PDAs, CFLs, and Parsing, Concluded

November 9, 2010

PDAs

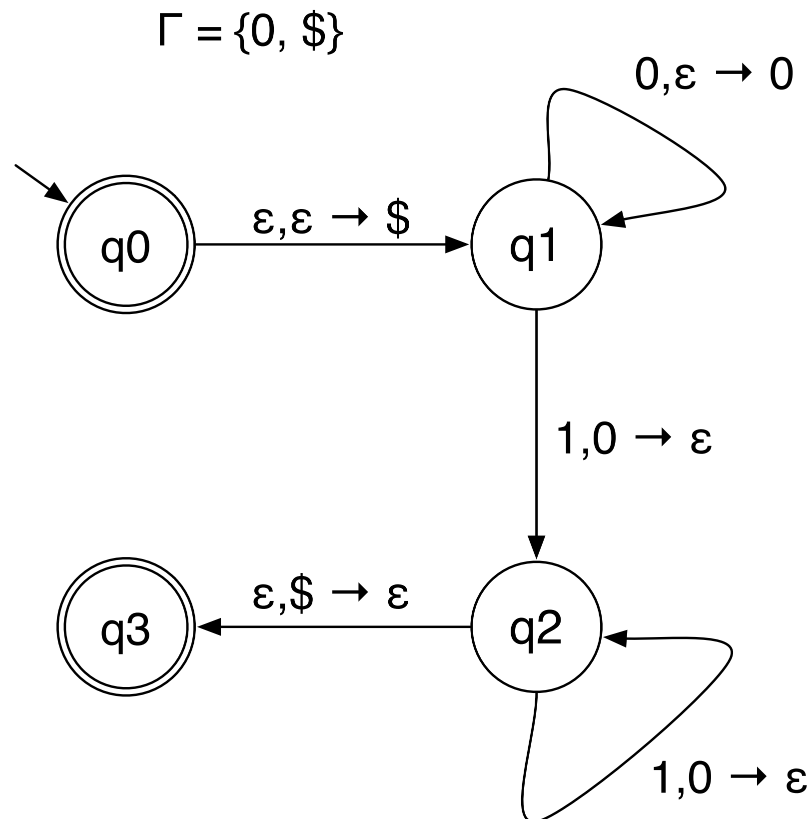
- Q: If regular languages are recognized by finite-state machines, what abstract machines recognize the context-free languages?
- A: Pushdown Automata (PDAs)
  - Finite state machine + stack
  - Transitions
    - Depend on the state and input symbol and top of stack!
    - Changes state and removes/replaces/ top of stack.
  - Accepting states as before (or accept on empty stack)
  - In general, can be **nondeterministic**.

Official Definition

- A PDA is a tuple $(Q, \Sigma, \Gamma, q_0, F, \delta)$
  - Q is a finite set of states
  - Σ is a finite alphabet
  - Γ is a finite "stack" alphabet
  - $q_0 \in Q$ is a start state
  - $F \subseteq Q$ is a set of accepting states
  - $\delta : Q \times (\Sigma \cup \{\varepsilon\}) \times (\Gamma \cup \{\varepsilon\}) \rightarrow \mathcal{P}(Q \times (\Gamma \cup \{\varepsilon\}))$

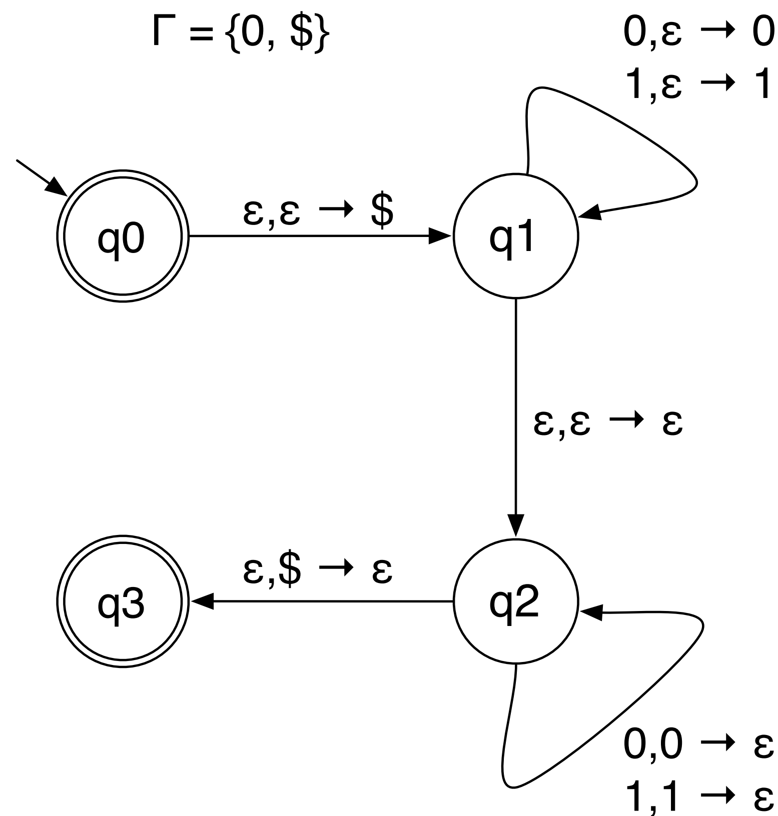
Example

- Using a state machine and a stack, how can we recognize $\{ 0^n1^n \mid n \geq 0 \}$?



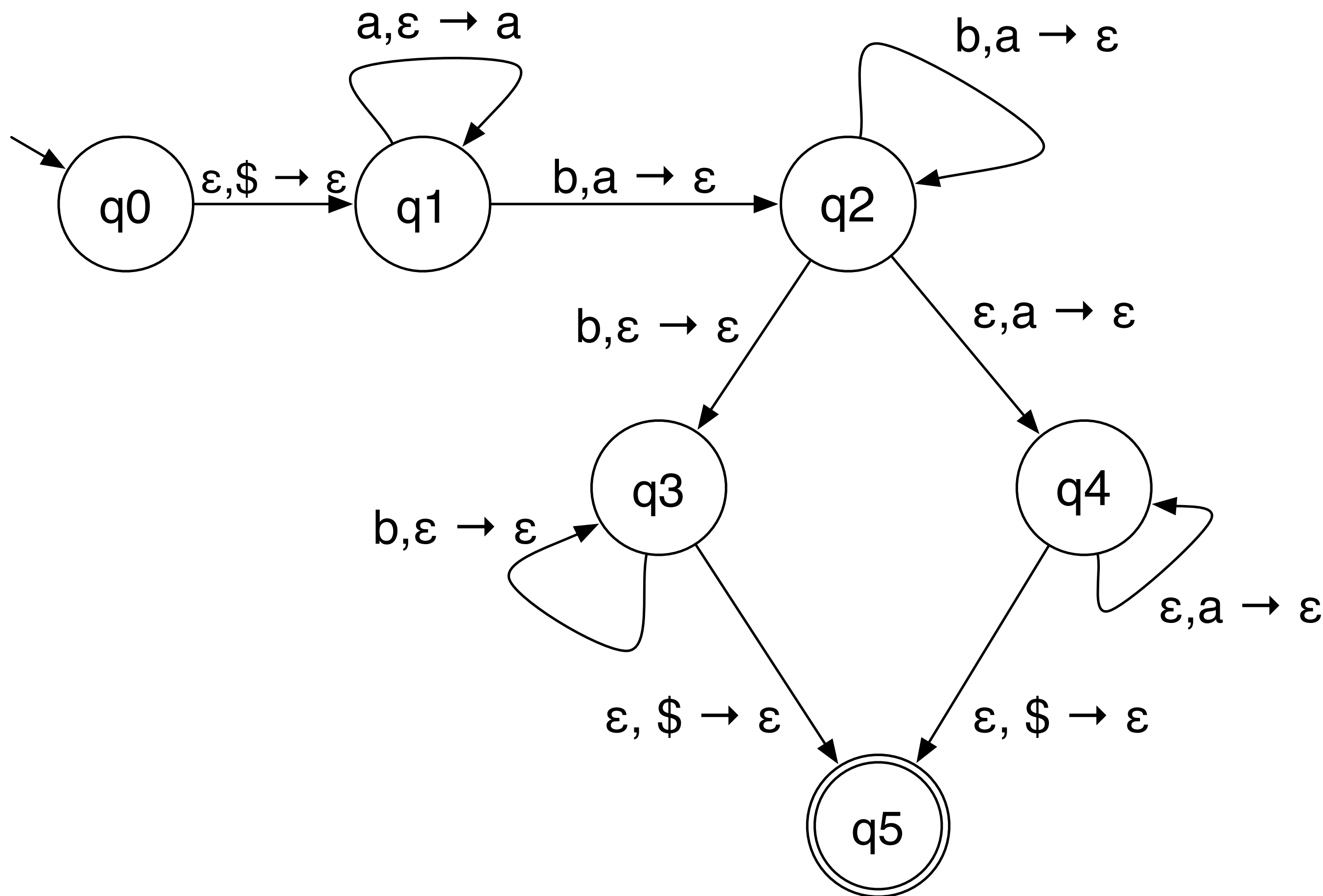
Example

- Using a state machine and a stack, how can we recognize $\{ ww^R \mid w \in \Sigma^* \}$?



Bigger Example

$\{ a^i b^j \mid i \neq j \wedge i, j > 0 \}$



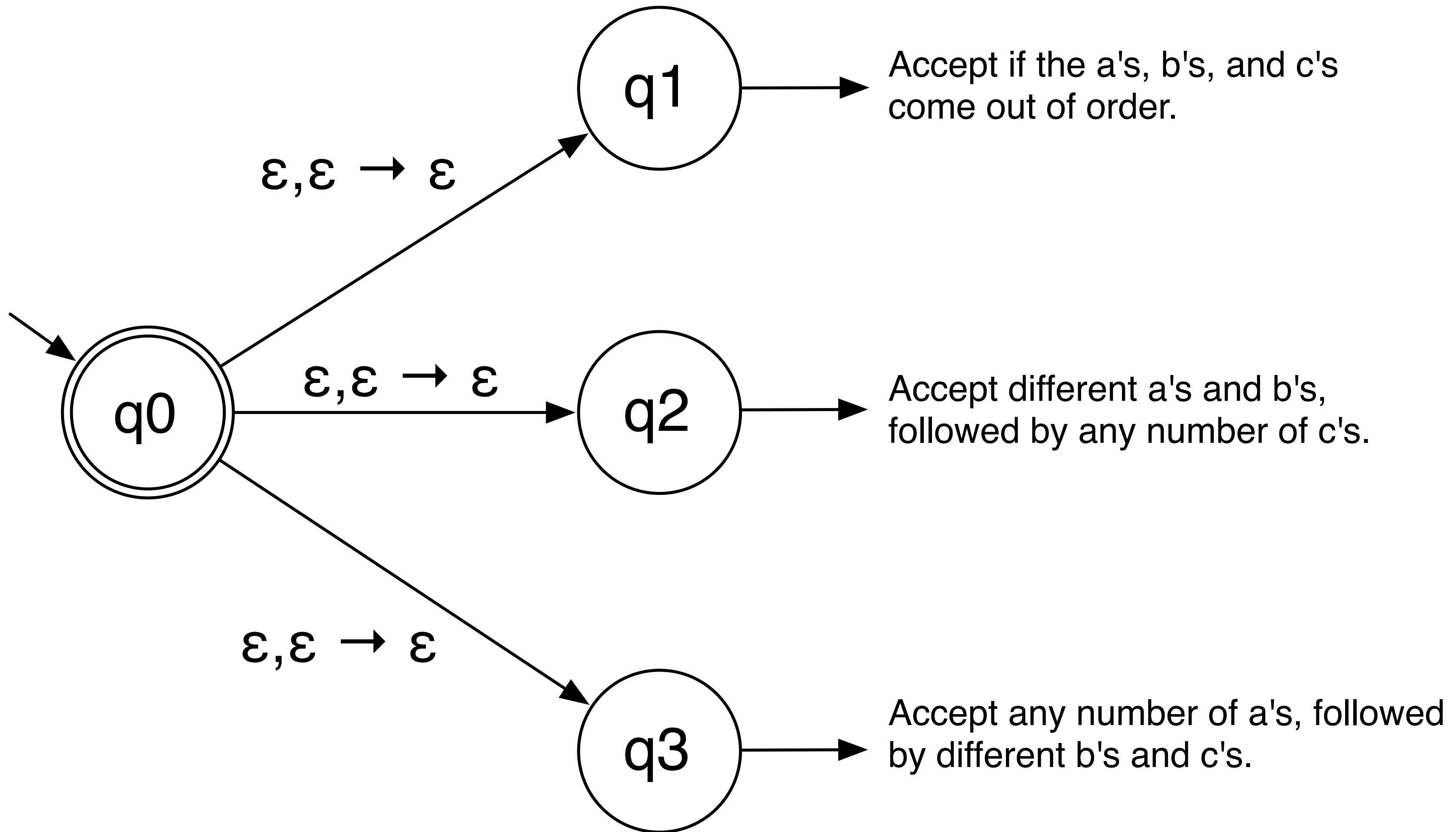
PDAs vs CFGs

- PDAs recognize **all** context-free languages.
  - Given a grammar, construct a PDA to do predict-match parsing
  - Use nondeterminism to **always** guess the correct prediction! (No backtracking officially required)
- PDAs recognize only context-free languages.
  - Turn the PDA into a grammar that simulates it.
  - See the book for details.
  - Basically, for each pair of states (p,q), the nonterminal A_{pq} produces strings that get you from p to q starting and ending with an empty stack.

The Family of CFLs

- Closed under star, union, concatenation.
- Not closed under intersection, complement.
  - Key issue: nondeterministic PDAs are strictly more powerful than deterministic PDAs!
  - No analogy to the subset construction!

$\Sigma^* \setminus \{ a^n b^n c^n \mid n \geq 0 \}$



Parsing Methods

- Recursive Descent
  - Form of code follows the grammar.
  - Efficient and correct for LL(k) grammars.
- Another practical method: LR grammars
  - Parser code is usually computer-generated from the grammar!
- But neither of these handle ALL context-free grammars...

CYK (a.k.a. CKY)

- Parses any grammar in $O(n^3)$ time, where n is the length of the input.
  - Constants depend on the size of the grammar.
  - Requires a grammar in Chomsky Normal Form
- For every substring, what nonterminals produce it?
- Use dynamic programming for efficiency.

Dynamic Programming

- Recursive expressions, such as

$$f(n) = f(n-1) + f(n-3)$$
$$f(n) = 1 \text{ if } n < 3$$

are very clear, but inefficient if taken literally.

- Two roughly-equivalent solutions:
  - **Memoization**: keep track of what f's have been computed
  - **Dynamic Programming**: Compute all f's in a good order

CYK Algorithm

- Let $x = x_1 x_2 \cdots x_n$ be the string to be parsed.
- Define $a(i, j) := \{ B \mid B \Rightarrow^* x_i x_{i+1} \cdots x_j \}$
- Then $x \in L(G)$ iff $S \in a(1,n)$

- How can we compute $a(1,n)$ efficiently?

Recursive Definition

$$a(i,k) = \{\ C \mid C \rightarrow AB,\ A \in a(i,j),\ B \in a(j+1,k)\ \}$$

$$a(i,i) = \{\ C \mid C \rightarrow x_i\ \}$$

Dynamic Programming

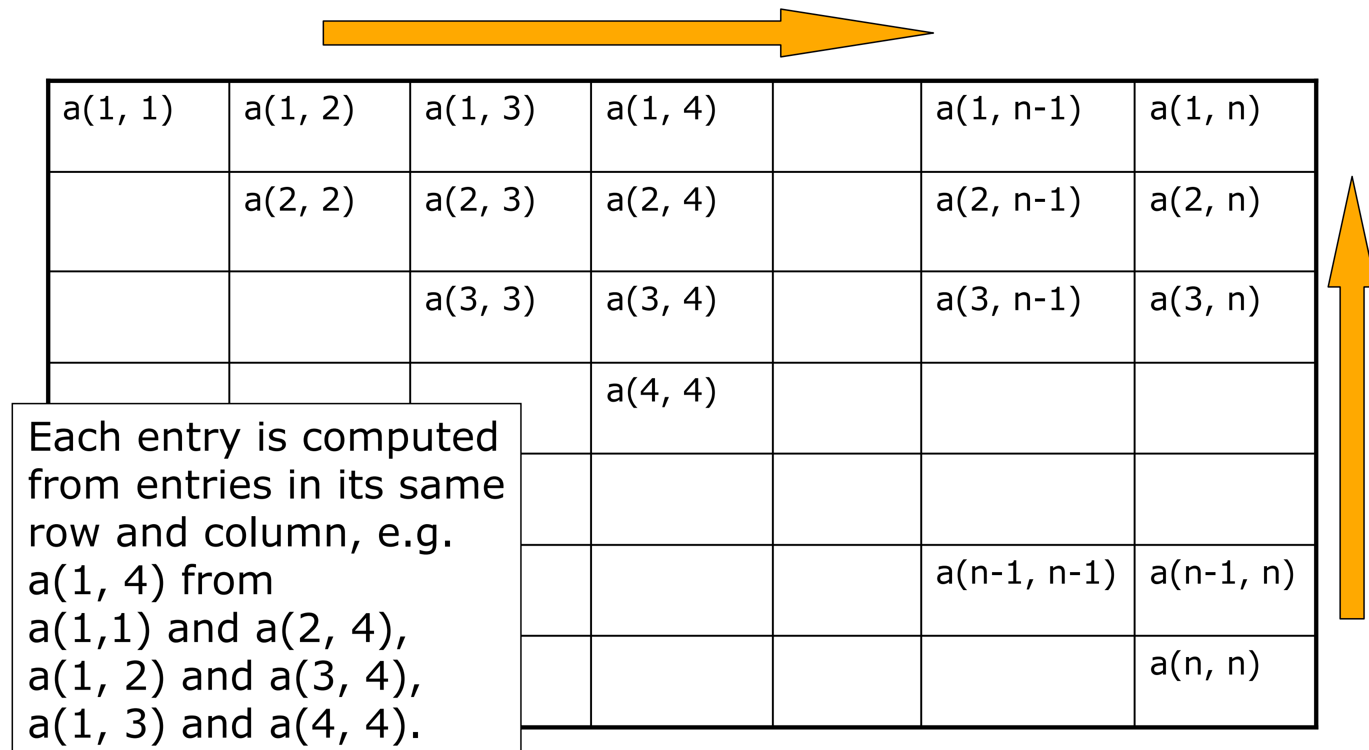
- First, compute a(i,i) for each i. — Substrings of length 1
- Then, compute a(i,i+1) for each i. — Substrings of length 2
  - All ways of breaking it up into two parts of length < 2
- Then, compute a(i,i+2) for each i. — Substrings of length 3
  - All ways of breaking it up into two parts of length < 3

⋮

- Then, compute a(i,i+n-1) for each i. — Substrings of length n

Matrix Representation

CYK “Wavefront” Matrix



Example

- Parse (()())

$S \rightarrow LT$

$T \rightarrow SR$

$S \rightarrow LR$

$S \rightarrow SS$

$L \rightarrow ($

$R \rightarrow)$

Digression: LR Parsing

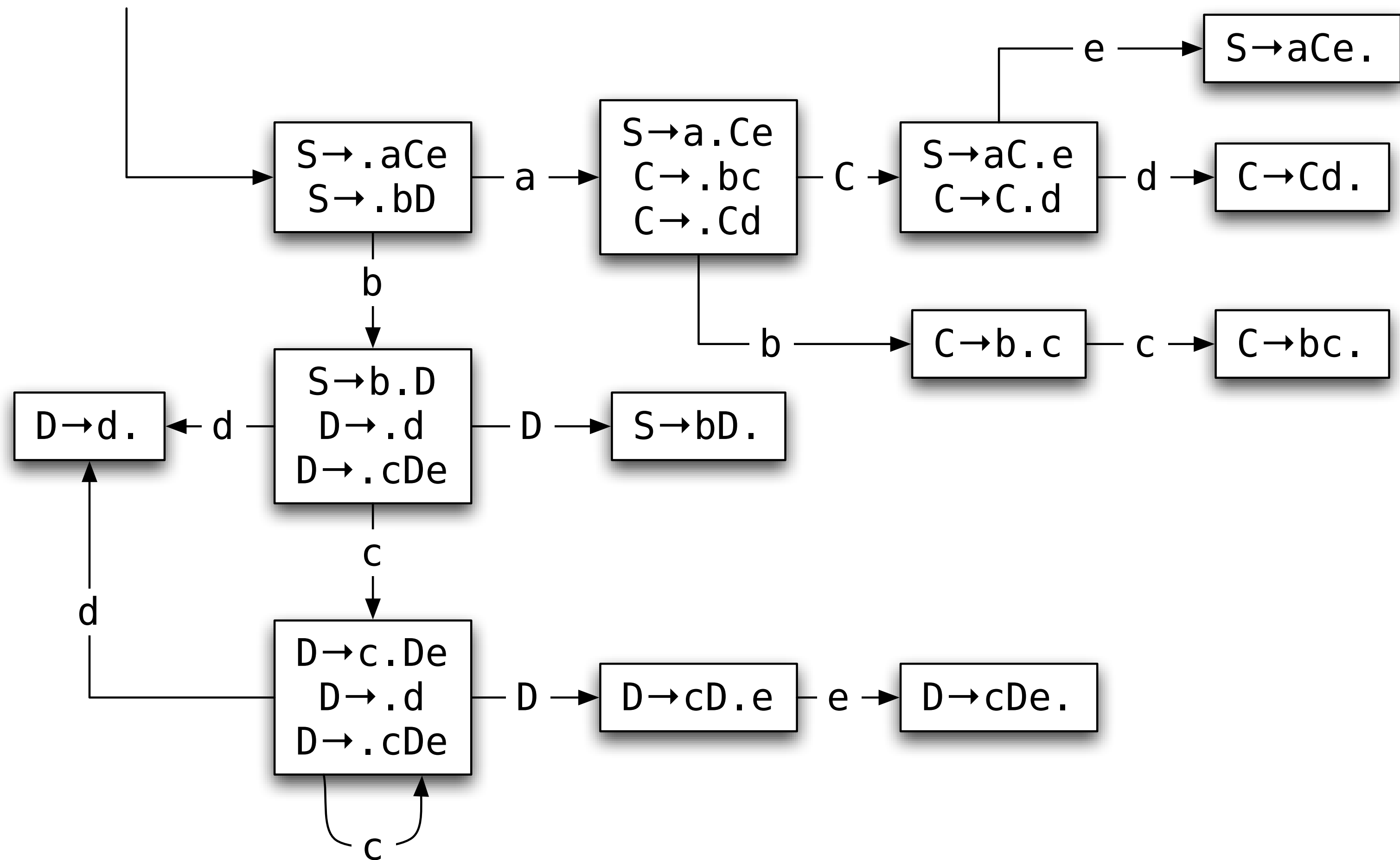
- Like Predict/Match (LL) parsing, we have a stack. But, our two choices are:
  - Shift: Put the next item of the input on the stack
  - Reduce: Run a production rule backwards!
- Success if we run out of input when stack holds just the start symbol.
- Effectively builds parse tree from the leaves up

Example

$$S \rightarrow aCe \mid bD$$
$$C \rightarrow bc \mid Cd$$
$$D \rightarrow d \mid cDe$$

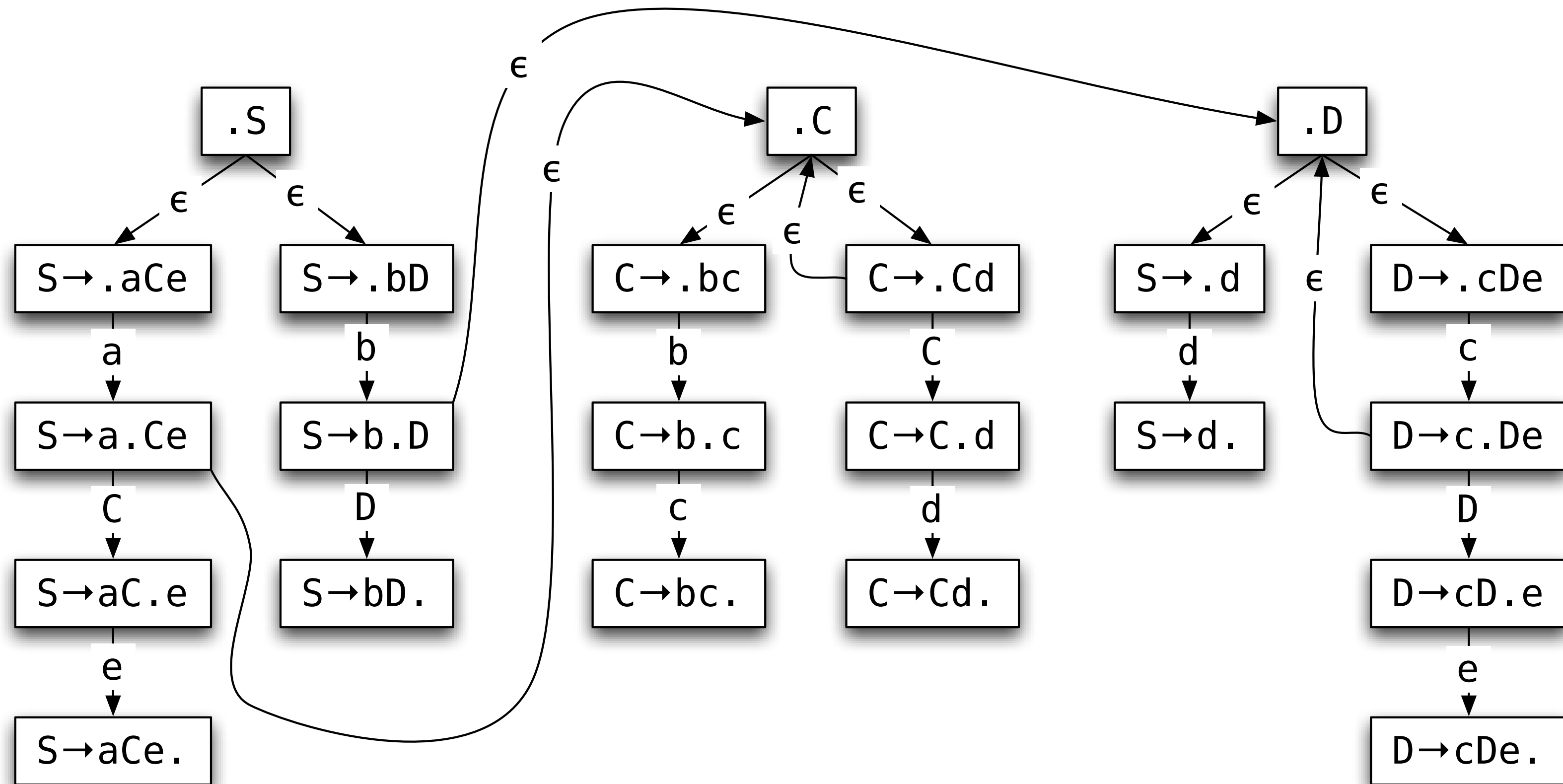
- Parse abcde
- Parse bcde

DFAs to the Rescue

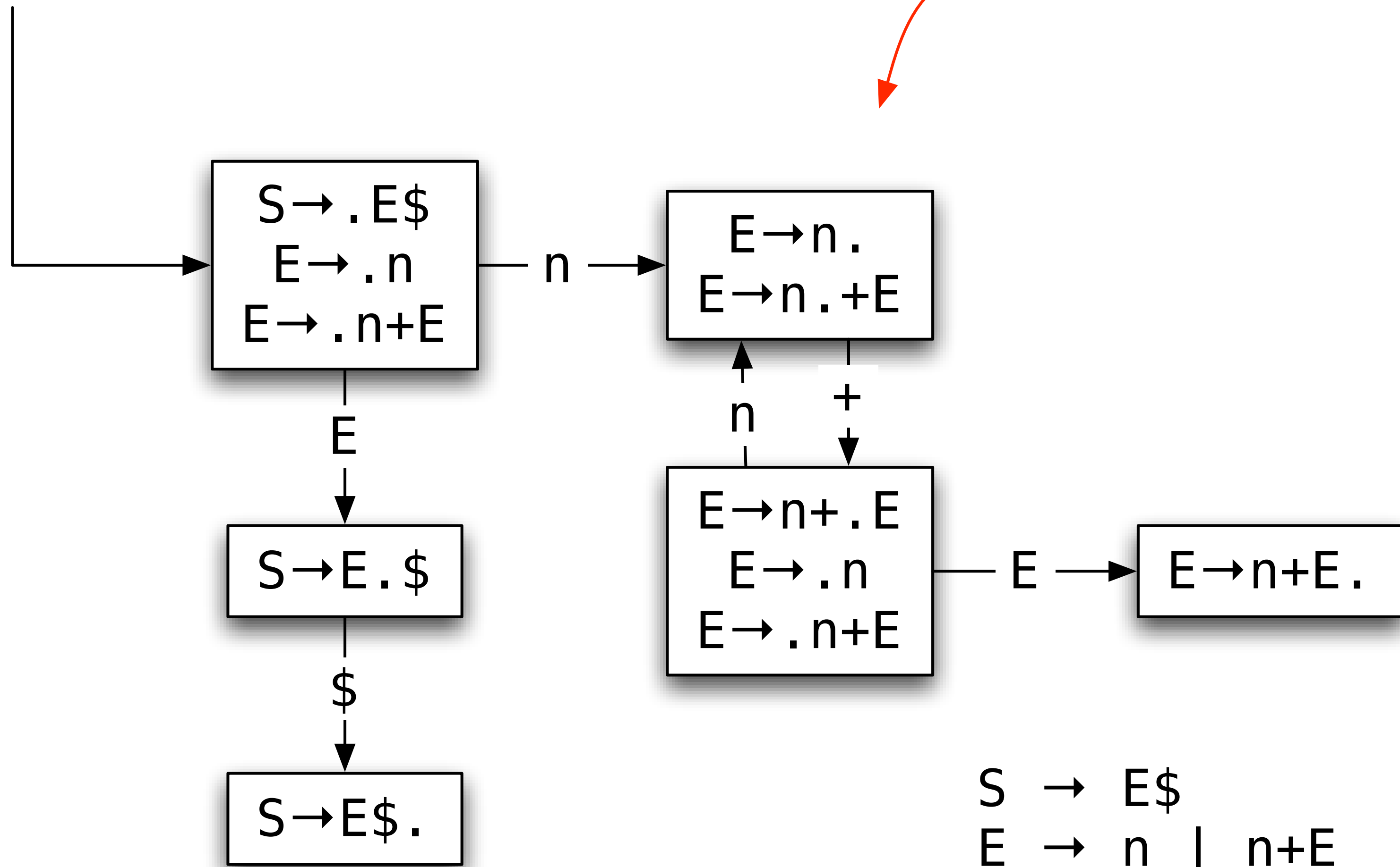


LR(0)

S → aCe | bD
C → bc | Cd
D → d | cDe

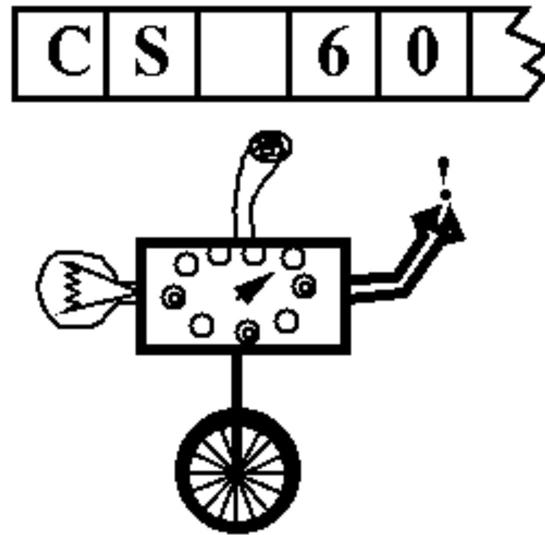


S → aCe | bD
C → bc | Cd
D → d | cDe

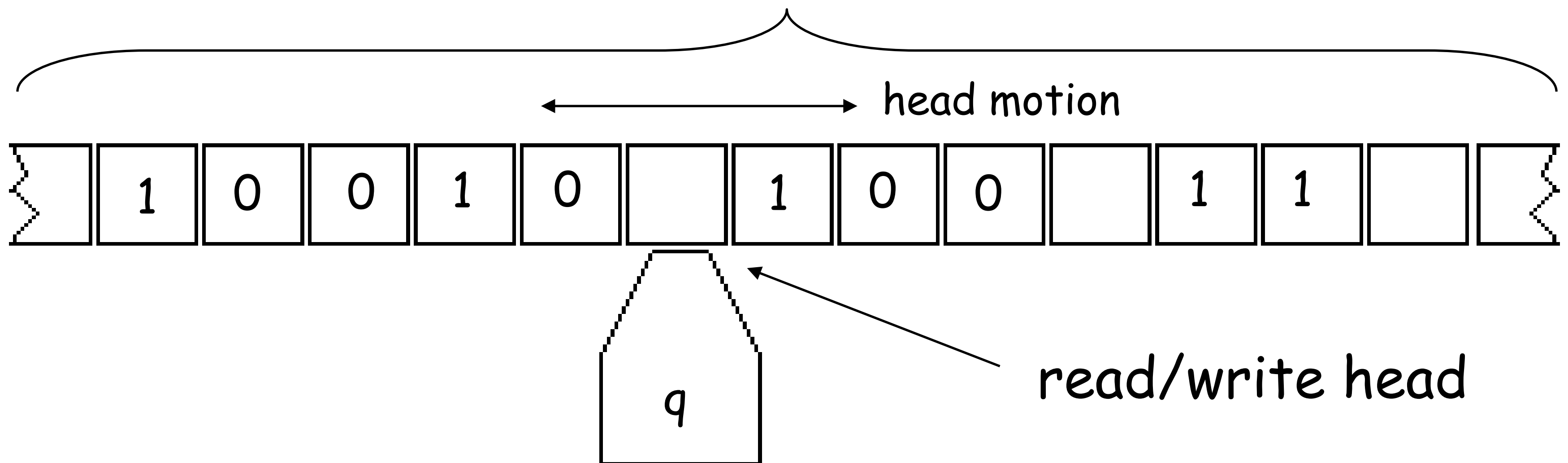


not LR(0)
S→.E$
E→.n
E→.n+E
n
E→n.
E→n.+E
E
n
+
S→E.$
E→n+.E
E→.n
E→.n+E
E
E→n+E.
$
S→E$.
S → E$
E → n | n+E

Coming Up...Turing Machines



(Artist's Conception)



control, in control state q

Relation to PDAs

- Like a FSA but
  - can write as well as read its tape
  - can move in both directions
- Like a PDA with 2 stacks (or a queue)
- More powerful than a locomotive, able to ...