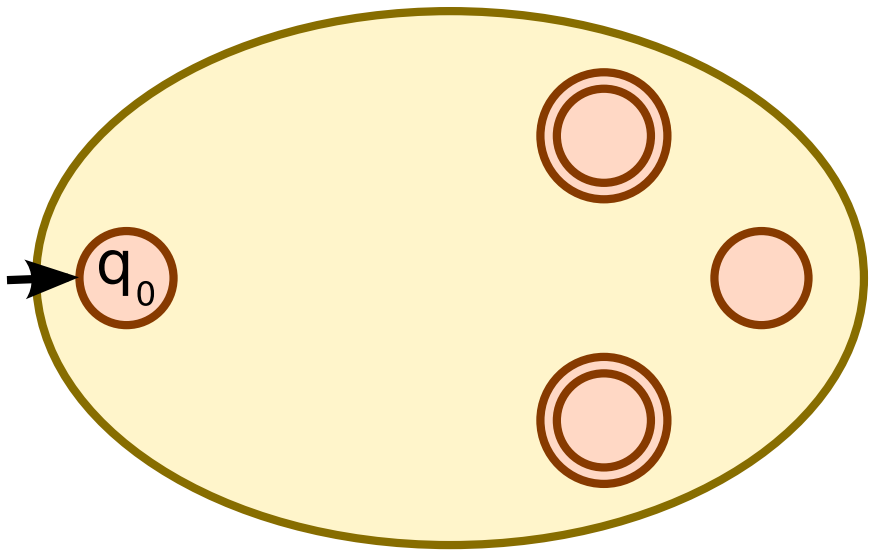

**Idea: move to final state when stack is empty.**

# Pushdown Automata: Acceptance Criteria

How do we go from **acceptance by final state** to **acceptance by empty stack**?

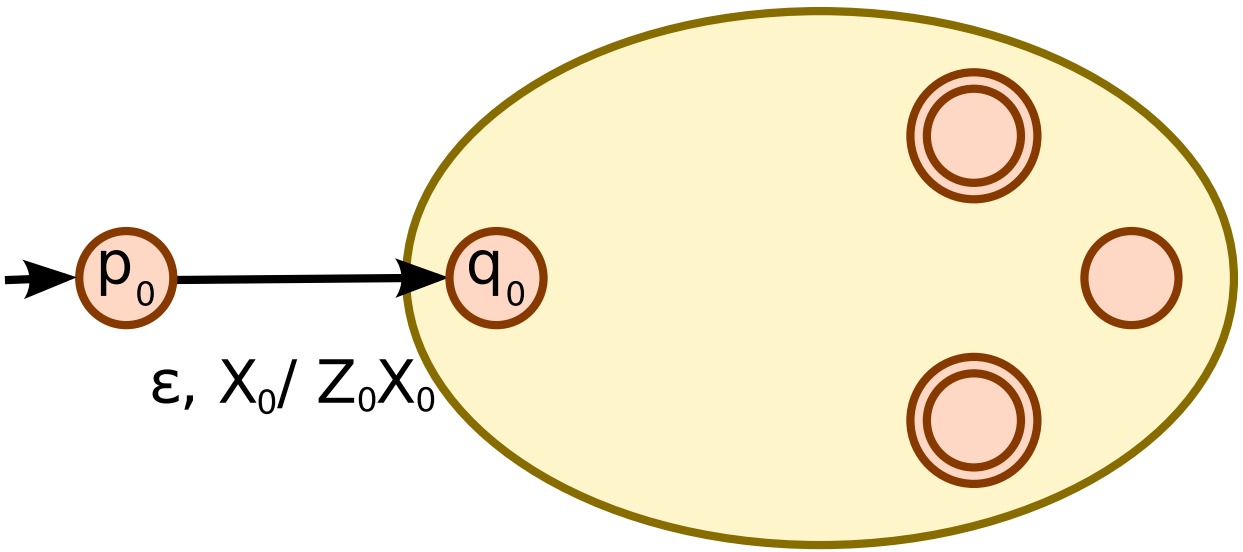
# Pushdown Automata: Acceptance Criteria

How do we go from **acceptance by final state** to **acceptance by empty stack**?



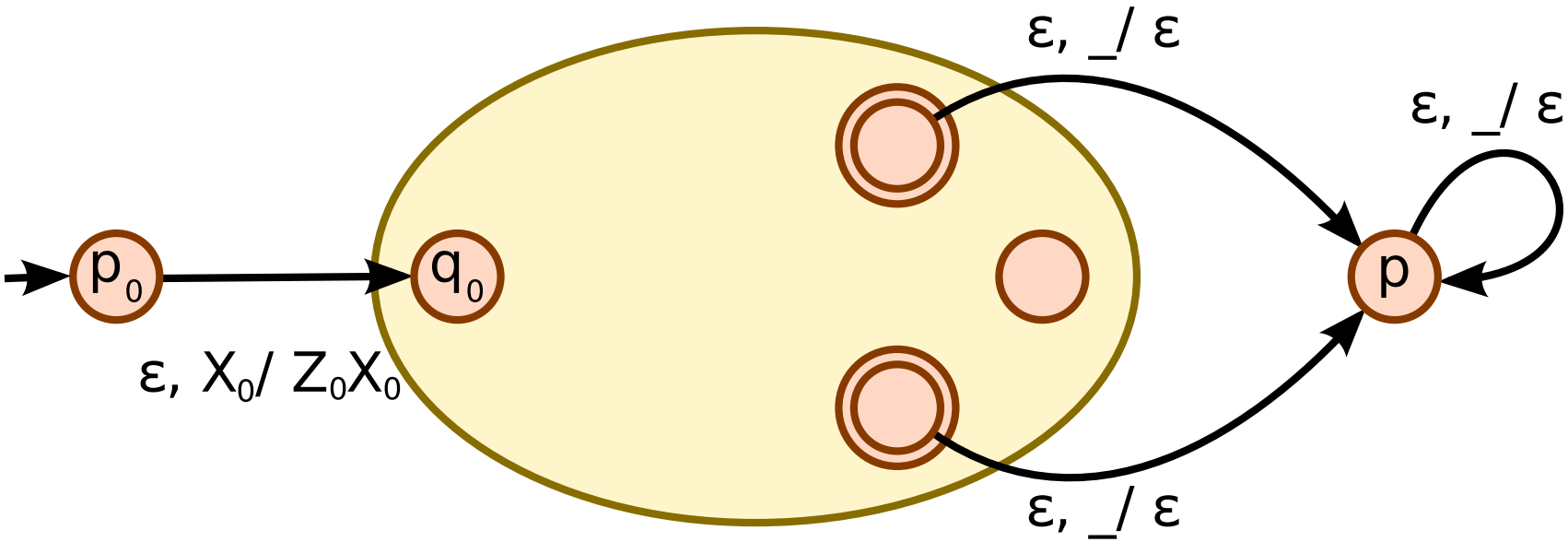
# Pushdown Automata: Acceptance Criteria

How do we go from **acceptance by final state** to **acceptance by empty stack**?



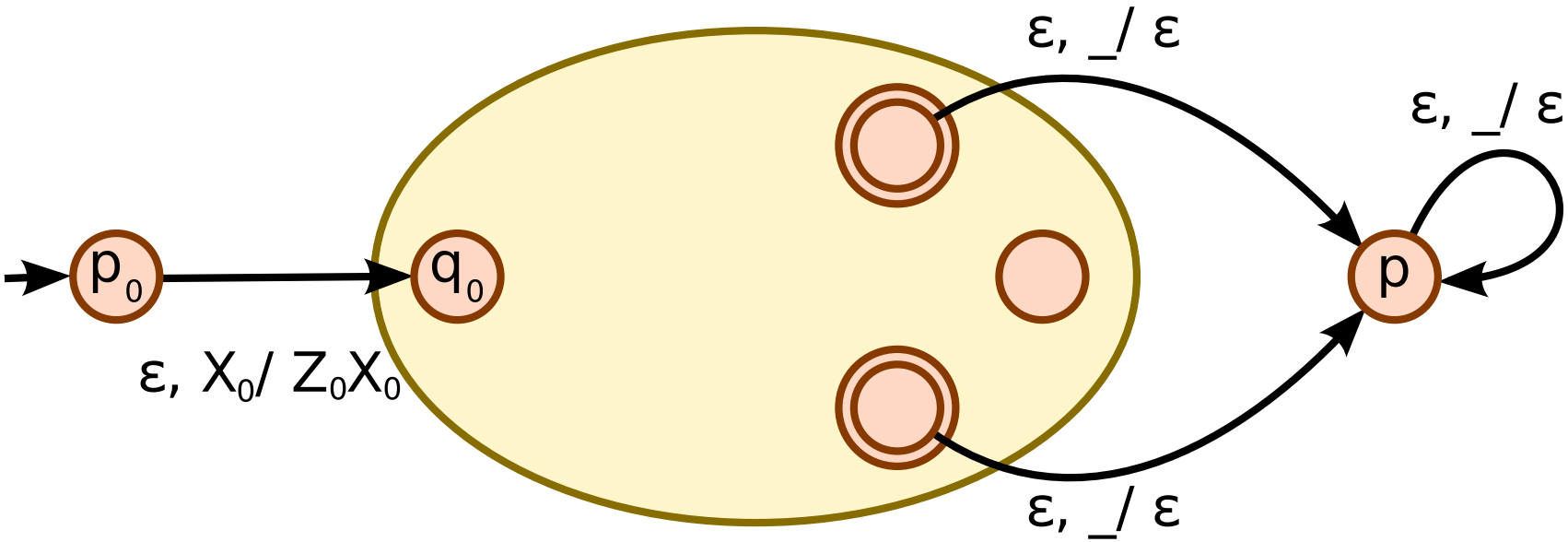
# Pushdown Automata: Acceptance Criteria

How do we go from **acceptance by final state** to **acceptance by empty stack**?



# Pushdown Automata: Acceptance Criteria

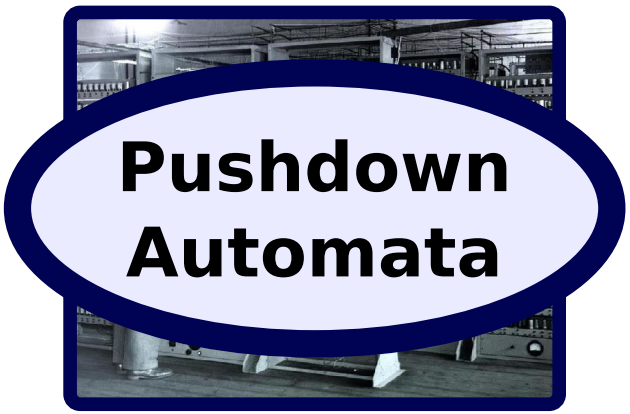
How do we go from **acceptance by final state** to **acceptance by empty stack**?




**Idea: empty the stack when we reach a final state.**

# Relating PDAs to Context-Free Grammars

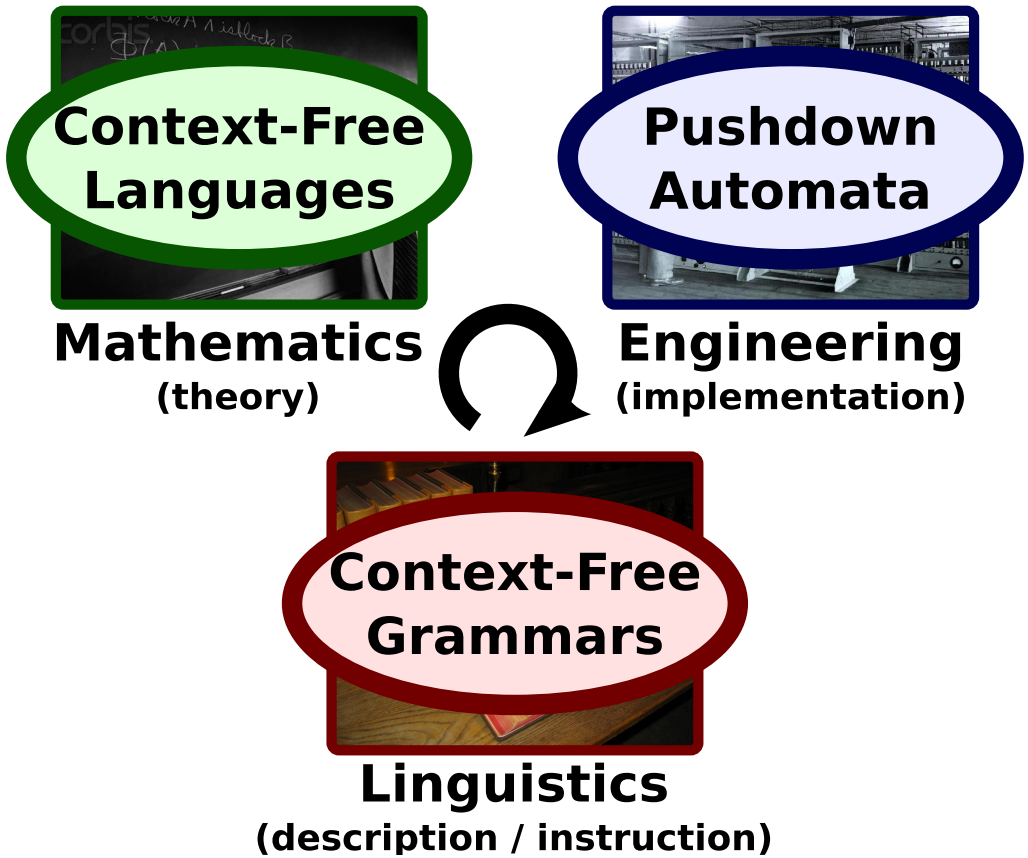

**Mathematics**
**(theory)**




**Engineering**
**(implementation)**

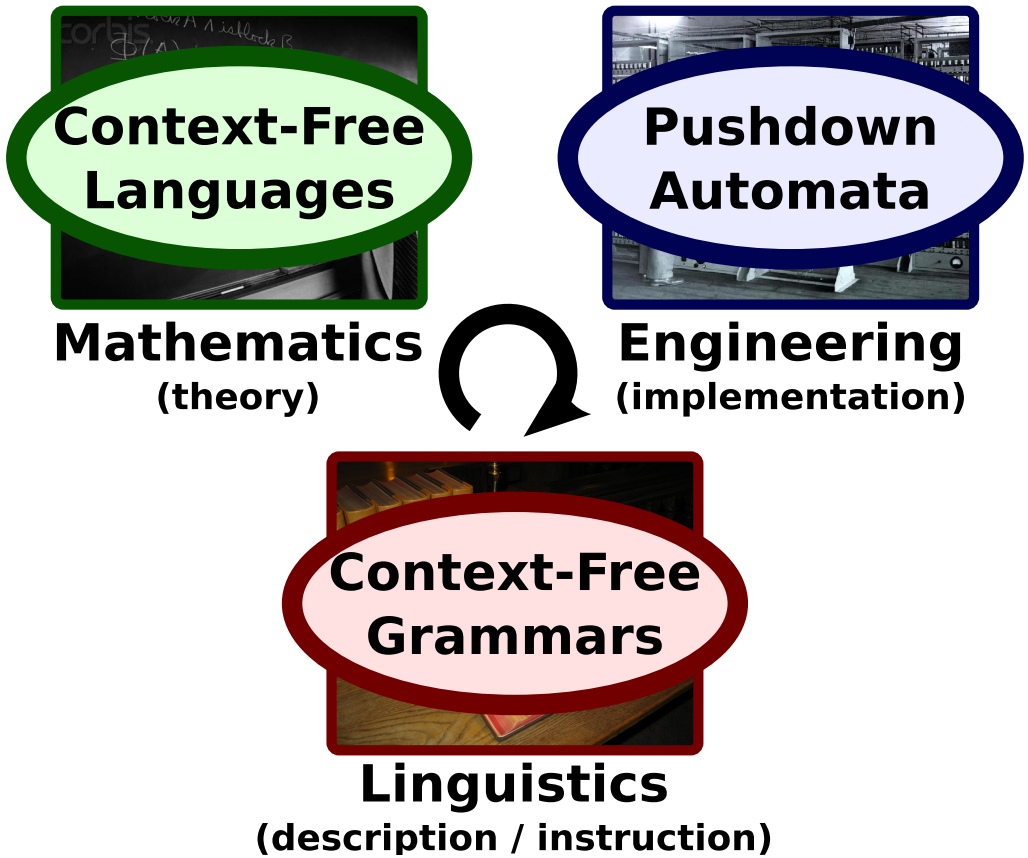

**Linguistics**
**(description / instruction)**

# Relating PDAs to Context-Free Grammars




Pushdown automata describe exactly the class of context-free languages:

# Relating PDAs to Context-Free Grammars




Pushdown automata describe exactly the class of context-free languages:
**Theorem.** A language $L$ is context-free if and only if it is $L = N(P)$ for some pushdown automaton $P$.

# Relating PDAs to Context-Free Grammars

One direction of this proof is hard: given a PDA $P$, find a context-free grammar $G$ with $N(P) = L(G)$.

# Relating PDAs to Context-Free Grammars

One direction of this proof is hard: given a PDA $P$, find a context-free grammar $G$ with $N(P) = L(G)$. You find it in § 6.3.2 in the book.

# Relating PDAs to Context-Free Grammars

We show the other direction: **convert** a grammar $G$ into a PDA $P$ so that $N(P) = L(G)$.

# Relating PDAs to Context-Free Grammars

We show the other direction: **convert** a grammar $G$ into a PDA $P$ so that $N(P) = L(G)$.

Starting with the **grammar** $G = (V, T, R, S)$, we use the following idea:

# Relating PDAs to Context-Free Grammars

We show the other direction: **convert** a grammar $G$ into a PDA $P$ so that $N(P) = L(G)$.

Starting with the **grammar** $G = (V, T, R, S)$, we use the following idea:

- Start with the symbol $S$ on the stack.

# Relating PDAs to Context-Free Grammars

We show the other direction: **convert** a grammar $G$ into a PDA $P$ so that $N(P) = L(G)$.

Starting with the **grammar** $G = (V, T, R, S)$, we use the following idea:

- Start with the symbol $S$ on the stack.
- If we see a **variable** on the top of the stack, replace it with the body of one of its productions.

# Relating PDAs to Context-Free Grammars

We show the other direction: **convert** a grammar $G$ into a PDA $P$ so that $N(P) = L(G)$.

Starting with the **grammar** $G = (V, T, R, S)$, we use the following idea:

- Start with the symbol $S$ on the stack.
- If we see a **variable** on the top of the stack, replace it with the body of one of its productions.
- If we see a **terminal** on the top of the stack, pop it from the stack if we can read it from the input.

# Relating PDAs to Context-Free Grammars

We show the other direction: **convert** a grammar $G$ into a PDA $P$ so that $N(P) = L(G)$.

Starting with the **grammar** $G = (V, T, R, S)$, we use the following idea:

- Start with the symbol $S$ on the stack.
- If we see a **variable** on the top of the stack, replace it with the body of one of its productions.
- If we see a **terminal** on the top of the stack, pop it from the stack if we can read it from the input.

This gives us a **PDA** $P = (\{ q \}, T, V \cup T, \delta, q, S)$ with $\delta$ defined by:

# Relating PDAs to Context-Free Grammars

We show the other direction: **convert** a grammar $G$ into a PDA $P$ so that $N(P) = L(G)$.

Starting with the **grammar** $G = (V, T, R, S)$, we use the following idea:

- Start with the symbol $S$ on the stack.
- If we see a **variable** on the top of the stack, replace it with the body of one of its productions.
- If we see a **terminal** on the top of the stack, pop it from the stack if we can read it from the input.

This gives us a **PDA** $P = (\{ q \}, T, V \cup T, \delta, q, S)$ with $\delta$ defined by:

- For each variable $A \in V$:

$$\delta(q, \epsilon, A) = \{ (q, \gamma) \mid A \to \gamma \in R \}$$

# Relating PDAs to Context-Free Grammars

We show the other direction: **convert** a grammar $G$ into a PDA $P$ so that $N(P) = L(G)$.

Starting with the **grammar** $G = (V, T, R, S)$, we use the following idea:

- Start with the symbol $S$ on the stack.
- If we see a **variable** on the top of the stack, replace it with the body of one of its productions.
- If we see a **terminal** on the top of the stack, pop it from the stack if we can read it from the input.

This gives us a **PDA** $P = (\{ q \}, T, V \cup T, \delta, q, S)$ with $\delta$ defined by:

- For each variable $A \in V$:

$$\delta(q, \epsilon, A) = \{ (q, \gamma) \mid A \to \gamma \in R \}$$

- For each terminal $a \in T$:

$$\delta(q, a, a) = \{ (q, \epsilon) \}$$

# Example

$$S \rightarrow 0S0 \mid 1S1 \mid 2$$

# Example

$$S \to 0S0 \mid 1S1 \mid 2$$

This describes all palindromes over $\{0, 1\}$ with a 2 in the middle.

# Example

$$S \to 0S0 \mid 1S1 \mid 2$$

Create PDA $P = (\{q\}, \{0, 1, 2\}, \{0, 1, 2, S\}, \delta, q, S)$ where

$$\begin{aligned}
\delta(q, \epsilon, S) &= \{(q, 0S0), (q, 1S1), (q, 2)\} \\
\delta(q, 0, 0) &= \{(q, \epsilon)\} \\
\delta(q, 1, 1) &= \{(q, \epsilon)\} \\
\delta(q, 2, 2) &= \{(q, \epsilon)\}
\end{aligned}$$

# Exercise 4.2

Consider the CFG

$$E \to 0 \mid 1 \mid E + E \mid E * E \mid (E)$$

1. Create an equivalent PDA using the construction shown in the slides.

# Example

$$S \rightarrow 0S0 \mid 1S1 \mid 2$$

# Example

$$S \to 0S0 \mid 1S1 \mid 2$$

This describes all palindromes over $\{0, 1\}$ with a 2 in the middle.

# Example

$$S \to 0S0 \mid 1S1 \mid 2$$

Create PDA $P = (\{q\}, \{0, 1, 2\}, \{0, 1, 2, S\}, \delta, q, S)$ where

$$\begin{aligned}
\delta(q, \epsilon, S) &= \{(q, 0S0), (q, 1S1), (q, 2)\} \\
\delta(q, 0, 0) &= \{(q, \epsilon)\} \\
\delta(q, 1, 1) &= \{(q, \epsilon)\} \\
\delta(q, 2, 2) &= \{(q, \epsilon)\}
\end{aligned}$$

# Example

$$S \to 0S0 \mid 1S1 \mid 2$$

Create PDA $P = (\{q\}, \{0, 1, 2\}, \{0, 1, 2, S\}, \delta, q, S)$ where

$$\begin{aligned}
\delta(q, \epsilon, S) &= \{(q, 0S0), (q, 1S1), (q, 2)\} \\
\delta(q, 0, 0) &= \{(q, \epsilon)\} \\
\delta(q, 1, 1) &= \{(q, \epsilon)\} \\
\delta(q, 2, 2) &= \{(q, \epsilon)\}
\end{aligned}$$

This PDA behaves as follows:

- First, non-deterministically guess a production.
- Then check that the next letter can be parsed.

This is maybe **not** how you want to implement a parser!

# Example

$$S \to 0S0 \mid 1S1 \mid 2$$

Create PDA $P = (\{q\}, \{0,1,2\}, \{0,1,2,S\}, \delta, q, S)$ where

$$\begin{aligned}
\delta(q, \epsilon, S) &= \{(q, 0S0), (q, 1S1), (q, 2)\} \\
\delta(q, 0, 0) &= \{(q, \epsilon)\} \\
\delta(q, 1, 1) &= \{(q, \epsilon)\} \\
\delta(q, 2, 2) &= \{(q, \epsilon)\}
\end{aligned}$$

This PDA behaves as follows:

- First, non-deterministically guess a production.
- Then check that the next letter can be parsed.

This is maybe **not** how you want to implement a parser!

**Instead, you should look ahead!**

# Example

$$S \to 0S0 \mid 1S1 \mid 2$$

Equivalent deterministic PDA: $P = (\{q\}, \{0, 1, 2\}, \{0, 1, 2, S\}, \delta, q, S)$ where

$$\begin{aligned}
\delta(q, 0, S) &= (q, S0) \\
\delta(q, 1, S) &= (q, S1) \\
\delta(q, 2, S) &= (q, \epsilon) \\
\delta(q, 0, 0) &= (q, \epsilon) \\
\delta(q, 1, 1) &= (q, \epsilon)
\end{aligned}$$

**Looking ahead!**

# Non-Determinism in PDAs

What happens if we leave out the "middle marker" 2?

$$S \Rightarrow 0S0 \mid 1S1 \mid \epsilon$$

i.e. the set of all palindromes over $\{0, 1\}$ of even length.

# Non-Determinism in PDAs

What happens if we leave out the "middle marker" 2?

$$S \Rightarrow 0S0 \mid 1S1 \mid \epsilon$$

i.e. the set of all palindromes over $\{0, 1\}$ of even length.
The general grammar-to-PDA construction yields:

$$\begin{aligned} \delta(q, \epsilon, S) &= \{(q, 0S0), (q, 1S1), (q, \epsilon)\} \\ \delta(q, 0, 0) &= \{(q, \epsilon)\} \\ \delta(q, 1, 1) &= \{(q, \epsilon)\} \end{aligned}$$

# Non-Determinism in PDAs

What happens if we leave out the "middle marker" 2?

$$S \Rightarrow 0S0 \mid 1S1 \mid \epsilon$$

i.e. the set of all palindromes over $\{0, 1\}$ of even length.
The general grammar-to-PDA construction yields:

$$\begin{array}{lcl} \delta(q, \epsilon, S) & = & \{(q, 0S0), (q, 1S1), (q, \epsilon)\} \\ \delta(q, 0, 0) & = & \{(q, \epsilon)\} \\ \delta(q, 1, 1) & = & \{(q, \epsilon)\} \end{array}$$

Can we construct a deterministic PDA?

# Non-Determinism in PDAs

What happens if we leave out the "middle marker" 2?

$$S \Rightarrow 0S0 \mid 1S1 \mid \epsilon$$

i.e. the set of all palindromes over $\{0, 1\}$ of even length.
Can we construct a deterministic PDA?
We can improve it a bit...

$$\begin{aligned}
\delta(q, \epsilon, S) &= (q, \epsilon) \\
\delta(q, 0, S) &= (q, S0) \\
\delta(q, 1, S) &= (q, S1) \\
\delta(q, 0, 0) &= (q, \epsilon) \\
\delta(q, 1, 1) &= (q, \epsilon)
\end{aligned}$$

# Non-Determinism in PDAs

What happens if we leave out the "middle marker" 2?

$$S \Rightarrow 0S0 \mid 1S1 \mid \epsilon$$

i.e. the set of all palindromes over $\{0, 1\}$ of even length.
Can we construct a deterministic PDA?
We can improve it a bit...

$$\begin{aligned}
\delta(q, \epsilon, S) &= (q, \epsilon) \\
\delta(q, 0, S) &= (q, S0) \\
\delta(q, 1, S) &= (q, S1) \\
\delta(q, 0, 0) &= (q, \epsilon) \\
\delta(q, 1, 1) &= (q, \epsilon)
\end{aligned}$$

This automaton non-deterministically guesses whether it has reached the middle of the palindrome.

# Non-Determinism in PDAs

What happens if we leave out the "middle marker" 2?

$$S \Rightarrow 0S0 \mid 1S1 \mid \epsilon$$

i.e. the set of all palindromes over $\{0, 1\}$ of even length.
Can we construct a deterministic PDA?
We can improve it a bit...

$$\begin{aligned}
\delta(q, \epsilon, S) &= (q, \epsilon) \\
\delta(q, 0, S) &= (q, S0) \\
\delta(q, 1, S) &= (q, S1) \\
\delta(q, 0, 0) &= (q, \epsilon) \\
\delta(q, 1, 1) &= (q, \epsilon)
\end{aligned}$$

This automaton non-deterministically guesses whether it has reached the middle of the palindrome.
**There is no deterministic PDA for this language.**

# Deterministic PDAs

- PDAs correspond to the context-free languages.

# Deterministic PDAs

- PDAs correspond to the context-free languages.
- Deterministic PDAs (DPDAs) are a strictly weaker concept: not all context-free languages can be recognized by DPDAs.

# Deterministic PDAs

- PDAs correspond to the context-free languages.
- Deterministic PDAs (DPDAs) are a strictly weaker concept:
  not all context-free languages can be recognized by DPDAs.
- Luckily, most languages of practical use can be recognized by DPDAs.

# Deterministic PDAs

- PDAs correspond to the context-free languages.
- Deterministic PDAs (DPDAs) are a strictly weaker concept:
  not all context-free languages can be recognized by DPDAs.
- Luckily, most languages of practical use can be recognized by DPDAs.
- DPDAs are the basis for most parser generators.

# Deterministic PDAs

A deterministic PDA (DPDA) is a PDA such that for

- every state $q \in Q$
- every string $a \in \Sigma$
- every stack symbol $A \in \Gamma$

we have that

$$|\bigcup_{\alpha \in \{a, \epsilon\}} \delta(q, \alpha, A)| \leq 1$$

This implies that every configuration is either stuck or has exactly one possible follow-up configuration... it never has a choice!

If $P$ is not deterministic, then we say that it is non-deterministic.

# Relating PDAs to Context-Free Grammars

**Example**: a very simple grammar:

$$\begin{aligned} E &\rightarrow A + E \mid A * E \mid A \bullet \\ A &\rightarrow 0 \end{aligned}$$

# Relating PDAs to Context-Free Grammars

**Example**: a very simple grammar:

$$\begin{array}{lcl} E & \rightarrow & A + E \mid A * E \mid A \bullet \\ A & \rightarrow & 0 \end{array}$$

We create a PDA $P = (\{ q \}, \{ 0, +, *, \bullet \}, \{ A, E, 0, +, *, \bullet \}, \delta, q, E)$, where:

# Relating PDAs to Context-Free Grammars

**Example**: a very simple grammar:

$$\begin{array}{lcl} E & \rightarrow & A + E \mid A * E \mid A \bullet \\ A & \rightarrow & 0 \end{array}$$

We create a PDA $P = (\{ q \}, \{ 0, +, *, \bullet \}, \{ A, E, 0, +, *, \bullet \}, \delta, q, E)$, where:

$$\begin{array}{lcl} \delta(q, \epsilon, E) & = & \{ (q, A + E), (q, A * E), (q, A \bullet) \} \\ \delta(q, \epsilon, A) & = & \{ (q, 0) \} \\ \delta(q, 0, 0) & = & \{ (q, \epsilon) \} \\ \delta(q, +, +) & = & \{ (q, \epsilon) \} \\ \delta(q, *, *) & = & \{ (q, \epsilon) \} \\ \delta(q, \bullet, \bullet) & = & \{ (q, \epsilon) \} \end{array}$$

# Relating PDAs to Context-Free Grammars

**Example**: a very simple grammar:

$$\begin{array}{lcl} E & \rightarrow & A + E \mid A * E \mid A \bullet \\ A & \rightarrow & 0 \end{array}$$

We create a PDA $P = (\{ q \}, \{ 0, +, *, \bullet \}, \{ A, E, 0, +, *, \bullet \}, \delta, q, E)$, where:

$$\begin{array}{lcl} \delta(q, \epsilon, E) & = & \{ (q, A + E), (q, A * E), (q, A \bullet) \} \\ \delta(q, \epsilon, A) & = & \{ (q, 0) \} \\ \delta(q, 0, 0) & = & \{ (q, \epsilon) \} \\ \delta(q, +, +) & = & \{ (q, \epsilon) \} \\ \delta(q, *, *) & = & \{ (q, \epsilon) \} \\ \delta(q, \bullet, \bullet) & = & \{ (q, \epsilon) \} \end{array}$$

**Note**: this PDA is **non-deterministic**!

# Table-Driven Parsing

If we implement this PDA, how do we know **which rule to pick** for $E$?

# Table-Driven Parsing

If we implement this PDA, how do we know **which rule to pick** for $E$?

We could just pick a rule randomly, and **backtrack** if we make a mistake...

# Table-Driven Parsing

If we implement this PDA, how do we know **which rule to pick** for $E$?

We could just pick a rule randomly, and **backtrack** if we make a mistake...
but this is **horribly inefficient**.

# Table-Driven Parsing

If we implement this PDA, how do we know **which rule to pick** for $E$?

We could just pick a rule randomly, and **backtrack** if we make a mistake... but this is **horribly inefficient**.

Instead, we can pick the rule by sneaking a look at the input!

# Table-Driven Parsing

If we implement this PDA, how do we know **which rule to pick** for $E$?

We could just pick a rule randomly, and **backtrack** if we make a mistake... but this is **horribly inefficient**.

Instead, we can pick the rule by sneaking a look at the input!

We can make our choice based on the **next token** `t` that we have not yet read.

# Table-Driven Parsing

If we implement this PDA, how do we know **which rule to pick** for $E$?

We could just pick a rule randomly, and **backtrack** if we make a mistake... but this is **horribly inefficient**.

Instead, we can pick the rule by sneaking a look at the input!

We can make our choice based on the **next token** `t` that we have not yet read. We call this a **lookahead** of one token.

# Table-Driven Parsing

Let us build a **parser** based on the **pushdown automaton**:

```
boolean read (Symbol startSymbol) {

    Stack<Symbol> stack = new Stack<Symbol>();
    stack.push(startSymbol);

    Token t = lex();

    while(!stack.empty()) {
        Symbol s = stack.pop();
        if (s instanceof Token) {
            if (!s.equals(t)) return false;
            t = lex();
        } else {
            Symbol[] body = chooseRule(s,t);
            for (int i = body.length - 1; i >= 0; i--) {
                stack.push(body[i]);
            }
        }
    }

    return t == null;

}
```

# Table-Driven Parsing

Let us build a **parser** based on the **pushdown automaton**:

```
boolean read (Symbol startSymbol) {

    Stack<Symbol> stack = new Stack<Symbol>();
    stack.push(startSymbol);

    Token t = lex();

    while(!stack.empty()) {
        Symbol s = stack.pop();
        if (s instanceof Token) {
            if (!s.equals(t)) return false;
            t = lex();
        } else {
            Symbol[] body = chooseRule(s,t);
            for (int i = body.length - 1; i >= 0; i--) {
                stack.push(body[i]);
            }
        }
    }

    return t == null;

}
```

# Table-Driven Parsing

This type of parser is called an **LL(1) parser**.

- It reads the input from **L**eft to right.
- It produces a **L**eftmost derivation.
- It uses **1** token of lookahead.

# Table-Driven Parsing

This type of parser is called an **LL(1) parser**.

- It reads the input from **L**eft to right.
- It produces a **L**eftmost derivation.
- It uses **1** token of lookahead.

Let us try to **implement** the function `chooseRule(s,t)` for this grammar

$$S \quad \rightarrow \quad 0S0 \mid 1S1 \mid 2$$

We can draw this as a table, hence the term **table-driven parsing**:

| Variable on stack | Lookahead '0' | Lookahead '1' | Lookahead '2' | |
|---|---|---|---|---|
| Variable $S$ | | | | |

# Table-Driven Parsing

Let us try now with this other grammar:

$$\begin{array}{lcl} E & \rightarrow & A + E \mid A * E \mid A \bullet \\ A & \rightarrow & 0 \end{array}$$

# Table-Driven Parsing

Let us try now with this other grammar:

$$\begin{array}{lcl} E & \rightarrow & A + E \mid A * E \mid A \bullet \\ A & \rightarrow & 0 \end{array}$$

| Variable on stack | '0' | '+' | '*' | '•' |
|---|---|---|---|---|
| $E$ | | | | |
| $A$ | | | | |

# Table-Driven Parsing

Even with lookahead, we failed to determine which rule to pick.

# Table-Driven Parsing

Even with lookahead, we failed to determine which rule to pick.

The problem is that our grammar is **not an LL(1) grammar**.

$$\begin{array}{lcl} E & \rightarrow & A + E \mid A * E \mid A \bullet \\ A & \rightarrow & 0 \end{array}$$

# Table-Driven Parsing

Even with lookahead, we failed to determine which rule to pick.

The problem is that our grammar is **not an LL(1) grammar**.

$$\begin{array}{lcl} E & \rightarrow & A + E \mid A * E \mid A \bullet \\ A & \rightarrow & 0 \end{array}$$

Luckily, we can **fix things**, since $A$ occurs at the start of each body for $E$: **left-factoring**!

# Table-Driven Parsing

Even with lookahead, we failed to determine which rule to pick.

The problem is that our grammar is **not an LL(1) grammar**.

$$\begin{array}{lcl} E & \rightarrow & A\ (+\ E \mid *\ E \mid \bullet) \\ A & \rightarrow & 0 \end{array}$$

Luckily, we can **fix things**, since $A$ occurs at the start of each body for $E$: **left-factoring**!

# Table-Driven Parsing

Even with lookahead, we failed to determine which rule to pick.

The problem is that our grammar is **not an LL(1) grammar**.

$$
\begin{array}{lcl}
E & \rightarrow & A\ T \\
T & \rightarrow & +\ E \mid *\ E \mid \bullet \\
A & \rightarrow & 0
\end{array}
$$

Luckily, we can **fix things**, since $A$ occurs at the start of each body for $E$: **left-factoring**!

# Table-Driven Parsing

$$
\begin{array}{lcl}
E & \rightarrow & A\ T \\
T & \rightarrow & +\ E \mid *\ E \mid \bullet \\
A & \rightarrow & 0
\end{array}
$$

We can now implement `chooseRule(s,t)` for our new **LL(1) grammar**!

| | Input '0' | Input '+' | Input '*' | Input '•' |
|---|---|---|---|---|
| Variable $E$ | | | | |
| Variable $T$ | | | | |
| Variable $A$ | | | | |

# Table-Driven Parsing

$$
\begin{array}{lcl}
E & \rightarrow & A\ T \\
T & \rightarrow & +\ E \mid *\ E \mid \bullet \\
A & \rightarrow & 0
\end{array}
$$

We can now implement `chooseRule(s,t)` for our new **LL(1) grammar**!

| | Input '0' | Input '+' | Input '*' | Input '•' |
|---|---|---|---|---|
| Variable $E$ | $E \rightarrow A\ T$ | | | |
| Variable $T$ | | | | |
| Variable $A$ | | | | |

# Table-Driven Parsing

$$
\begin{array}{lcl}
E & \rightarrow & A\ T \\
T & \rightarrow & +\ E \mid *\ E \mid \bullet \\
A & \rightarrow & 0
\end{array}
$$

We can now implement `chooseRule(s,t)` for our new **LL(1) grammar**!

| | Input '0' | Input '+' | Input '*' | Input '•' |
|---|---|---|---|---|
| Variable $E$ | $E \rightarrow A\ T$ | $\times$ | | |
| Variable $T$ | | | | |
| Variable $A$ | | | | |

# Table-Driven Parsing

$$
\begin{array}{lcl}
E & \rightarrow & A\ T \\
T & \rightarrow & +\ E \mid *\ E \mid \bullet \\
A & \rightarrow & 0
\end{array}
$$

We can now implement `chooseRule(s,t)` for our new **LL(1) grammar**!

| | Input '0' | Input '+' | Input '*' | Input '•' |
|---|---|---|---|---|
| Variable $E$ | $E \rightarrow A\ T$ | $\times$ | $\times$ | |
| Variable $T$ | | | | |
| Variable $A$ | | | | |

# Table-Driven Parsing

$$
\begin{array}{lcl}
E & \rightarrow & A\ T \\
T & \rightarrow & +\ E \mid *\ E \mid \bullet \\
A & \rightarrow & 0
\end{array}
$$

We can now implement `chooseRule(s,t)` for our new **LL(1) grammar**!

| | Input '0' | Input '+' | Input '*' | Input '•' |
|---|---|---|---|---|
| Variable $E$ | $E \rightarrow A\ T$ | $\times$ | $\times$ | $\times$ |
| Variable $T$ | | | | |
| Variable $A$ | | | | |

# Table-Driven Parsing

$$
\begin{array}{lcl}
E & \rightarrow & A\ T \\
T & \rightarrow & +\ E \mid *\ E \mid \bullet \\
A & \rightarrow & 0
\end{array}
$$

We can now implement `chooseRule(s,t)` for our new **LL(1) grammar**!

| | Input '0' | Input '+' | Input '*' | Input '•' |
|---|---|---|---|---|
| Variable $E$ | $E \rightarrow A\ T$ | $\times$ | $\times$ | $\times$ |
| Variable $T$ | $\times$ | | | |
| Variable $A$ | | | | |

# Table-Driven Parsing

$$
\begin{array}{lcl}
E & \rightarrow & A\ T \\
T & \rightarrow & +\ E \mid *\ E \mid \bullet \\
A & \rightarrow & 0
\end{array}
$$

We can now implement `chooseRule(s,t)` for our new **LL(1) grammar**!

| | Input '0' | Input '$+$' | Input '$*$' | Input '$\bullet$' |
|---|---|---|---|---|
| Variable $E$ | $E \rightarrow A\ T$ | $\times$ | $\times$ | $\times$ |
| Variable $T$ | $\times$ | $T \rightarrow +\ E$ | | |
| Variable $A$ | | | | |

# Table-Driven Parsing

$$
\begin{array}{lcl}
E & \rightarrow & A\ T \\
T & \rightarrow & +\ E \mid *\ E \mid \bullet \\
A & \rightarrow & 0
\end{array}
$$

We can now implement `chooseRule(s,t)` for our new **LL(1) grammar**!

| | Input '0' | Input '$+$' | Input '$*$' | Input '$\bullet$' |
|---|---|---|---|---|
| Variable $E$ | $E \rightarrow A\ T$ | $\times$ | $\times$ | $\times$ |
| Variable $T$ | $\times$ | $T \rightarrow +\ E$ | $T \rightarrow *\ E$ | |
| Variable $A$ | | | | |

# Table-Driven Parsing

$$
\begin{array}{lcl}
E & \rightarrow & A\ T \\
T & \rightarrow & +\ E \mid *\ E \mid \bullet \\
A & \rightarrow & 0
\end{array}
$$

We can now implement `chooseRule(s,t)` for our new **LL(1) grammar**!

| | Input '0' | Input '+' | Input '*' | Input '•' |
|---|---|---|---|---|
| Variable $E$ | $E \rightarrow A\ T$ | $\times$ | $\times$ | $\times$ |
| Variable $T$ | $\times$ | $T \rightarrow +\ E$ | $T \rightarrow *\ E$ | $T \rightarrow \bullet$ |
| Variable $A$ | | | | |

# Table-Driven Parsing

$$
\begin{array}{lcl}
E & \rightarrow & A\ T \\
T & \rightarrow & +\ E \mid *\ E \mid \bullet \\
A & \rightarrow & 0
\end{array}
$$

We can now implement `chooseRule(s,t)` for our new **LL(1) grammar**!

| | Input '0' | Input '$+$' | Input '$*$' | Input '$\bullet$' |
|---|---|---|---|---|
| Variable $E$ | $E \rightarrow A\ T$ | $\times$ | $\times$ | $\times$ |
| Variable $T$ | $\times$ | $T \rightarrow +\ E$ | $T \rightarrow *\ E$ | $T \rightarrow \bullet$ |
| Variable $A$ | $A \rightarrow 0$ | | | |

# Table-Driven Parsing

$$
\begin{array}{lcl}
E & \rightarrow & A\ T \\
T & \rightarrow & +\ E \mid *\ E \mid \bullet \\
A & \rightarrow & 0
\end{array}
$$

We can now implement `chooseRule(s,t)` for our new **LL(1) grammar**!

| | Input '0' | Input '$+$' | Input '$*$' | Input '$\bullet$' |
|---|---|---|---|---|
| Variable $E$ | $E \rightarrow A\ T$ | $\times$ | $\times$ | $\times$ |
| Variable $T$ | $\times$ | $T \rightarrow +\ E$ | $T \rightarrow *\ E$ | $T \rightarrow \bullet$ |
| Variable $A$ | $A \rightarrow 0$ | $\times$ | | |

# Table-Driven Parsing

$$
\begin{array}{lcl}
E & \rightarrow & A\ T \\
T & \rightarrow & +\ E \mid *\ E \mid \bullet \\
A & \rightarrow & 0
\end{array}
$$

We can now implement `chooseRule(s,t)` for our new **LL(1) grammar**!

| | Input '0' | Input '$+$' | Input '$*$' | Input '$\bullet$' |
|---|---|---|---|---|
| Variable $E$ | $E \rightarrow A\ T$ | $\times$ | $\times$ | $\times$ |
| Variable $T$ | $\times$ | $T \rightarrow +\ E$ | $T \rightarrow *\ E$ | $T \rightarrow \bullet$ |
| Variable $A$ | $A \rightarrow 0$ | $\times$ | $\times$ | |

# Table-Driven Parsing

$$
\begin{array}{lcl}
E & \rightarrow & A\ T \\
T & \rightarrow & +\ E \mid *\ E \mid \bullet \\
A & \rightarrow & 0
\end{array}
$$

We can now implement `chooseRule(s,t)` for our new **LL(1) grammar**!

| | Input '0' | Input '$+$' | Input '$*$' | Input '$\bullet$' |
|---|---|---|---|---|
| Variable $E$ | $E \rightarrow A\ T$ | $\times$ | $\times$ | $\times$ |
| Variable $T$ | $\times$ | $T \rightarrow +\ E$ | $T \rightarrow *\ E$ | $T \rightarrow \bullet$ |
| Variable $A$ | $A \rightarrow 0$ | $\times$ | $\times$ | $\times$ |

# Table-Driven Parsing

$$
\begin{array}{lcl}
E & \rightarrow & A\ T \\
T & \rightarrow & +\ E \mid *\ E \mid \bullet \\
A & \rightarrow & 0
\end{array}
$$

We can now implement `chooseRule(s,t)` for our new **LL(1) grammar**!

| | Input '0' | Input '$+$' | Input '$*$' | Input '$\bullet$' |
|---|---|---|---|---|
| Variable $E$ | $E \rightarrow A\ T$ | $\times$ | $\times$ | $\times$ |
| Variable $T$ | $\times$ | $T \rightarrow +\ E$ | $T \rightarrow *\ E$ | $T \rightarrow \bullet$ |
| Variable $A$ | $A \rightarrow 0$ | $\times$ | $\times$ | $\times$ |

This gives us an efficient parsing algorithm, which corresponds to a **deterministic pushdown automaton**.

# How to build a parsing table (no $\epsilon$ productions)

- For every production $A \rightarrow \gamma$ we compute the set of lookahead symbols (which will indicate that we have to choose $A \rightarrow \gamma$) as
  - $First(\gamma)$

- $First(\gamma)$ are all the symbols that may appear as the starting symbol of some string generated by $\gamma$. $First(\gamma)$ can be recursively defined as

$$\begin{array}{rcl} First(a\gamma) & = & a \\ First(B\gamma) & = & \cup_{B \rightarrow \gamma_i} First(\gamma_i) \quad \text{if } B \text{ cannot produce } \epsilon \end{array}$$

# How to build a parsing table (general case)

- For every production $A \to \gamma$ we compute the set of lookahead symbols (which will indicate that we have to choose $A \to \gamma$) as the union of
  - $First(\gamma)$ and
  - $Follow(A)$ if $\gamma$ can produce $\epsilon$
- $First(\gamma)$ are all the symbols that may appear as the starting symbol of some string generated by $\gamma$. $First(\gamma)$ can be recursively defined as

$$
\begin{array}{rcll}
First(a\gamma) & = & a & \\
First(B\gamma) & = & \cup_{B \to \gamma_i} First(\gamma_i) & \text{if } B \text{ cannot produce } \epsilon \\
First(B\gamma) & = & \cup_{B \to \gamma_i} First(\gamma_i) \cup First(\gamma) & \text{if } B \text{ can produce } \epsilon
\end{array}
$$

- $Follow(A)$ are all the symbols that can follow a string generated by $A$. We need to consider all occurrences of $A$ in the rhs of productions:
  - if $A$ is followed by a terminal $a$ then include $a$ in $Follow(A)$;
  - if $A$ is followed by a non-terminal $B$ then include $First(B)$ in $Follow(A)$;
  - if $A$ appears at the end of a production for some non-terminal $B$ then include $Follow(B)$ in $Follow(A)$.

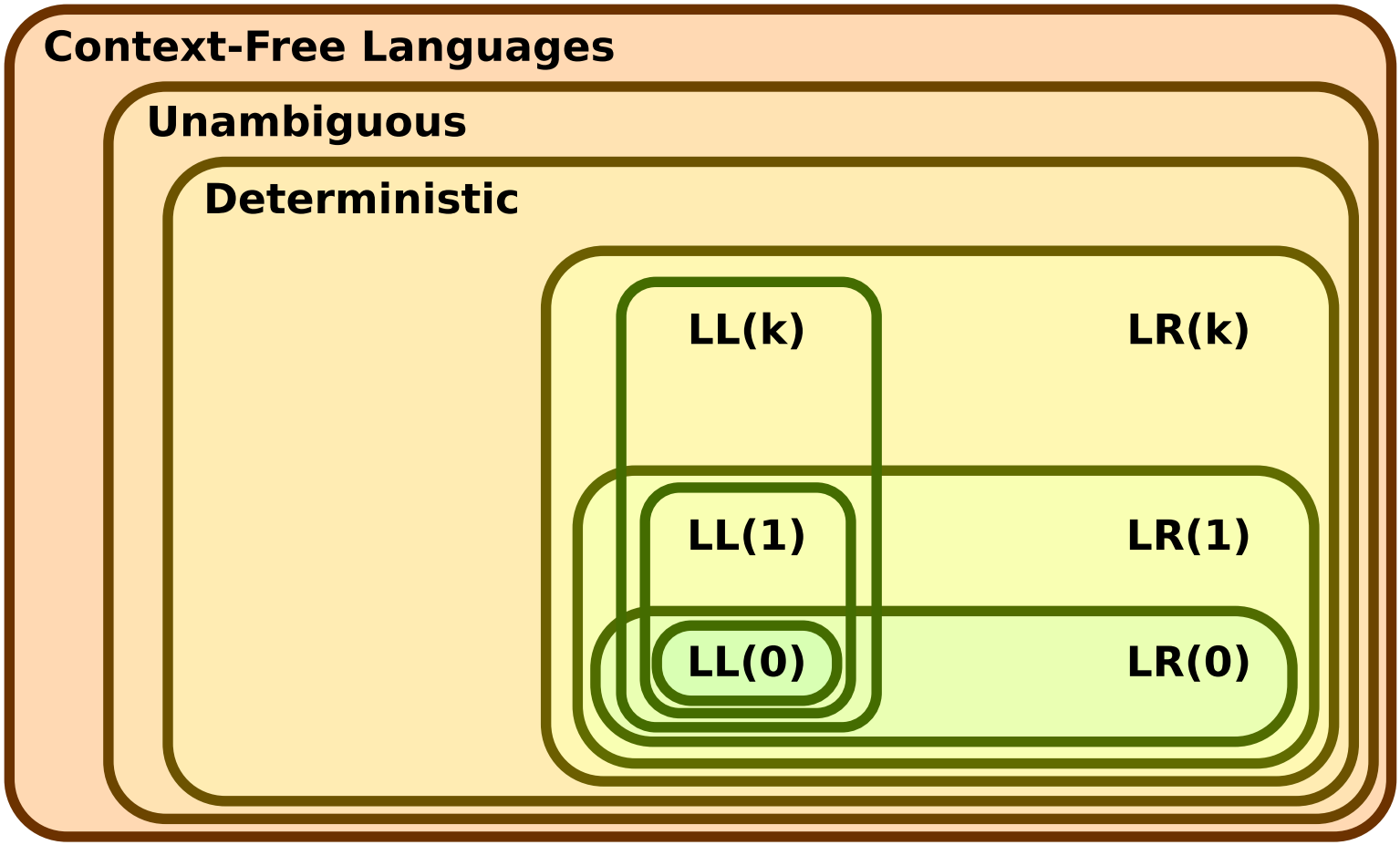
# Exercise 4.3

Build the parsing table for the following grammar:

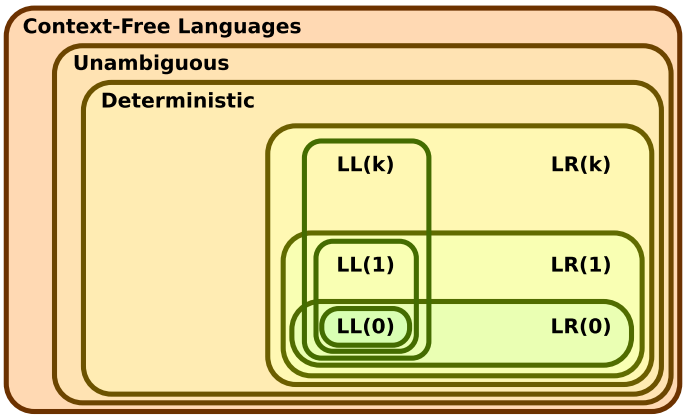
$$
\begin{array}{lcl}
B & \rightarrow & V\bullet \mid \texttt{not}\ B \mid V\ \texttt{or}\ B \mid V\ \texttt{and}\ B \\
V & \rightarrow & \texttt{true} \mid \texttt{false}
\end{array}
$$

1. Transform the grammar by left-factorising the productions of $B$ with a common prefix.
2. Build the parsing table.

# A Hierarchy of Context-Free Languages



# A Family of Parsing Algorithms




Top-Down

- Deterministic / predictive
  - LL: reads input Left-to-right, produces Leftmost derivation.
    - LL(k): lookahead k tokens.
    - LL(*): lookahead a regular expression. [ANTLR]
- Non-deterministic: add backtracking to the above.

Bottom-up

- (LA)LR(k): reads input Left-to-right, produces Rightmost derivation. [Bison,Yacc,...]

# LL Parsing — A Top-Down Approach

$E \to A\ T$
$T \to +\ E \mid *\ E \mid \bullet$
$A \to 0$

| | Input '0' | Input '$+$' | Input '$*$' | Input '$\bullet$' |
|---|---|---|---|---|
| Variable $E$ | $E \to A\ T$ | $\times$ | $\times$ | $\times$ |
| Variable $T$ | $\times$ | $T \to +\ E$ | $T \to *\ E$ | $T \to \bullet$ |
| Variable $A$ | $A \to 0$ | $\times$ | $\times$ | $\times$ |

We can use the table to parse a string like a DPDA with two basic rules:

1. CHOOSE: When the top of the stack is a non-terminal $T$ and the input starts with terminal $a$... choose production $(T, a)$ from the table and push it on the stack. Do not consume $a$
2. MATCH: When the top of the stack is a terminal $a$ and the input starts with terminal $a$... Pop and consume $a$

# LL Parsing — A Top-Down Approach

$E \to A\ T$
$T \to +\ E \mid *\ E \mid \bullet$
$A \to 0$

| | Input '0' | Input '$+$' | Input '$*$' | Input '$\bullet$' |
|---|---|---|---|---|
| Variable $E$ | $E \to A\ T$ | $\times$ | $\times$ | $\times$ |
| Variable $T$ | $\times$ | $T \to +\ E$ | $T \to *\ E$ | $T \to \bullet$ |
| Variable $A$ | $A \to 0$ | $\times$ | $\times$ | $\times$ |

Example on $0\ +\ 0\ *\ 0\ \bullet$

| ($\leftarrow$) Stack | Input | Rule |
|---|---|---|
| $E$ | $0\ +\ 0\ *\ 0\ \bullet$ | |

# LL Parsing — A Top-Down Approach

$E \to A\ T$
$T \to +\ E \mid *\ E \mid \bullet$
$A \to 0$

| | Input '0' | Input '$+$' | Input '$*$' | Input '$\bullet$' |
|---|---|---|---|---|
| Variable $E$ | $E \to A\ T$ | $\times$ | $\times$ | $\times$ |
| Variable $T$ | $\times$ | $T \to +\ E$ | $T \to *\ E$ | $T \to \bullet$ |
| Variable $A$ | $A \to 0$ | $\times$ | $\times$ | $\times$ |

Example on $0\ +\ 0\ *\ 0\ \bullet$

| ($\leftarrow$) Stack | Input | Rule |
|---|---|---|
| $E$ | $0\ +\ 0\ *\ 0\ \bullet$ | |

# LL Parsing — A Top-Down Approach

$E \to A\ T$
$T \to +\ E \mid *\ E \mid \bullet$
$A \to 0$

| | Input '0' | Input '$+$' | Input '$*$' | Input '$\bullet$' |
|---|---|---|---|---|
| Variable $E$ | $E \to A\ T$ | $\times$ | $\times$ | $\times$ |
| Variable $T$ | $\times$ | $T \to +\ E$ | $T \to *\ E$ | $T \to \bullet$ |
| Variable $A$ | $A \to 0$ | $\times$ | $\times$ | $\times$ |

Example on $0\ +\ 0\ *\ 0\ \bullet$

| ($\leftarrow$) Stack | Input | Rule |
|---|---|---|
| $E$ | $0 + 0 * 0\bullet$ | CHOOSE $E \to AT$ |
| $AT$ | $0 + 0 * 0\bullet$ | |

# LL Parsing — A Top-Down Approach

$E \to A\ T$
$T \to +\ E \mid *\ E \mid \bullet$
$A \to 0$

| | Input '0' | Input '$+$' | Input '$*$' | Input '$\bullet$' |
|---|---|---|---|---|
| Variable $E$ | $E \to A\ T$ | $\times$ | $\times$ | $\times$ |
| Variable $T$ | $\times$ | $T \to +\ E$ | $T \to *\ E$ | $T \to \bullet$ |
| Variable $A$ | $A \to 0$ | $\times$ | $\times$ | $\times$ |

Example on $0\ +\ 0\ *\ 0\ \bullet$

| ($\leftarrow$) Stack | Input | Rule |
|---|---|---|
| $E$ | $0 + 0 * 0\bullet$ | Choose $E \to AT$ |
| $AT$ | $0 + 0 * 0\bullet$ | Choose $A \to 0$ |
| $0T$ | $0 + 0 * 0\bullet$ | |

# LL Parsing — A Top-Down Approach

$E \to A\ T$
$T \to +\ E \mid *\ E \mid \bullet$
$A \to 0$

| | Input '0' | Input '$+$' | Input '$*$' | Input '$\bullet$' |
|---|---|---|---|---|
| Variable $E$ | $E \to A\ T$ | $\times$ | $\times$ | $\times$ |
| Variable $T$ | $\times$ | $T \to +\ E$ | $T \to *\ E$ | $T \to \bullet$ |
| Variable $A$ | $A \to 0$ | $\times$ | $\times$ | $\times$ |

Example on $0\ +\ 0\ *\ 0\ \bullet$

| ($\leftarrow$) Stack | Input | Rule |
|---|---|---|
| $E$ | $0 + 0 * 0\bullet$ | Choose $E \to AT$ |
| $AT$ | $0 + 0 * 0\bullet$ | Choose $A \to 0$ |
| $0T$ | $0 + 0 * 0\bullet$ | Match |
| $T$ | $+0 * 0\bullet$ | |

# LL Parsing — A Top-Down Approach

$E \to A\ T$
$T \to +\ E \mid *\ E \mid \bullet$
$A \to 0$

| | Input '0' | Input '$+$' | Input '$*$' | Input '$\bullet$' |
|---|---|---|---|---|
| Variable $E$ | $E \to A\ T$ | $\times$ | $\times$ | $\times$ |
| Variable $T$ | $\times$ | $T \to +\ E$ | $T \to *\ E$ | $T \to \bullet$ |
| Variable $A$ | $A \to 0$ | $\times$ | $\times$ | $\times$ |

Example on $0\ +\ 0\ *\ 0\ \bullet$

| ($\leftarrow$) Stack | Input | Rule |
|---|---|---|
| $E$ | $0 + 0 * 0\bullet$ | Choose $E \to AT$ |
| $AT$ | $0 + 0 * 0\bullet$ | Choose $A \to 0$ |
| $0T$ | $0 + 0 * 0\bullet$ | Match |
| $T$ | $+0 * 0\bullet$ | Choose $T \to +E$ |
| $+E$ | $+0 * 0\bullet$ | |

# LL Parsing — A Top-Down Approach

$E \to A\ T$
$T \to +\ E \mid *\ E \mid \bullet$
$A \to 0$

| | Input '0' | Input '+' | Input '∗' | Input '•' |
|---|---|---|---|---|
| Variable $E$ | $E \to A\ T$ | × | × | × |
| Variable $T$ | × | $T \to +\ E$ | $T \to *\ E$ | $T \to \bullet$ |
| Variable $A$ | $A \to 0$ | × | × | × |

Example on $0\ +\ 0\ *\ 0\ \bullet$

| (←) Stack | Input | Rule |
|---|---|---|
| $E$ | $0 + 0 * 0\bullet$ | Choose $E \to AT$ |
| $AT$ | $0 + 0 * 0\bullet$ | Choose $A \to 0$ |
| $0T$ | $0 + 0 * 0\bullet$ | Match |
| $T$ | $+0 * 0\bullet$ | Choose $T \to +E$ |
| $+E$ | $+0 * 0\bullet$ | Match |
| $E$ | $0 * 0\bullet$ | |

# LL Parsing — A Top-Down Approach

$E \to A\ T$
$T \to +\ E \mid *\ E \mid \bullet$
$A \to 0$

| | Input '0' | Input '+' | Input '∗' | Input '•' |
|---|---|---|---|---|
| Variable $E$ | $E \to A\ T$ | × | × | × |
| Variable $T$ | × | $T \to +\ E$ | $T \to *\ E$ | $T \to \bullet$ |
| Variable $A$ | $A \to 0$ | × | × | × |

Example on $0\ +\ 0\ *\ 0\ \bullet$

| (←) Stack | Input | Rule |
|---|---|---|
| $E$ | $0 + 0 * 0\bullet$ | Choose $E \to AT$ |
| $AT$ | $0 + 0 * 0\bullet$ | Choose $A \to 0$ |
| $0T$ | $0 + 0 * 0\bullet$ | Match |
| $T$ | $+0 * 0\bullet$ | Choose $T \to +E$ |
| $+E$ | $+0 * 0\bullet$ | Match |
| $E$ | $0 * 0\bullet$ | Choose $E \to AT$ |
| $AT$ | $0 * 0\bullet$ | |

# LL Parsing — A Top-Down Approach

$E \to A\ T$
$T \to +\ E \mid *\ E \mid \bullet$
$A \to 0$

| | Input '0' | Input '+' | Input '∗' | Input '•' |
|---|---|---|---|---|
| Variable $E$ | $E \to A\ T$ | × | × | × |
| Variable $T$ | × | $T \to +\ E$ | $T \to *\ E$ | $T \to \bullet$ |
| Variable $A$ | $A \to 0$ | × | × | × |

Example on $0\ +\ 0\ *\ 0\ \bullet$

| (←) Stack | Input | Rule |
|---|---|---|
| $E$ | $0 + 0 * 0\bullet$ | Choose $E \to AT$ |
| $AT$ | $0 + 0 * 0\bullet$ | Choose $A \to 0$ |
| $0T$ | $0 + 0 * 0\bullet$ | Match |
| $T$ | $+0 * 0\bullet$ | Choose $T \to +E$ |
| $+E$ | $+0 * 0\bullet$ | Match |
| $E$ | $0 * 0\bullet$ | Choose $E \to AT$ |
| $AT$ | $0 * 0\bullet$ | Choose $A \to 0$ |
| $0T$ | $0 * 0\bullet$ | |

# LL Parsing — A Top-Down Approach

$E \to A\ T$
$T \to +\ E \mid *\ E \mid \bullet$
$A \to 0$

| | Input '0' | Input '+' | Input '*' | Input '•' |
|---|---|---|---|---|
| Variable $E$ | $E \to A\ T$ | × | × | × |
| Variable $T$ | × | $T \to +\ E$ | $T \to *\ E$ | $T \to \bullet$ |
| Variable $A$ | $A \to 0$ | × | × | × |

Example on $0\ +\ 0\ *\ 0\ \bullet$

| (←) Stack | Input | Rule |
|---|---|---|
| $E$ | $0 + 0 * 0\bullet$ | Choose $E \to AT$ |
| $AT$ | $0 + 0 * 0\bullet$ | Choose $A \to 0$ |
| $0T$ | $0 + 0 * 0\bullet$ | Match |
| $T$ | $+0 * 0\bullet$ | Choose $T \to +E$ |
| $+E$ | $+0 * 0\bullet$ | Match |
| $E$ | $0 * 0\bullet$ | Choose $E \to AT$ |
| $AT$ | $0 * 0\bullet$ | Choose $A \to 0$ |
| $0T$ | $0 * 0\bullet$ | Match |
| $T$ | $*0\bullet$ | |

# LL Parsing — A Top-Down Approach

$E \to A\ T$
$T \to +\ E \mid *\ E \mid \bullet$
$A \to 0$

| | Input '0' | Input '+' | Input '*' | Input '•' |
|---|---|---|---|---|
| Variable $E$ | $E \to A\ T$ | × | × | × |
| Variable $T$ | × | $T \to +\ E$ | $T \to *\ E$ | $T \to \bullet$ |
| Variable $A$ | $A \to 0$ | × | × | × |

Example on $0\ +\ 0\ *\ 0\ \bullet$

| (←) Stack | Input | Rule |
|---|---|---|
| $E$ | $0 + 0 * 0\bullet$ | Choose $E \to AT$ |
| $AT$ | $0 + 0 * 0\bullet$ | Choose $A \to 0$ |
| $0T$ | $0 + 0 * 0\bullet$ | Match |
| $T$ | $+0 * 0\bullet$ | Choose $T \to +E$ |
| $+E$ | $+0 * 0\bullet$ | Match |
| $E$ | $0 * 0\bullet$ | Choose $E \to AT$ |
| $AT$ | $0 * 0\bullet$ | Choose $A \to 0$ |
| $0T$ | $0 * 0\bullet$ | Match |
| $T$ | $*0\bullet$ | Choose $T \to *E$ |
| $*E$ | $*0\bullet$ | |

# LL Parsing — A Top-Down Approach

$E \to A\ T$
$T \to +\ E \mid *\ E \mid \bullet$
$A \to 0$

| | Input '0' | Input '+' | Input '*' | Input '•' |
|---|---|---|---|---|
| Variable $E$ | $E \to A\ T$ | × | × | × |
| Variable $T$ | × | $T \to +\ E$ | $T \to *\ E$ | $T \to \bullet$ |
| Variable $A$ | $A \to 0$ | × | × | × |

Example on $0\ +\ 0\ *\ 0\ \bullet$

| (←) Stack | Input | Rule |
|---|---|---|
| $E$ | $0 + 0 * 0\bullet$ | Choose $E \to AT$ |
| $AT$ | $0 + 0 * 0\bullet$ | Choose $A \to 0$ |
| $0T$ | $0 + 0 * 0\bullet$ | Match |
| $T$ | $+0 * 0\bullet$ | Choose $T \to +E$ |
| $+E$ | $+0 * 0\bullet$ | Match |
| $E$ | $0 * 0\bullet$ | Choose $E \to AT$ |
| $AT$ | $0 * 0\bullet$ | Choose $A \to 0$ |
| $0T$ | $0 * 0\bullet$ | Match |
| $T$ | $*0\bullet$ | Choose $T \to *E$ |
| $*E$ | $*0\bullet$ | Match |
| $E$ | $0\bullet$ | |

# LL Parsing — A Top-Down Approach

$E \to A\ T$
$T \to +\ E \mid *\ E \mid \bullet$
$A \to 0$

| | Input '0' | Input '+' | Input '*' | Input '•' |
|---|---|---|---|---|
| Variable $E$ | $E \to A\ T$ | × | × | × |
| Variable $T$ | × | $T \to +\ E$ | $T \to *\ E$ | $T \to \bullet$ |
| Variable $A$ | $A \to 0$ | × | × | × |

Example on $0\ +\ 0\ *\ 0\ \bullet$

| (←) Stack | Input | Rule |
|---|---|---|
| $E$ | $0 + 0 * 0\bullet$ | Choose $E \to AT$ |
| $AT$ | $0 + 0 * 0\bullet$ | Choose $A \to 0$ |
| $0T$ | $0 + 0 * 0\bullet$ | Match |
| $T$ | $+0 * 0\bullet$ | Choose $T \to +E$ |
| $+E$ | $+0 * 0\bullet$ | Match |
| $E$ | $0 * 0\bullet$ | Choose $E \to AT$ |
| $AT$ | $0 * 0\bullet$ | Choose $A \to 0$ |
| $0T$ | $0 * 0\bullet$ | Match |
| $T$ | $*0\bullet$ | Choose $T \to *E$ |
| $*E$ | $*0\bullet$ | Match |
| $E$ | $0\bullet$ | Choose $E \to AT$ |
| $AT$ | $0\bullet$ | |

# LL Parsing — A Top-Down Approach

$E \to A\ T$
$T \to +\ E \mid *\ E \mid \bullet$
$A \to 0$

| | Input '0' | Input '$+$' | Input '$*$' | Input '$\bullet$' |
|---|---|---|---|---|
| Variable $E$ | $E \to A\ T$ | $\times$ | $\times$ | $\times$ |
| Variable $T$ | $\times$ | $T \to +\ E$ | $T \to *\ E$ | $T \to \bullet$ |
| Variable $A$ | $A \to 0$ | $\times$ | $\times$ | $\times$ |

Example on $0\ +\ 0\ *\ 0\ \bullet$

| ($\leftarrow$) Stack | Input | Rule |
|---|---|---|
| $E$ | $0 + 0 * 0\bullet$ | Choose $E \to AT$ |
| $AT$ | $0 + 0 * 0\bullet$ | Choose $A \to 0$ |
| $0T$ | $0 + 0 * 0\bullet$ | Match |
| $T$ | $+0 * 0\bullet$ | Choose $T \to +E$ |
| $+E$ | $+0 * 0\bullet$ | Match |
| $E$ | $0 * 0\bullet$ | Choose $E \to AT$ |
| $AT$ | $0 * 0\bullet$ | Choose $A \to 0$ |
| $0T$ | $0 * 0\bullet$ | Match |
| $T$ | $*0\bullet$ | Choose $T \to *E$ |
| $*E$ | $*0\bullet$ | Match |
| $E$ | $0\bullet$ | Choose $E \to AT$ |
| $AT$ | $0\bullet$ | Choose $A \to 0$ |

# LL Parsing — A Top-Down Approach

$E \to A\ T$
$T \to +\ E \mid *\ E \mid \bullet$
$A \to 0$

| | Input '0' | Input '$+$' | Input '$*$' | Input '$\bullet$' |
|---|---|---|---|---|
| Variable $E$ | $E \to A\ T$ | $\times$ | $\times$ | $\times$ |
| Variable $T$ | $\times$ | $T \to +\ E$ | $T \to *\ E$ | $T \to \bullet$ |
| Variable $A$ | $A \to 0$ | $\times$ | $\times$ | $\times$ |

Example on $0\ +\ 0\ *\ 0\ \bullet$

| ($\leftarrow$) Stack | Input | Rule |
|---|---|---|
| $E$ | $0 + 0 * 0\bullet$ | Choose $E \to AT$ |
| $AT$ | $0 + 0 * 0\bullet$ | Choose $A \to 0$ |
| $0T$ | $0 + 0 * 0\bullet$ | Match |
| $T$ | $+0 * 0\bullet$ | Choose $T \to +E$ |
| $+E$ | $+0 * 0\bullet$ | Match |
| $E$ | $0 * 0\bullet$ | Choose $E \to AT$ |
| $AT$ | $0 * 0\bullet$ | Choose $A \to 0$ |
| $0T$ | $0 * 0\bullet$ | Match |
| $T$ | $*0\bullet$ | Choose $T \to *E$ |
| $*E$ | $*0\bullet$ | Match |
| $E$ | $0\bullet$ | Choose $E \to AT$ |
| $AT$ | $0\bullet$ | Choose $A \to 0$ |
| $0T$ | $0\bullet$ | |

# LL Parsing — A Top-Down Approach

$E \to A\ T$
$T \to +\ E \mid *\ E \mid \bullet$
$A \to 0$

| | Input '0' | Input '+' | Input '*' | Input '•' |
|---|---|---|---|---|
| Variable $E$ | $E \to A\ T$ | × | × | × |
| Variable $T$ | × | $T \to +\ E$ | $T \to *\ E$ | $T \to \bullet$ |
| Variable $A$ | $A \to 0$ | × | × | × |

Example on $0\ +\ 0\ *\ 0\ \bullet$

| (←) Stack | Input | Rule |
|---|---|---|
| $E$ | $0 + 0 * 0\bullet$ | Choose $E \to AT$ |
| $AT$ | $0 + 0 * 0\bullet$ | Choose $A \to 0$ |
| $0T$ | $0 + 0 * 0\bullet$ | Match |
| $T$ | $+0 * 0\bullet$ | Choose $T \to +E$ |
| $+E$ | $+0 * 0\bullet$ | Match |
| $E$ | $0 * 0\bullet$ | Choose $E \to AT$ |
| $AT$ | $0 * 0\bullet$ | Choose $A \to 0$ |
| $0T$ | $0 * 0\bullet$ | Match |
| $T$ | $*0\bullet$ | Choose $T \to *E$ |
| $*E$ | $*0\bullet$ | Match |
| $E$ | $0\bullet$ | Choose $E \to AT$ |
| $AT$ | $0\bullet$ | Choose $A \to 0$ |
| $0T$ | $0\bullet$ | Match |
| $T$ | $\bullet$ | |

# LL Parsing — A Top-Down Approach

$E \to A\ T$
$T \to +\ E \mid *\ E \mid \bullet$
$A \to 0$

| | Input '0' | Input '+' | Input '$*$' | Input '$\bullet$' |
|---|---|---|---|---|
| Variable $E$ | $E \to A\ T$ | $\times$ | $\times$ | $\times$ |
| Variable $T$ | $\times$ | $T \to +\ E$ | $T \to *\ E$ | $T \to \bullet$ |
| Variable $A$ | $A \to 0$ | $\times$ | $\times$ | $\times$ |

Example on $0\ +\ 0\ *\ 0\ \bullet$

| ($\leftarrow$) Stack | Input | Rule |
|---|---|---|
| $E$ | $0 + 0 * 0\bullet$ | Choose $E \to AT$ |
| $AT$ | $0 + 0 * 0\bullet$ | Choose $A \to 0$ |
| $0T$ | $0 + 0 * 0\bullet$ | Match |
| $T$ | $+0 * 0\bullet$ | Choose $T \to +E$ |
| $+E$ | $+0 * 0\bullet$ | Match |
| $E$ | $0 * 0\bullet$ | Choose $E \to AT$ |
| $AT$ | $0 * 0\bullet$ | Choose $A \to 0$ |
| $0T$ | $0 * 0\bullet$ | Match |
| $T$ | $*0\bullet$ | Choose $T \to *E$ |
| $*E$ | $*0\bullet$ | Match |
| $E$ | $0\bullet$ | Choose $E \to AT$ |
| $AT$ | $0\bullet$ | Choose $A \to 0$ |
| $0T$ | $0\bullet$ | Match |
| $T$ | $\bullet$ | Choose $T \to \bullet$ |
| $\bullet$ | $\bullet$ | |

# LL Parsing — A Top-Down Approach

$E \to A\ T$
$T \to +\ E \mid *\ E \mid \bullet$
$A \to 0$

| | Input '0' | Input '+' | Input '$*$' | Input '$\bullet$' |
|---|---|---|---|---|
| Variable $E$ | $E \to A\ T$ | $\times$ | $\times$ | $\times$ |
| Variable $T$ | $\times$ | $T \to +\ E$ | $T \to *\ E$ | $T \to \bullet$ |
| Variable $A$ | $A \to 0$ | $\times$ | $\times$ | $\times$ |

Example on $0\ +\ 0\ *\ 0\ \bullet$

| ($\leftarrow$) Stack | Input | Rule |
|---|---|---|
| $E$ | $0 + 0 * 0\bullet$ | Choose $E \to AT$ |
| $AT$ | $0 + 0 * 0\bullet$ | Choose $A \to 0$ |
| $0T$ | $0 + 0 * 0\bullet$ | Match |
| $T$ | $+0 * 0\bullet$ | Choose $T \to +E$ |
| $+E$ | $+0 * 0\bullet$ | Match |
| $E$ | $0 * 0\bullet$ | Choose $E \to AT$ |
| $AT$ | $0 * 0\bullet$ | Choose $A \to 0$ |
| $0T$ | $0 * 0\bullet$ | Match |
| $T$ | $*0\bullet$ | Choose $T \to *E$ |
| $*E$ | $*0\bullet$ | Match |
| $E$ | $0\bullet$ | Choose $E \to AT$ |
| $AT$ | $0\bullet$ | Choose $A \to 0$ |
| $0T$ | $0\bullet$ | Match |
| $T$ | $\bullet$ | Choose $T \to \bullet$ |
| $\bullet$ | $\bullet$ | Match |

# Exercise session

Do the exercises in the next slides.
If you are done keep working on the mandatory assignment.

# Exercise 4.4

Consider this PDA $P = (\{q\}, \{0,1,2\}, \{0,1,2,S\}, \delta, q, S)$ where

$$\begin{aligned}
\delta(q, \epsilon, S) &= \{(q, 0S0), (q, 1S1), (q, 2)\} \\
\delta(q, 0, 0) &= \{(q, \epsilon)\} \\
\delta(q, 1, 1) &= \{(q, \epsilon)\} \\
\delta(q, 2, 2) &= \{(q, \epsilon)\}
\end{aligned}$$

and assume it accepts words by empty stack.

1. Use the transformation of the slides to convert $P$ into an equivalent PDA $P'$ that accepts words by final state.
2. Use the transformation of the slides to convert $P'$ into an equivalent PDA $P''$ that accepts words by stack.

# Exercise 4.5

**Exercise 6.1.1:** Suppose the PDA $P = (\{q, p\}, \{0, 1\}, \{Z_0, X\}, \delta, q, Z_0, \{p\})$ has the following transition function:

1. $\delta(q, 0, Z_0) = \{(q, XZ_0)\}$.
2. $\delta(q, 0, X) = \{(q, XX)\}$.
3. $\delta(q, 1, X) = \{(q, X)\}$.
4. $\delta(q, \epsilon, X) = \{(p, \epsilon)\}$.
5. $\delta(p, \epsilon, X) = \{(p, \epsilon)\}$.
6. $\delta(p, 1, X) = \{(p, XX)\}$.
7. $\delta(p, 1, Z_0) = \{(p, \epsilon)\}$.

Starting from the initial ID $(q, w, Z_0)$, show all the reachable ID's when the input $w$ is:

* a) 01.

b) 0011.

c) 010.

# Exercise 4.6

**Exercise 6.2.1 :** Design a PDA to accept each of the following languages. You may accept either by final state or by empty stack, whichever is more convenient.

* a) $\{0^n 1^n \mid n \geq 1\}$.

b) The set of all strings of 0's and 1's such that no prefix has more 1's than 0's.

c) The set of all strings of 0's and 1's with an equal number of 0's and 1's.

# Exercise 4.7

**Exercise 6.2.5:** PDA $P = (\{q_0, q_1, q_2, q_3, f\}, \{a, b\}, \{Z_0, A, B\}, \delta, q_0, Z_0, \{f\})$ has the following rules defining $\delta$:

$$\begin{array}{lll}
\delta(q_0, a, Z_0) = (q_1, AAZ_0) & \delta(q_0, b, Z_0) = (q_2, BZ_0) & \delta(q_0, \epsilon, Z_0) = (f, \epsilon) \\
\delta(q_1, a, A) = (q_1, AAA) & \delta(q_1, b, A) = (q_1, \epsilon) & \delta(q_1, \epsilon, Z_0) = (q_0, Z_0) \\
\delta(q_2, a, B) = (q_3, \epsilon) & \delta(q_2, b, B) = (q_2, BB) & \delta(q_2, \epsilon, Z_0) = (q_0, Z_0) \\
\delta(q_3, \epsilon, B) = (q_2, \epsilon) & \delta(q_3, \epsilon, Z_0) = (q_1, AZ_0) &
\end{array}$$

Note that, since each of the sets above has only one choice of move, we have omitted the set brackets from each of the rules.

* a) Give an execution trace (sequence of ID's) showing that string $bab$ is in $L(P)$.

b) Give an execution trace showing that $abb$ is in $L(P)$.

# Exercise 4.8

Use the parsing table of Exercise 4.3 and the approach of slide "LL parsing - a top down approach" to parse the following string top-down:

`true and true or true` •

Write the content of the stack, the input and the rule used as done in the slide.

# Exercise 4.9

Use the approach of slide "LL parsing - a top down approach" to parse the expression $0 * 0 + 0\bullet$ using LL parsing. Write the content of the stack, the input and the rule used as done in the slide.

# Exercise 4.10

Consider the following grammar for a programming language whose main actions ($A$) are `ping` and `pong`, which can be composed in programs ($P$) with sequential composition (`;`), non-deterministic choice (`or`) or parallel execution (`and`). Programs can also `stop` or be grouped with curly brackets.

$$\begin{array}{lcl} P & \rightarrow & A \texttt{ ; } P \mid A \texttt{ or } P \mid A \texttt{ and } P \mid \texttt{stop} \mid (P) \\ A & \rightarrow & \texttt{ping} \mid \texttt{pong} \end{array}$$

1. Build a non-deterministic PDA that recognises programs with the above syntax using the construction seen in class.
2. Use the PDA to parse the program `ping and pong ; stop`: run the PDF by drawing the sequence of configurations that lead to successfully parsing the string.
3. Use left-factorisation to transform the grammar into an equivalent one for which you can provide deterministic parsing table.
4. Build the parsing table.
5. Use the table to parse the same program using "LL parsing" (as in Excercise 4.9).