Parsing and PDAs

April 6, 2011

CS 81: Computability and Logic

Review

- ✓ Grammars
- ✓ Context-Free Grammars
- ✓ Ambiguous and Unambiguous Grammars
- ✓ A Pumping Lemma for CFLs ($uvxyz$)
  - $\{ a^n b^n c^n \mid n \geqslant 0 \}$ is not context free.
  - $\{ ww \mid w \in \{a, b\}^* \}$ is not context free

Chomsky Hierarchy

- ✓ Type 0: Unrestricted Grammars
- ✓ Type 1: Context-Sensitive Grammars
- ✓ Type 2: Context-Free Grammars
- ✓ Type 3: Regular Grammars

By Popular Demand...

What is the language of the following *context-sensitive* (*monotone*) grammar?

$$
\begin{array}{lcl}
\mathrm{S} & \to & \mathrm{abc} \\
\mathrm{S} & \to & \mathrm{aSQ} \\
\mathrm{bQc} & \to & \mathrm{bbcc} \\
\mathrm{cQ} & \to & \mathrm{Qc}
\end{array}
$$

Strings with Composite (Non-Prime) Lengths

$$
\begin{aligned}
S &\to HaC \\
C &\to aC \\
C &\to aT \\
T &\to ZU \\
AZ &\to ZA \\
aZ &\to ZAa \\
HZ &\to HY \\
YA &\to AY \\
Ya &\to aY \\
YU &\to T \\
T &\to X \\
aX &\to Xaa \\
AX &\to Xa \\
HX &\to \varepsilon
\end{aligned}
$$

The Parsing Problem

Given a grammar and a string,

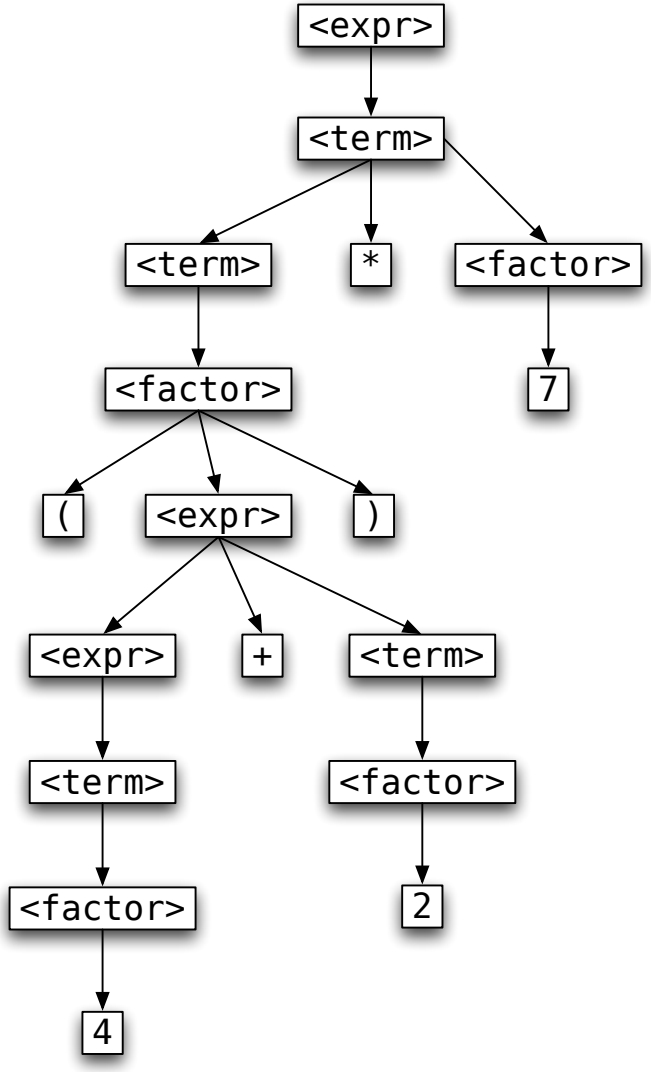
1. Is the string in the language?

The Parsing Problem

Given a grammar and a string,

1. Is the string in the language?
2. Why? (Parse tree or other evidence)

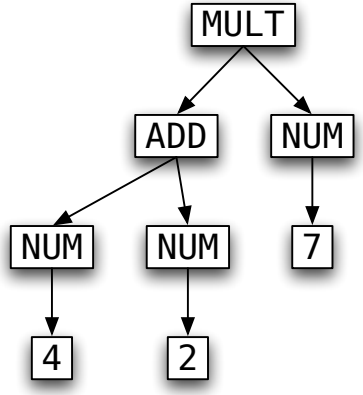
Evidence




```
<expr> ::= <term>
         | <expr> + <term>

<term> ::= <factor>
         | <term> * <factor>

<factor> ::= <int>
           | ( <expr> )
```



42

Naive Parsing

How about backtracking search? All ways to derive the string from S.

Naive Parsing

How about backtracking search? All ways to derive the string from S.

- ✓ Inefficient
- ✓ Harder to implement than you might think...

Bogus Backtracking

```
S -> Ac | Bc
A -> a | c | ac
B -> Bb | b
```

```
Consume_S():
    try Consume_A(), then consume c
    if fails, try Consume_B(), then consume c
```

Bogus Backtracking

```
S -> Ac | Bc
A -> a | c | ac
B -> Bb | b
```

```
Consume_S():
    try Consume_A(), then consume c
    if fails, try Consume_B(), then consume c

Consume_A():
    try consume a
    if fails, try consume c
    if fails, try consume a then consume c
```

[What's wrong?]

Bogus Backtracking

```
S -> Ac | Bc
A -> a | c | ac
B -> Bb | b
```

```
Consume_S():
    try Consume_A(), then consume c
    if fails, try Consume_B(), then consume c

Consume_A():
    try consume a
    if fails, try consume c
    if fails, try consume a then consume c
```

[What's wrong?]

```
Consume_B():
    try Consume_B(), then consume b
    if fails, try consume b
```

[What's wrong?]

LL(k) Grammars

```
S -> Aa | Ba
```

If *each* `Consume` function always "knew" which right-hand-side was correct, **we would never need to backtrack or and never get tangled in infinite loops**.

LL(k) Grammars

```
S -> Aa | Ba
```

If *each* `Consume` function always "knew" which right-hand-side was correct, **we would never need to backtrack or and never get tangled in infinite loops**.

Then

- ✓ It would be easy to write correct `Consume` functions
- ✓ Our parser would run in linear time.

LL(k) Grammars

```
S -> Aa | Ba
```

If each `Consume` function always "knew" which right-hand-side was correct, **we would never need to backtrack or and never get tangled in infinite loops**.

Then

- ✓ It would be easy to write correct `Consume` functions
- ✓ Our parser would run in linear time.

We say that a grammar is LL(k) if, by "peeking ahead" no more than k tokens, we can guarantee a decision that is

1. correct
2. unique

When will Top-Down Parsing Work?

Are these grammars $LL(k)$ for some k?

```
S ::= E $
E ::= n
    | plus E E
    | times E E
```

```
S ::= A
    | B
A ::= a
    | x A
B ::= b
    | y B
```

```
S ::= A
    | B
A ::= a
    | x A
B ::= b
    | x B
```

```
S ::= E $
E ::= n
    | n + E
```

```
S ::= E $
E ::= n
    | E + n
```

```
S ::= E $
E ::= E + E
    | E * E
    | n
```

Machines

Question: If FSMs recognize regular languages, what machines recognize CFLs?

Machines

Question: If FSMs recognize regular languages, what machines recognize CFLs?

Answer: Pushdown Automata (PDAs)

- ✓ Finite state machine + stack
- ✓ Transitions
  - Depend on the state and/or input symbol
  - Change state + add/remove/replace top-of-stack
- ✓ In general, can be nondeterministic.

Example

Using a state machine and a stack, how could we recognize

$$\{ 0^n 1^n \mid n \geqslant 0 \}$$

Official Definition

A PDA consists of

- ✓ A finite set Q of states
- ✓ A finite alphabet Σ
- ✓ A finite "stack alphabet" Γ
- ✓ A start state $q_0 \in Q$
- ✓ Accepting states $F \subseteq Q$
- ✓ Transitions

$$\delta : Q \times (\Sigma \cup \{\varepsilon\}) \times (\Gamma \cup \{\varepsilon\}) \to \mathcal{P}(Q \times (\Gamma \cup \{\varepsilon\}))$$

PDA Example

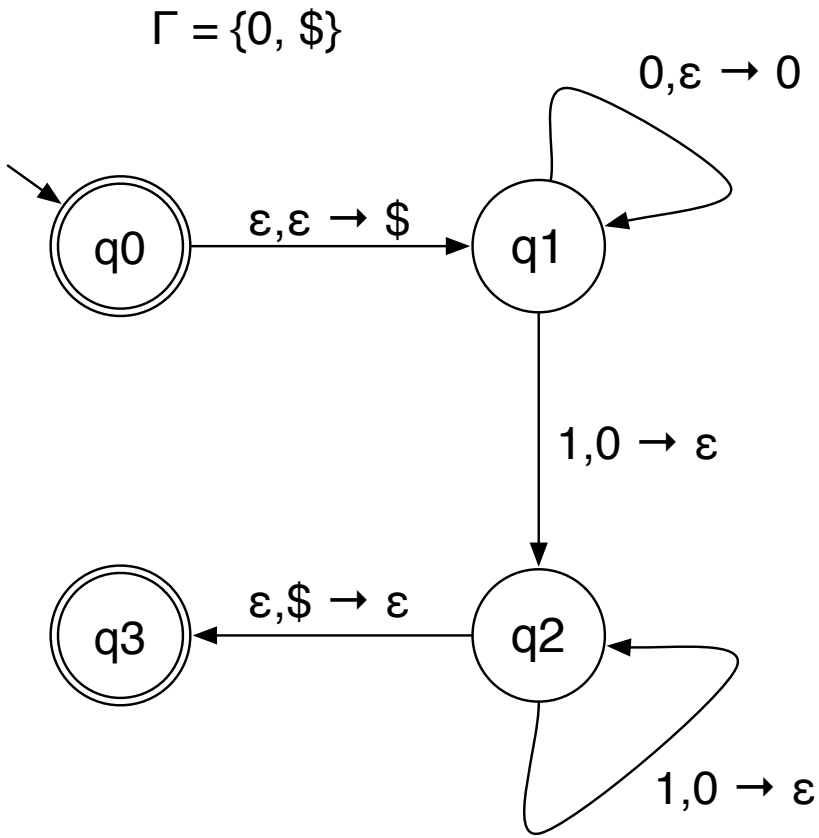
Using a state machine and a stack, how could we recognize

$$\{ 0^n 1^n \mid n \geqslant 0 \}$$

PDA Example

Using a state machine and a stack, how could we recognize

$$\{ 0^n 1^n \mid n \geqslant 0 \}$$



PDA Example

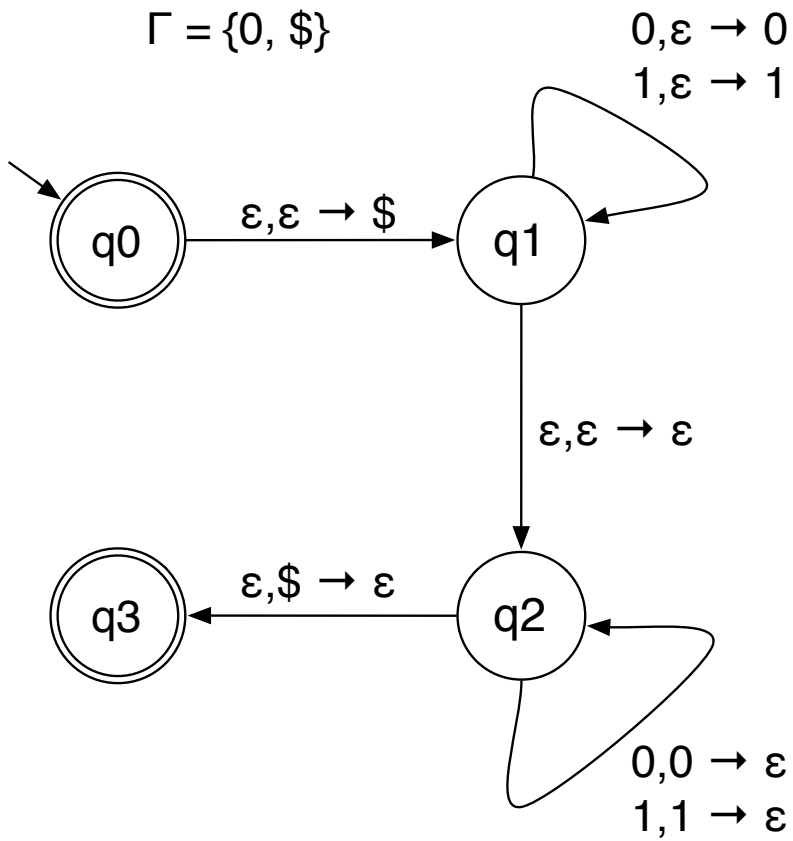
Using a state machine and a stack, how could we recognize

$$\{ ww^R \mid w \in \{0, 1\}^* \}$$

PDA Example

Using a state machine and a stack, how could we recognize

$$\{ ww^R \mid w \in \{0, 1\}^* \}$$



PDAs vs CFGs: Summary

1. PDAs can recognize any CFL
   - Given a grammar, construct a PDA for top-down parsing
   - Use nondeterminism to always guess the correct prediction (No backtracking required)

2. PDAs recognize only CFLs
   - Turn a PDA into a grammar that simulates it
   - See the book: for each pair of states $\langle p, q \rangle$, define a nonterminal A_{pq} (strings that get you from p to q, starting and ending with empty stack).

Closure Properties

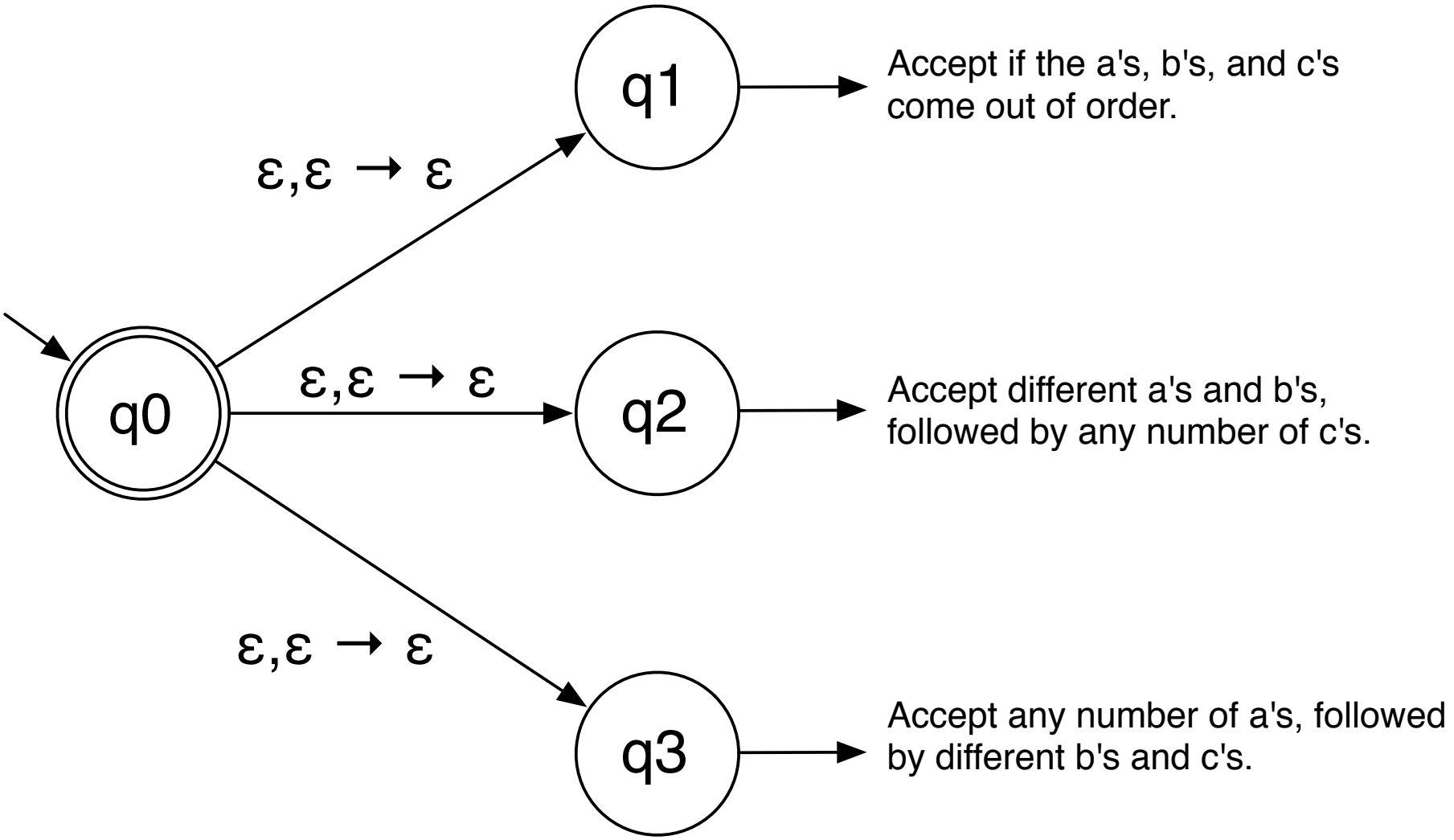
Context-Free Languages are:

- ✓ 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)

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



Practical Parsing

Recursive Descent

- ✓ Form of code follows the grammar.
- ✓ Efficient and correct for LL grammars.

Another very practical method: Shift-Reduce Parsing

- ✓ Efficient and correct for LR grammars
- ✓ Parser code is usually computer-generated!

But neither of these handle ALL CFGs…

CYK (CKY) Algorithm

- ✓ Works for any CFG.
- ✓ Parses inputs in $\mathbf{O(n^3)}$ ($\mathbf{n}$ = length of input)
- ✓ Requires a grammar in "Chomsky Normal Form"
- ✓ Key ideas:
  - For every substring, what nonterminals produce it?
  - Dynamic programming for efficiency