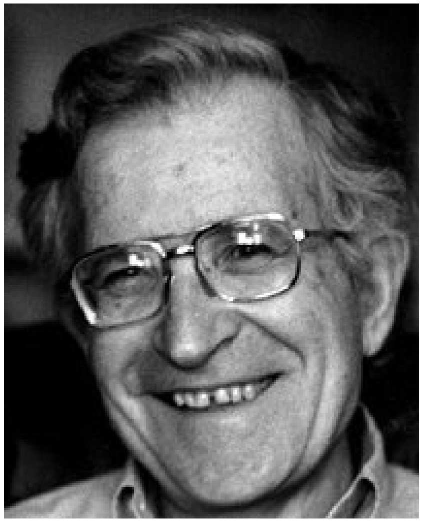
CS311 Computational Structures

Context-free Languages: Grammars and Automata

Lecture 8

Andrew Black
Andrew Tolmach

Chomsky hierarchy



In 1957, Noam Chomsky published *Syntactic Structures*, an landmark book that defined the so-called Chomsky hierarchy of languages

original name	language generated	productions:
Type-3 Grammars $\cap$	Regular	$A \to \alpha$ and $A \to \alpha B$
Type-2 Grammars $\cap$	Contex-free	$A \to \gamma$
Type-1 Grammars $\cap$	Context-sensitive	$\alpha A \beta \to \alpha \gamma \beta$
Type-0 Grammars	Recursively-enumerable	no restriction

A, B: variables, a, b terminals, α, β sequences of terminals and variables

Regular languages

- Closed under $\cup \cap * \cdot$ and $\overline{}$
- Recognizable by finite-state automata
- Denoted by Regular Expressions
- Generated by Regular Grammars

Context-free Grammars

Context-free Grammars

- More general productions than regular grammars

  $S \rightarrow w$ where w is any string of terminals and non-terminals

Context-free Grammars

- More general productions than regular grammars

  $S \rightarrow w$ where w is any string of terminals and non-terminals

- What languages do these grammars generate?

  $S \rightarrow (A)$
  $A \rightarrow \varepsilon \mid aA \mid ASA$

Portland State UNIVERSITY

Context-free Grammars

- More general productions than regular grammars

  $S \rightarrow w$ where w is any string of terminals and non-terminals

- What languages do these grammars generate?

  $S \rightarrow (A)$
  $A \rightarrow \varepsilon \mid aA \mid ASA$

  $S \rightarrow \varepsilon \mid aSb$

Portland State University

Context-free languages more general than regular languages

Context-free languages more general than regular languages

- $\{a^n b^n \mid n \geq 0\}$ is not regular

Portland State UNIVERSITY

Context-free languages more general than regular languages

- $\{a^n b^n \mid n \geq 0\}$ is not regular
  - but it *is* context-free

Context-free languages more general than regular languages

- $\{a^n b^n \mid n \geq 0\}$ is not regular
  - but it *is* context-free
- Why are they called “context-free”?

Portland State University

Context-free languages more general than regular languages

- $\{a^n b^n \mid n \geq 0\}$ is not regular
  - but it *is* context-free
- Why are they called "context-free"?
  - Context-sensitive grammars allow more than one symbol on the lhs of productions

Context-free languages more general than regular languages

- $\{a^n b^n \mid n \geq 0\}$ is not regular
  - but it *is* context-free
- Why are they called "context-free"?
  - Context-sensitive grammars allow more than one symbol on the lhs of productions
    - xAy → x(S)y can only be applied to the non-terminal A when it is in the *context* of x and y

      pin

Portland State UNIVERSITY

Context-free grammars are widely used for programming languages

- From the definition of Algol-60:

BNF
Backus-Naur Form

```
procedure_identifier::= identifier.
actual_parameter::= string_literal | expression | array_identifier | switch_identifier | procedure_identifier.
letter_string::= letter | letter_string letter.
parameter_delimiter::= "," | ")" letter_string ":" "(".
actual_parameter_list::= actual_parameter | actual_parameter_list parameter_delimiter actual_parameter.
actual_parameter_part::= empty | "(" actual_parameter_list} ")".
function_designator::= procedure_identifier actual_parameter_part.
```

- We say: “most programming languages are context-free”
  - This isn’t strictly true
  - … but we pretend that it is!

Example

adding_operator::= "+" | "−" .
multiplying_operator::= "×" | "/" | "÷" .
primary::= **unsigned_number** | **variable** | **function_designator** | "(" **arithmetic_expression** ")".
factor::= **primary** | **factor** | **factor power primary**.
term::= **factor** | **term multiplying_operator factor**.
simple_arithmetic_expression::= **term** | **adding_operator term** |
simple_arithmetic_expression adding_operator term.
if_clause::= **if Boolean_expression then**.
arithmetic_expression::= **simple_arithmetic_expression** |
if_clause simple_arithmetic_expression else arithmetic_expression.

if a < 0 then U+V else if a * b < 17 then U/V else if k <> y then V/U else 0

Example derivation in a Grammar

- Grammar: start symbol is A

$$A \rightarrow \mathrm{a}A\mathrm{a}$$
$$A \rightarrow B$$
$$B \rightarrow \mathrm{b}B$$
$$B \rightarrow \varepsilon$$

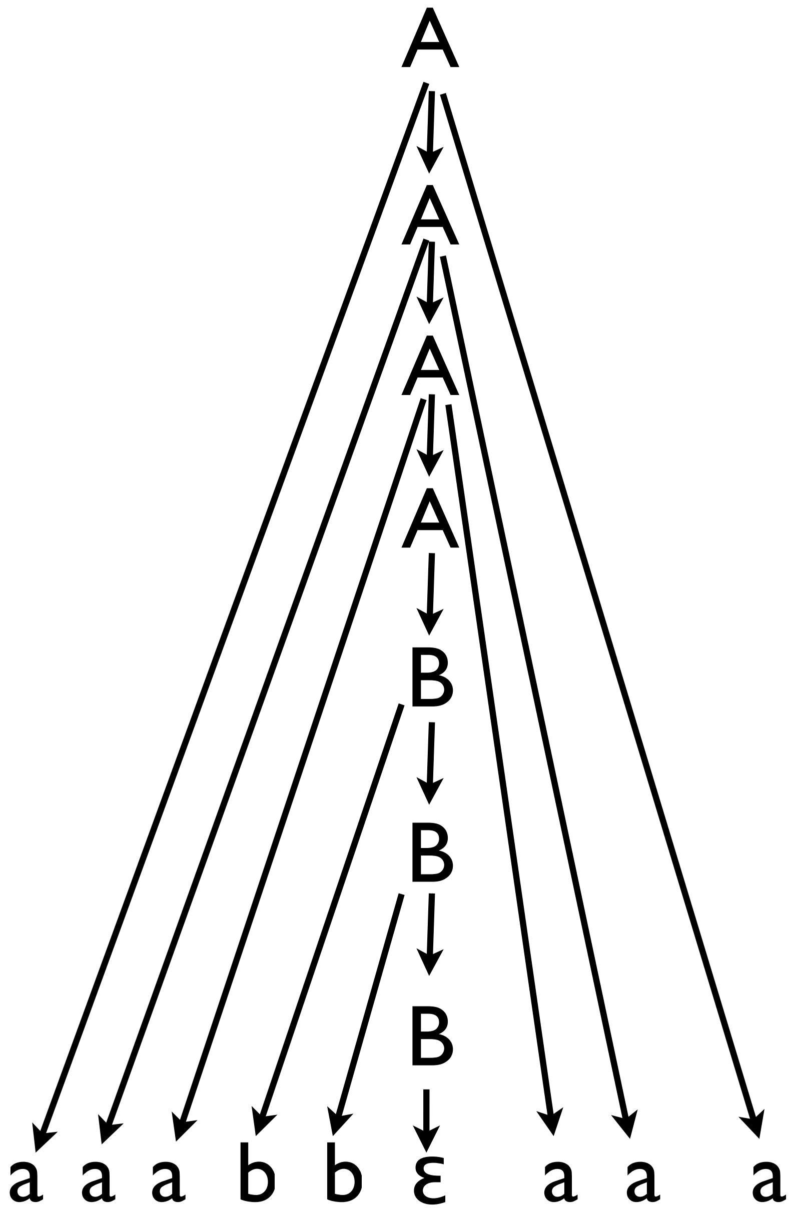
- Sample Derivation:

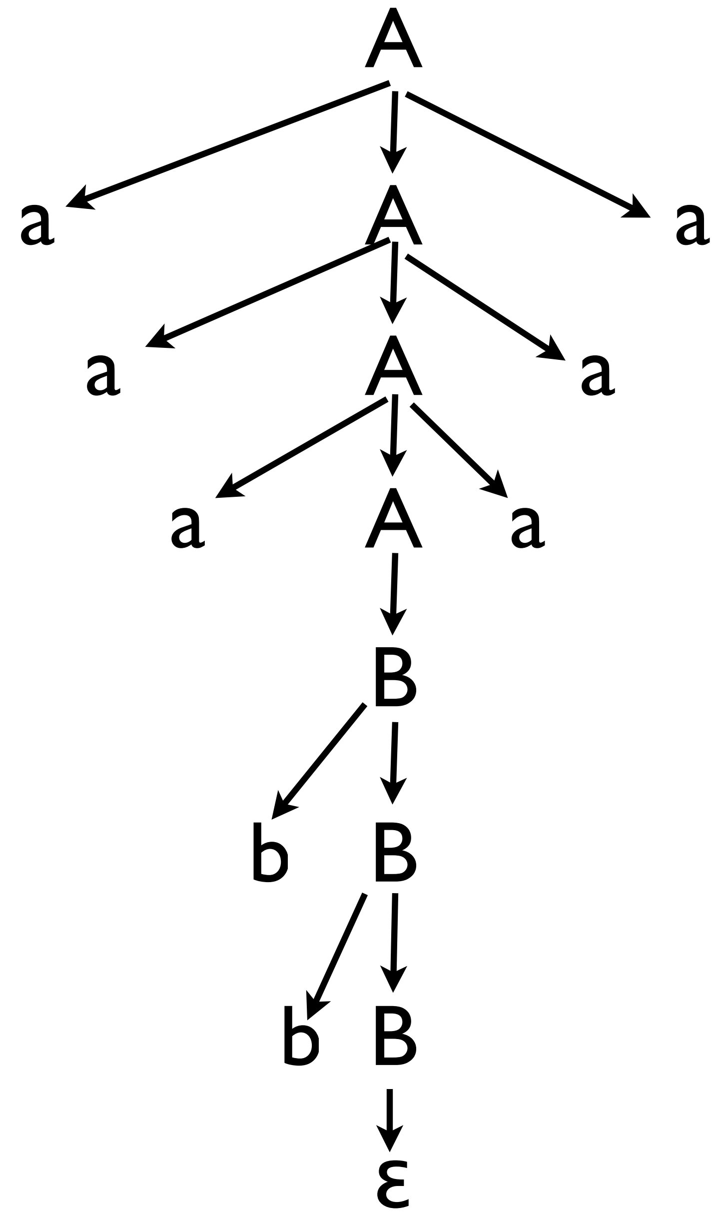
$$\underline{A} \Rightarrow \mathrm{a}\underline{A}\mathrm{a} \Rightarrow \mathrm{aa}\underline{A}\mathrm{aa} \Rightarrow \mathrm{aaa}\underline{A}\mathrm{aaa} \Rightarrow \mathrm{aaa}\underline{B}\mathrm{aaa}$$

$$\Rightarrow \mathrm{aaab}\underline{B}\mathrm{aaa} \Rightarrow \mathrm{aaabb}\underline{B}\mathrm{aaa} \Rightarrow \mathrm{aaabbaaa}$$

- Language?

Derivations in Tree Form





Arithmetic expressions in a programming language

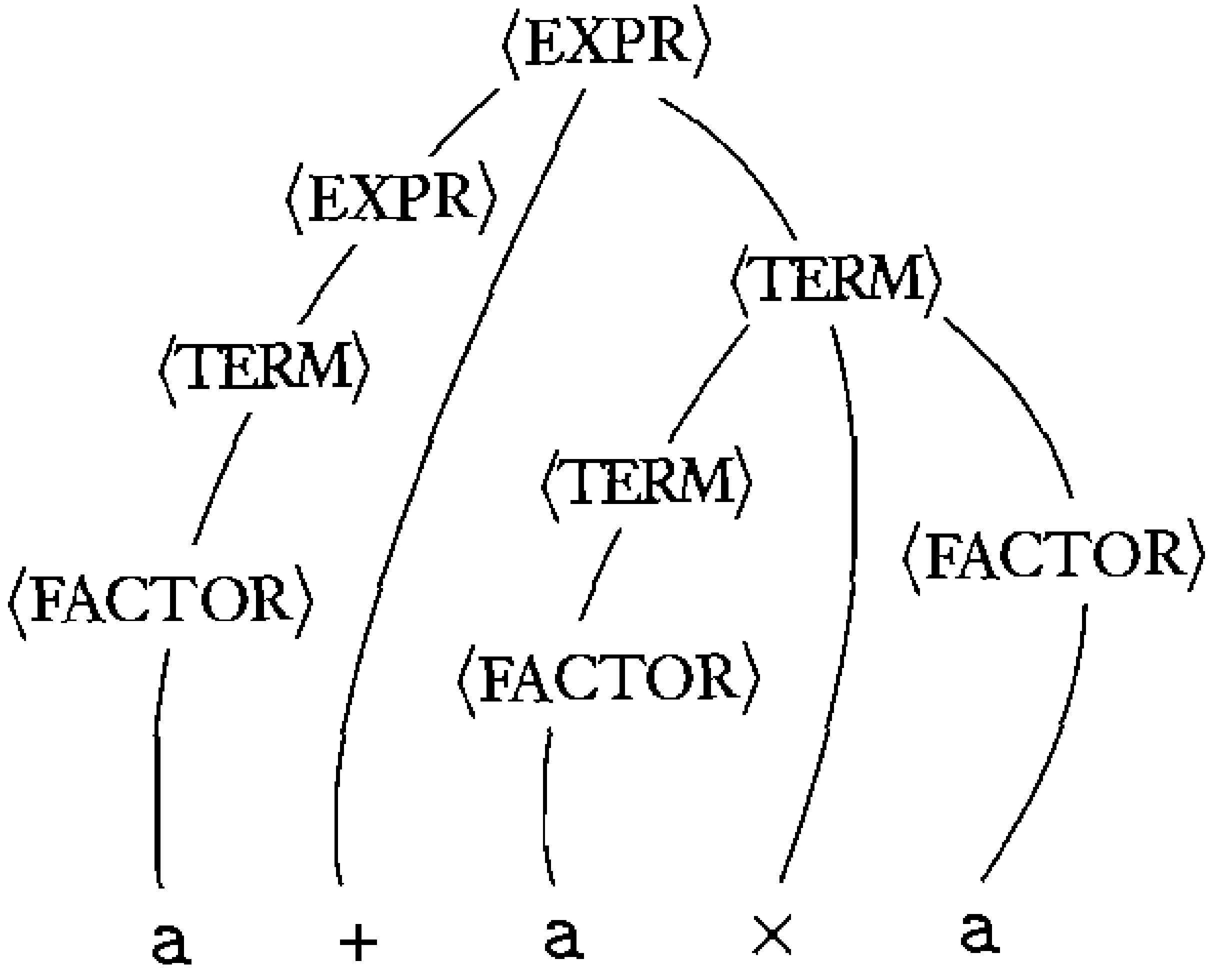
Consider grammar $G_4 = (V, \Sigma, R, \langle\text{EXPR}\rangle)$.

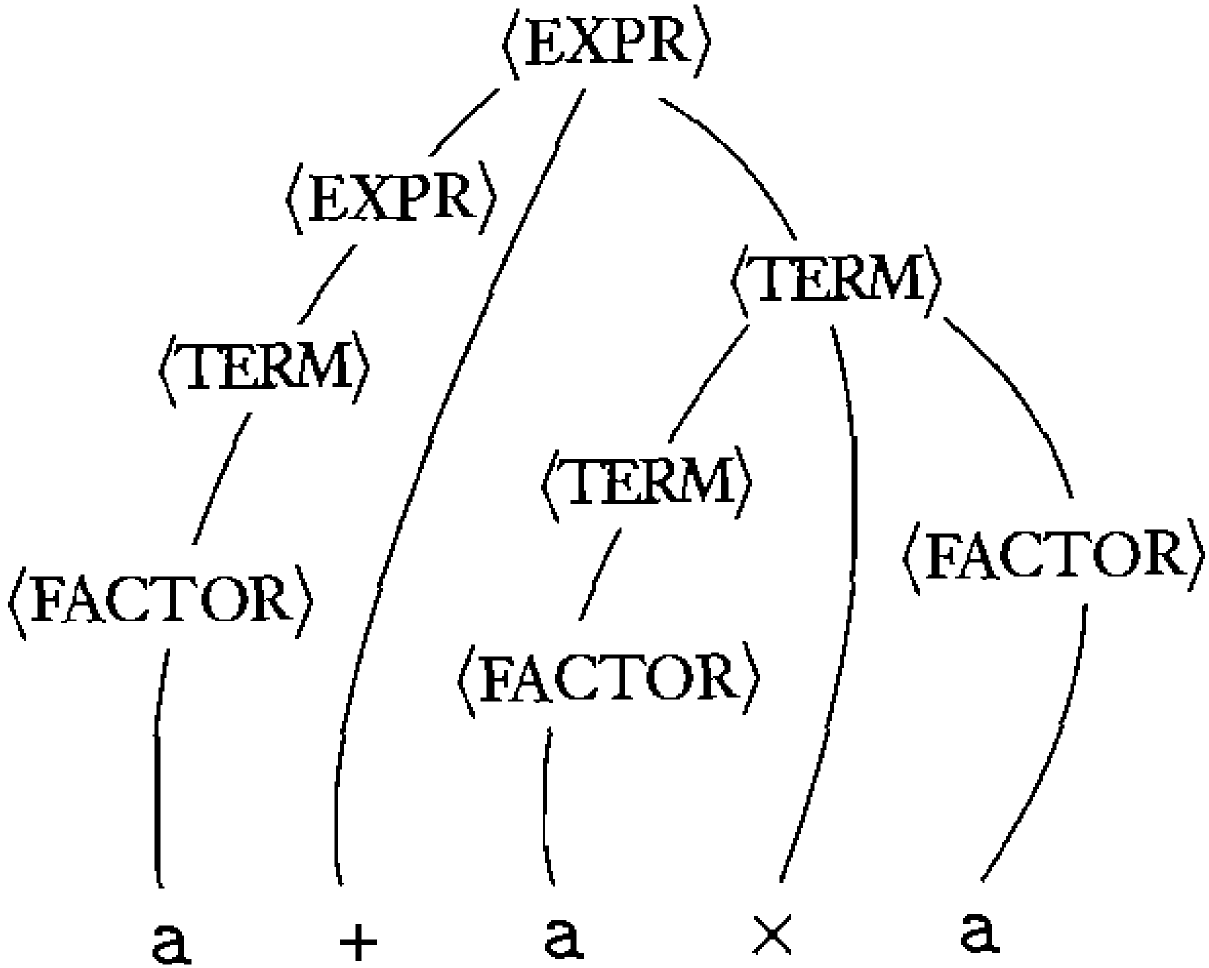
V is $\{\langle\text{EXPR}\rangle, \langle\text{TERM}\rangle, \langle\text{FACTOR}\rangle\}$ and Σ is $\{\mathtt{a}, \mathtt{+}, \mathtt{x}, \mathtt{(}, \mathtt{)}\}$. The rules are

$$\begin{aligned}
\langle\text{EXPR}\rangle &\to \langle\text{EXPR}\rangle\mathtt{+}\langle\text{TERM}\rangle \mid \langle\text{TERM}\rangle \\
\langle\text{TERM}\rangle &\to \langle\text{TERM}\rangle\mathtt{x}\langle\text{FACTOR}\rangle \mid \langle\text{FACTOR}\rangle \\
\langle\text{FACTOR}\rangle &\to \mathtt{(}\,\langle\text{EXPR}\rangle\,\mathtt{)} \mid \mathtt{a}
\end{aligned}$$

- Derive: a + a × a

Portland State UNIVERSITY


⟨EXPR⟩
⟨EXPR⟩
⟨TERM⟩
⟨TERM⟩
⟨FACTOR⟩
⟨TERM⟩
⟨FACTOR⟩
⟨FACTOR⟩
a
+
a
×
a



Notice how the grammar gives the meaning a + (a×a)

Grammars in real computing

- CFG's are universally used to describe the syntax of programming languages
  - Perfectly suited to describing **recursive** syntax of expressions and statements
  - Tools like compilers must **parse** programs; parsers can be generated automatically from CFG's
  - Real languages usually have a few non-CF bits
- CFG's are also used in XML DTD's

Formal definition of CFG

- A Context-free grammar is a 4-tuple (V, Σ, R, S) where

1. V is a finite set called the **variables** (non-terminals)
2. Σ is a finite set (disjoint from V) called the **terminals**,
3. R is a finite set of **rules**, where each rule maps a variable to a string $s \in (V \cup \Sigma)^*$
4. $S \in V$ is the start symbol

Definition of Derivation

- Let u, v and w be strings in $(V \cup \Sigma)^*$, and let $A \rightarrow w$ be a rule in R,
- then $uAv \Rightarrow uwv$ (read: uAv **yields** uwv)
- We say that $u \stackrel{*}{\Rightarrow} v$ (read: u **derives** v) if

  $u = v$ or there is a sequence $u_1, u_2, \ldots u_k$, $k \geq 0$, s.t. $u \Rightarrow u_1 \Rightarrow u_2 \Rightarrow \ldots \Rightarrow u_k \Rightarrow v$

- The **language** of the grammar is

$$\{w \in \Sigma^* \mid S \stackrel{*}{\Rightarrow} w\}$$

Derivations ⇔ Parse Trees

- Each derivation can be viewed as a **parse tree** with variables at the internal nodes and terminals at the leaves
  - Start symbol is at the root
  - Each yield step $uAv \Rightarrow uwv$ where $w=w_1w_2...w_n$ corresponds to a node labeled A with children $w_1,w_2,...,w_n$.
  - The final result in Σ^* can be seen by reading the leaves left-to-right

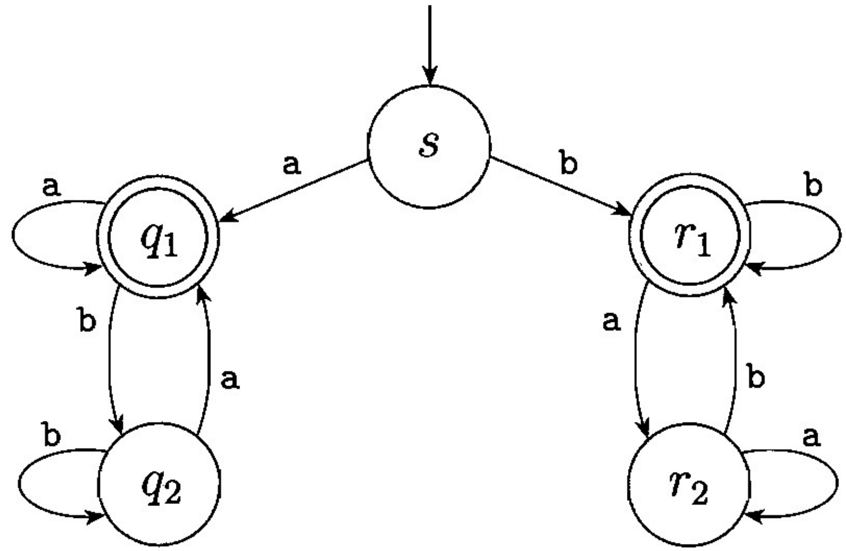
Simple CFG examples

- Find grammars for:
  - L = {w ∈ {a,b}* | w begins and ends with the same symbol}
  - L = {w ∈ {a,b}* | w contains an odd number of a's}
  - $\mathscr{L}[(\varepsilon + 1)(01)^*(\varepsilon + 0)]$
- Draw example derivations

All regular languages have context free grammars

- Proof:
  - Regular language is accepted by an NFA.
  - We can generate a regular grammar from the NFA (Lecture 6, Hein Alg. 11.11)
  - Any regular grammar is also a CFG. (Immediate from the definition of the grammars).

Example



- $S \rightarrow aQ_1$ $S \rightarrow bR_1$
- $Q_1 \rightarrow aQ_1$ $Q_1 \rightarrow bQ_2$
- $Q_2 \rightarrow aQ_1$ $Q_2 \rightarrow bQ_2$
- $R_1 \rightarrow aR_2$ $R_1 \rightarrow bR_1$
- $R_2 \rightarrow aR_2$ $R_2 \rightarrow bR_1$
- $Q_1 \rightarrow \varepsilon$ $R_1 \rightarrow \varepsilon$
    - Resulting grammar may be quite different from one we designed by hand.

Some CFG's generate non-regular languages

- Find grammars for the following languages
  - $L = \{a^n b^m a^n \mid a,b \geq 0\}$
  - $L = \{w \in \{a,b\}^* \mid w$ contains equal numbers of a's and b's$\}$
  - $L = \{ww^R \mid w \in \{a,b\}^*\}$
- Draw example derivations

Ambiguity

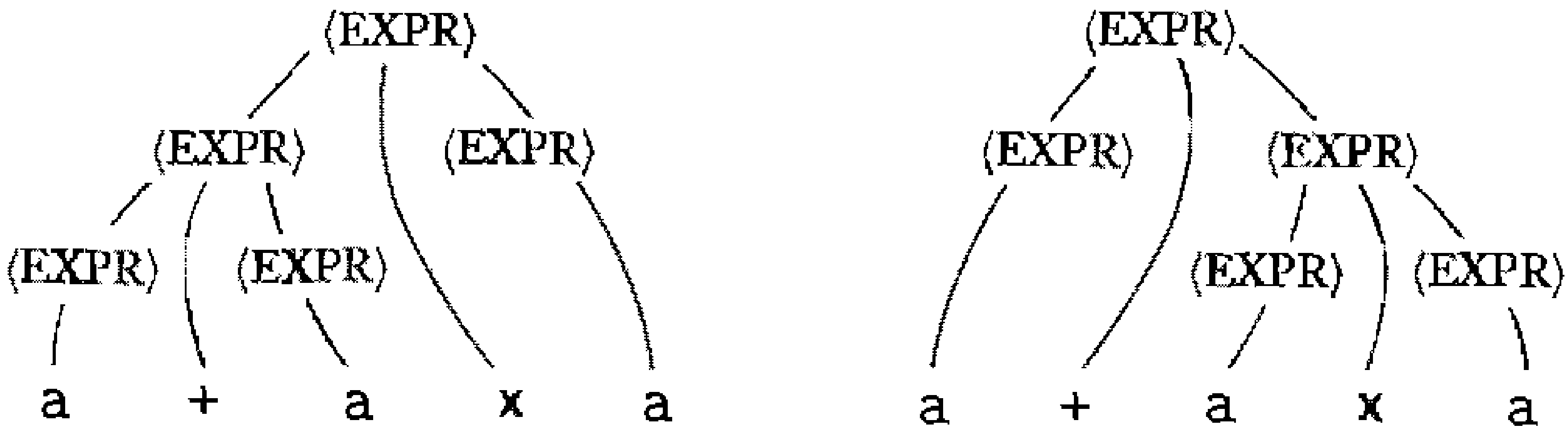
- A grammar in which the same string can be given more than one parse *tree* is **ambiguous**.

- Example: another grammar for arithmetic expressions

$$
\begin{aligned}
\langle \mathrm{EXPR} \rangle \rightarrow \quad & \langle \mathrm{EXPR} \rangle + \langle \mathrm{EXPR} \rangle \mid \\
& \langle \mathrm{EXPR} \rangle \times \langle \mathrm{EXPR} \rangle \mid \\
& (\langle \mathrm{EXPR} \rangle) \\
& \mid a
\end{aligned}
$$

- Derive: $a + a \times a$

- This grammar is ambiguous: there are two *different* parse trees for a + a × a



- Ambiguity is a bad thing if we're interested in the structure of the parse
    - Ambiguity doesn't matter if we're interested only in *defining* a language.

Leftmost Derivations

- In general, in any step of a derivation, there might be several variables that can be reduced by rules of the grammar.
- In a leftmost derivation, we choose to always reduce the leftmost variable.
  - Example: given grammar $S \rightarrow aSb \mid SS \mid \varepsilon$
    - A left-most derivation:

      $\underline{S} \Rightarrow a\underline{S}b \Rightarrow a\underline{S}Sb \Rightarrow aa\underline{S}bSb \Rightarrow aab\underline{S}b \Rightarrow aabb$

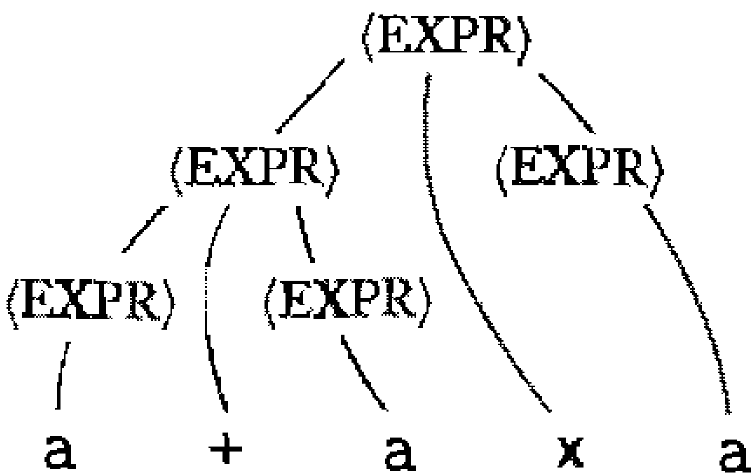
    - A non-left-most derivation:

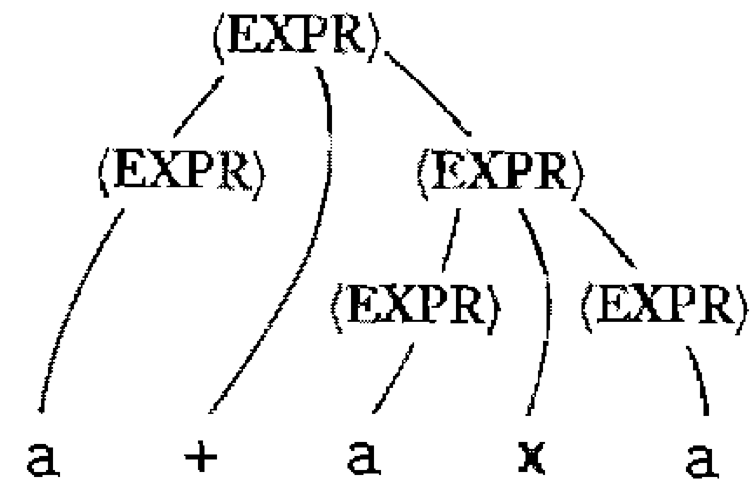
      $\underline{S} \Rightarrow a\underline{S}b \Rightarrow aS\underline{S}b \Rightarrow a\underline{S}b \Rightarrow aa\underline{S}bb \Rightarrow aabb$

Ambiguity *via* left-most derivations

- Every parse tree corresponds to a unique left-most derivation
- So if a grammar has more than one left-most derivation for some string, the grammar is ambiguous
  - Note: merely having two derivations (not necessarily left-most) for one string is **not** enough to show ambiguity

Ambiguity





$\underline{E} \Rightarrow \underline{E} \times E \Rightarrow \underline{E} + E \times E$

$\Rightarrow a + \underline{E} \times E \Rightarrow a + a \times \underline{E}$

$\Rightarrow a + a \times a$

$\underline{E} \Rightarrow \underline{E} + E \Rightarrow a + \underline{E} \Rightarrow$

$a + \underline{E} \times E \Rightarrow a + a \times \underline{E} \Rightarrow$

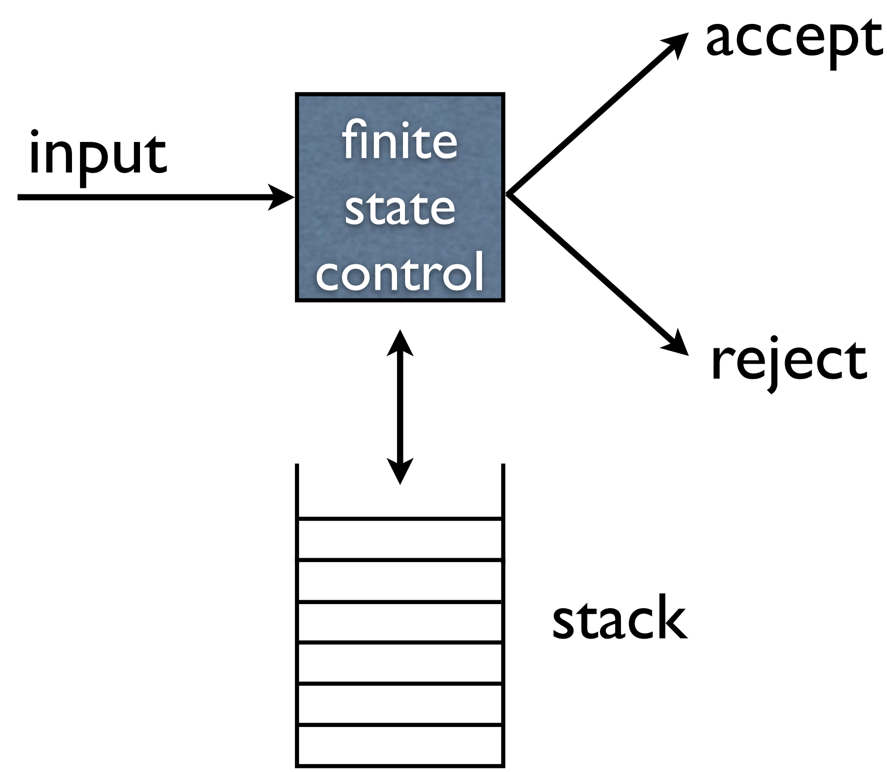
$a + a \times a$

Context-free languages

- Closed under $\cup$, $*$ and $\cdot$, and under $\cap$ with a regular language
  - How do we prove these properties?
- *Not* closed under intersection, complement or difference
- Recognizable by pushdown automata
  - A pushdown automaton is a generalization of a finite-state automaton

Pushdown Automata

- Why can't a FSA recognize $a^n b^n$?
    - "storage" is finite
- How can we fix the problem?
    - add unbounded storage
- What's the simplest kind of unbounded storage
    - a pushdown stack



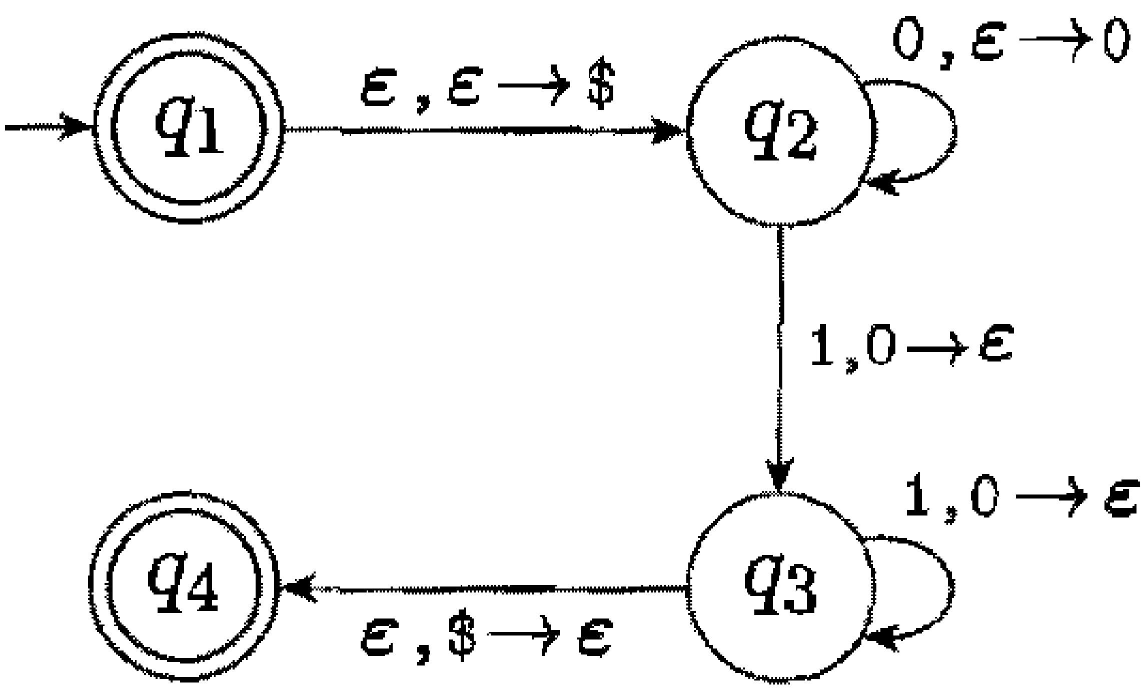
History

- PDAs independently invented by Oettinger [1961] and Schutzenberger [1963]
- Equivalence between PDAs and CFG known to Chomsky in 1961; first published by Evey [1963].

Executing a PDA

- The behavior of a PDA at any point depends on ⟨state,stack,unread input⟩
- PDA begins in start state with stated symbol on the stack
- On each step it optionally reads an input character, pops a stack symbol, and non-deterministically chooses a new state and optionally pushes one or more stack symbols
- PDA accepts if it is in a final state and there is no more input.

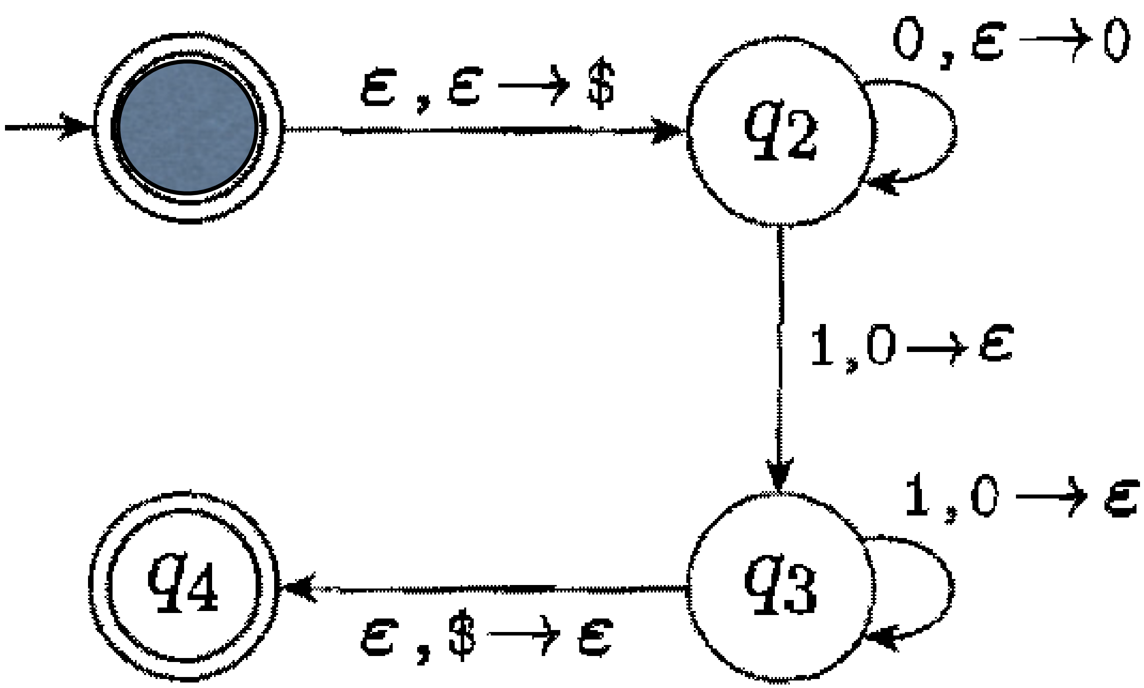
Example PDA

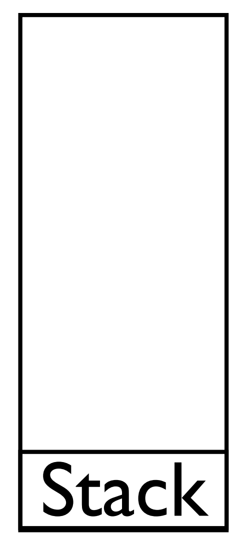


Portland State University

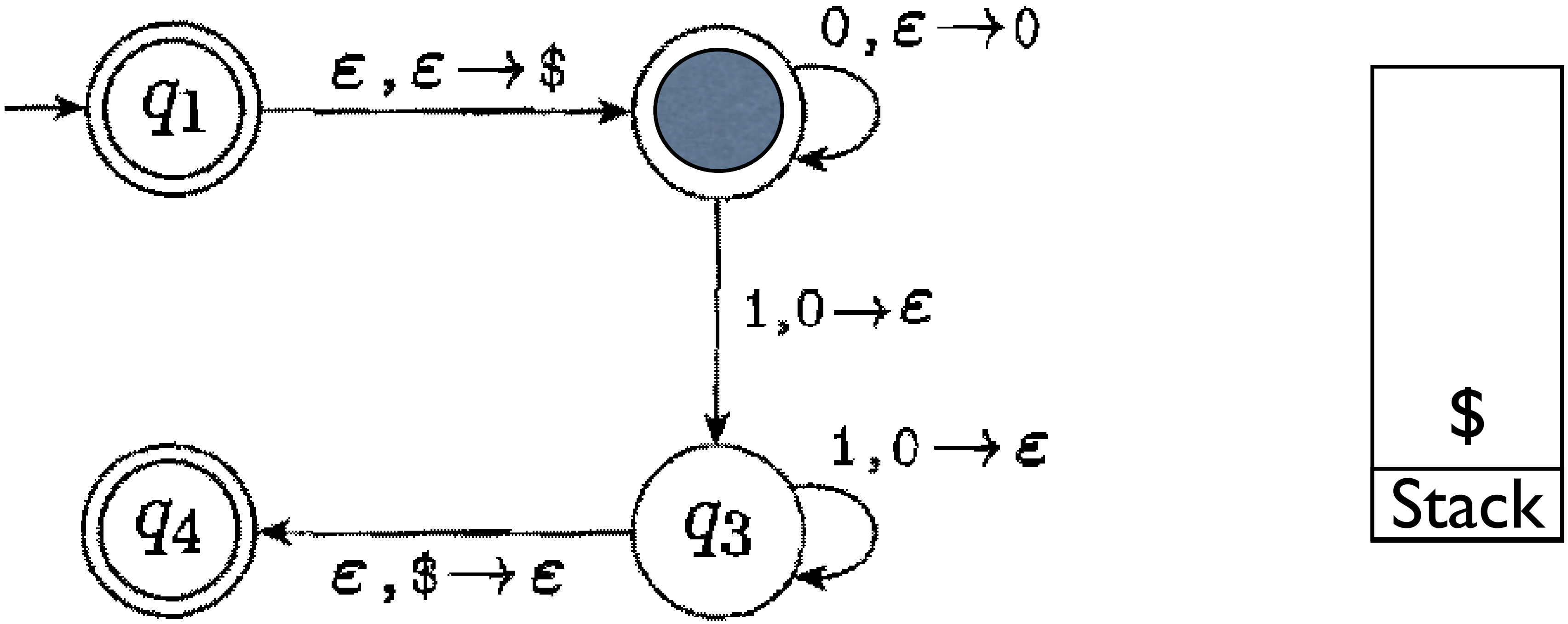
Example Execution: 0011

Begin in initial state
with empty stack

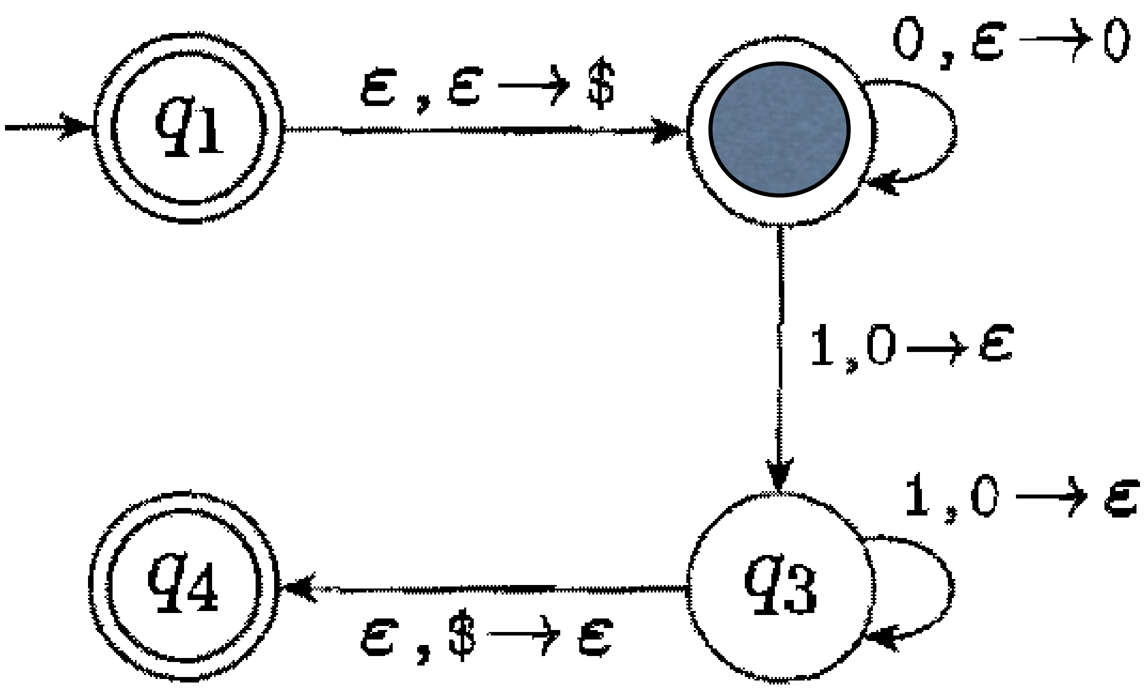


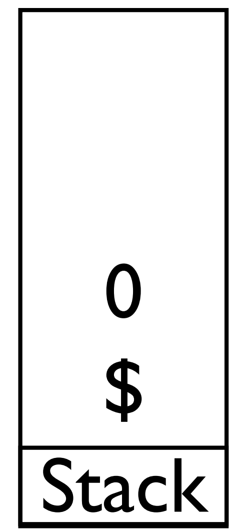


Example Execution: 0011

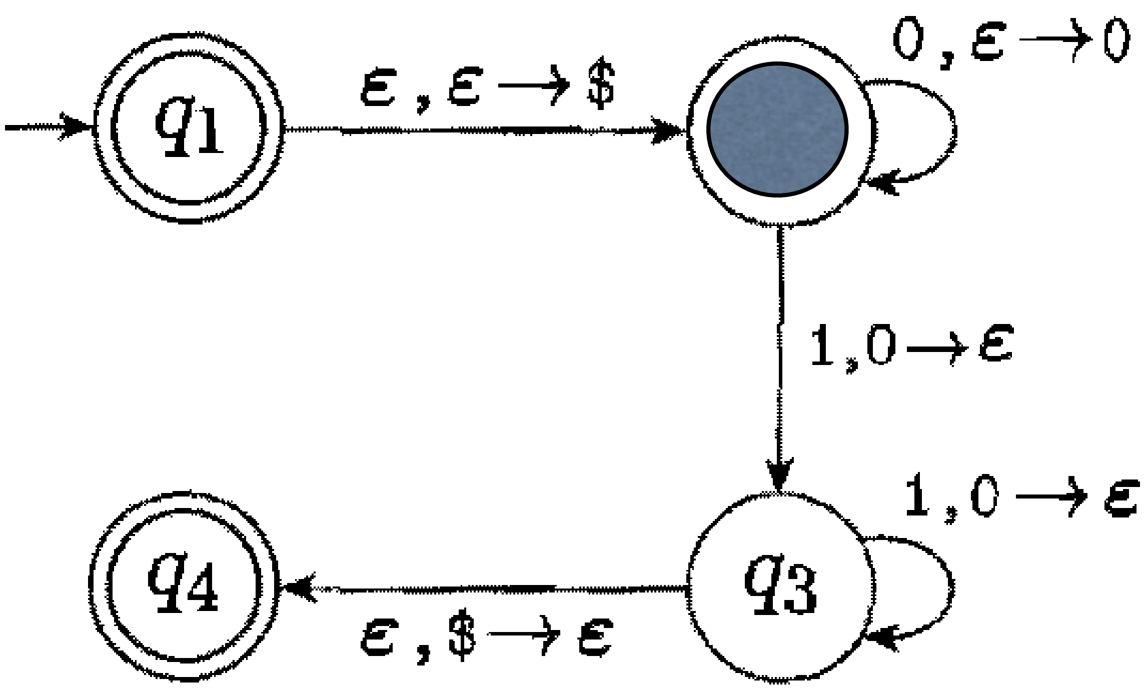


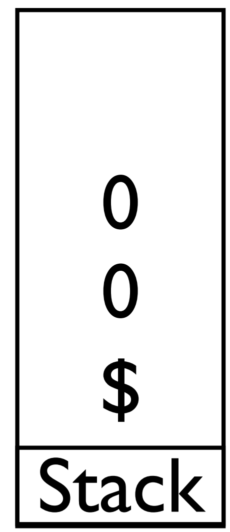
Example Execution: 0011



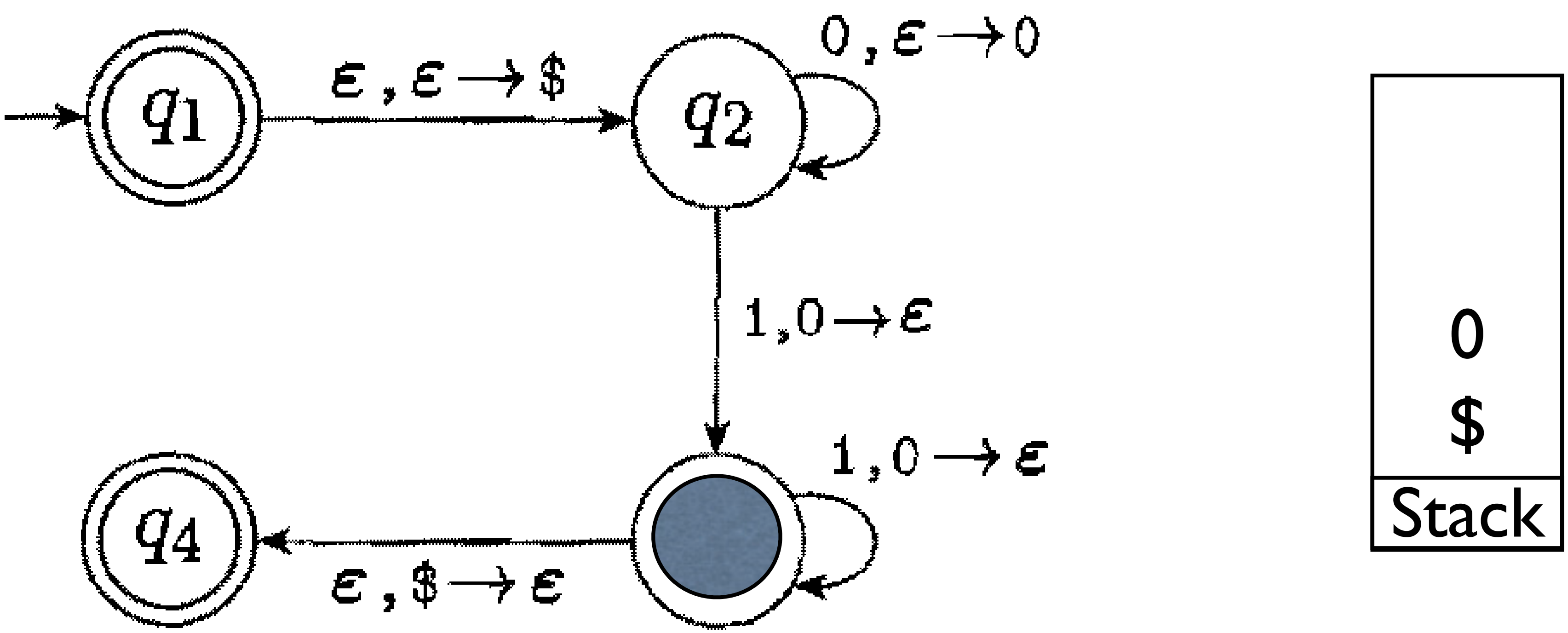


Example Execution: 0011

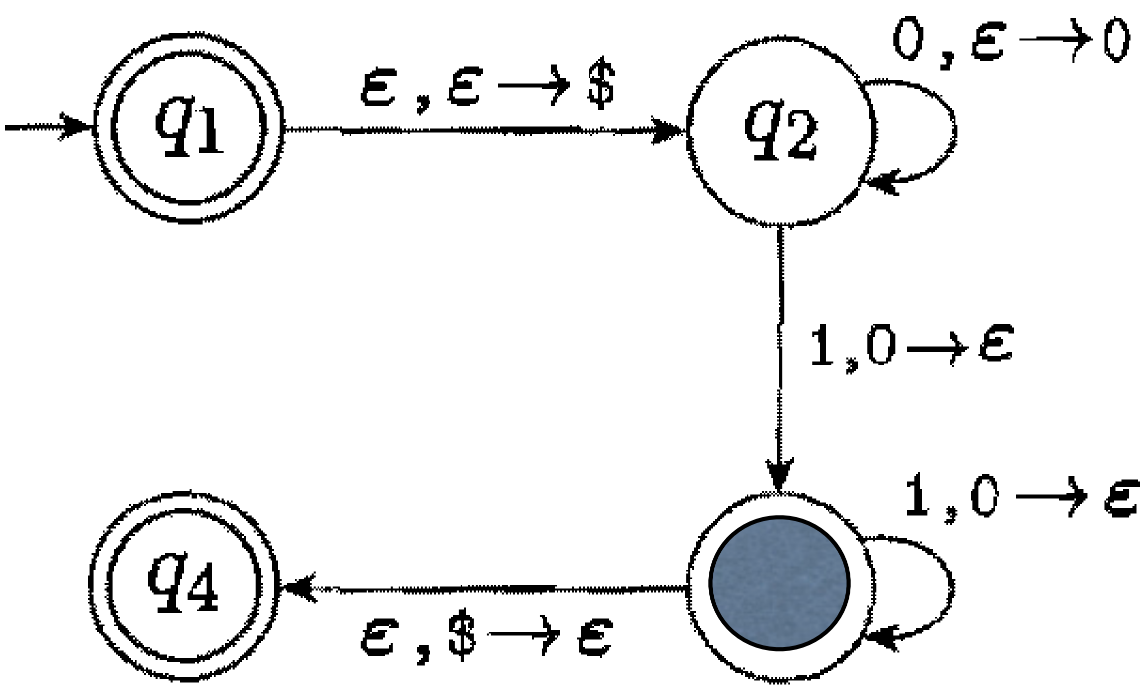


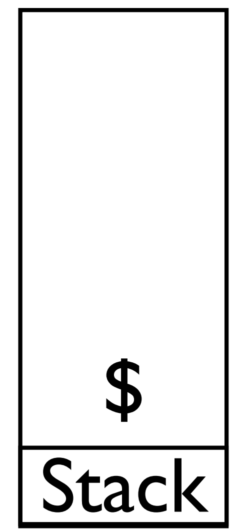


Example Execution: 0011

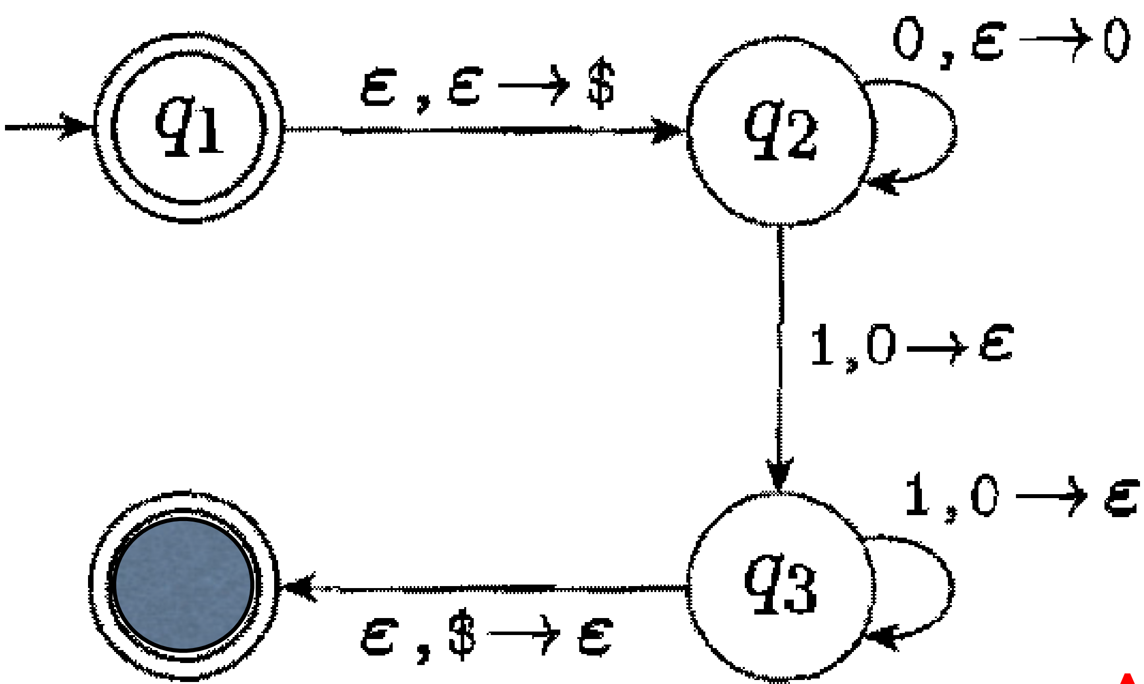


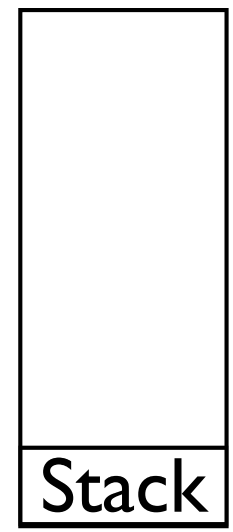
Example Execution: 0011





Example Execution: 0011





Accept!

PDA's can keep count!

- This PDA can recognize $\{0^n 1^n \mid n \geq 0\}$ by
  - First, pushing a symbol on the stack for each 0
  - Then, popping a symbol off the stack for each 1
  - Accepting iff the stack is empty when the end of input is reached (and not before)
- The size of the stack is unbounded.
  - That is, no matter how big the stack grows, it is always possible to push another symbol on it.
  - So PDA's can use the stack to count arbitrarily high

Pushdown Automata (PDA)

- A pushdown automaton M is defined as a 7-tuple: $M = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$, where:
  - Q is a set of states, $q_0 \in Q$ is the start state
  - Σ is the input alphabet,
  - Γ is the stack alphabet, $Z_0 \in \Gamma$ is the initial stack symbol
  - $\delta : (Q \times \Sigma_\varepsilon \times \Gamma_\varepsilon) \rightarrow \mathcal{P}\{Q \times \Gamma^*\}$ is the transition function
  - $F \subseteq Q$ is a set of final states, and
  - $X_\varepsilon = X \cup \{\varepsilon\}$, the set X augmented with ε