Parsing and PDAs

November 19, 2012

CS 81: Computability and Logic

Encoding Precedence

```
E ::= d
    | E + E
    | E - E
    | E * E
    | E / E
    | ( E )
```

```
E ::= T
    | E + T
    | E - T

T ::= F
    | T * F
    | T / F

F ::= d
    | ( E )
```

Show how to parse `3 + 4 * 5`

Digression

There exist *inherently ambiguous* languages that have no unambiguous grammar.

$$\{a^n b^n c^m d^m \mid n, m > 0\} \quad \cup \quad \{a^n b^m c^m d^n \mid n, m > 0\}$$

Obviously, we tend to avoid these when defining programming languages...

But does every language have at least one context-free grammar?

Another Pumping Lemma

If L is context-free, then

there exists a number p such that

For every $s \in L$ with $|s| \geqslant p$

we can decompose s into $uvxyz$ where

1. $vy \neq \varepsilon$
2. $|vxy| \leqslant p$
3. $uv^ixy^iz \in L$ for every $i \geqslant 0$.

$$L := \{ a^n b^n c^n \mid n \geqslant 0 \}$$

Suppose L were context-free

- ✓ Let p be the pumping length
- ✓ Consider, for example, $s := a^p b^p c^p$. (Note that $|s| \geqslant p$.)
- ✓ Consider *all* possible decompositions

$$s = uvxyz \quad \text{with} \quad vy \neq \varepsilon \ \wedge \ |vxy| \leqslant p.$$

- ✓ None of them stay in L when we pump. (Why?)
- ✓ Contradiction.

So L is not context-free. QED.

$L := \{ ww \mid w \in \{0,1\}^* \}$

Suppose L were context-free

- ✓ Let p be the pumping length
- ✓ Consider, for example, $s := 0^p 1 0^p 1$. (Note that $|s| \geqslant p$.)
- ✓ Consider, for example, $s := 0^p 1^p 0^p 1^p$. (Note that $|s| \geqslant p$.)
- ✓ Consider *all* possible decompositions

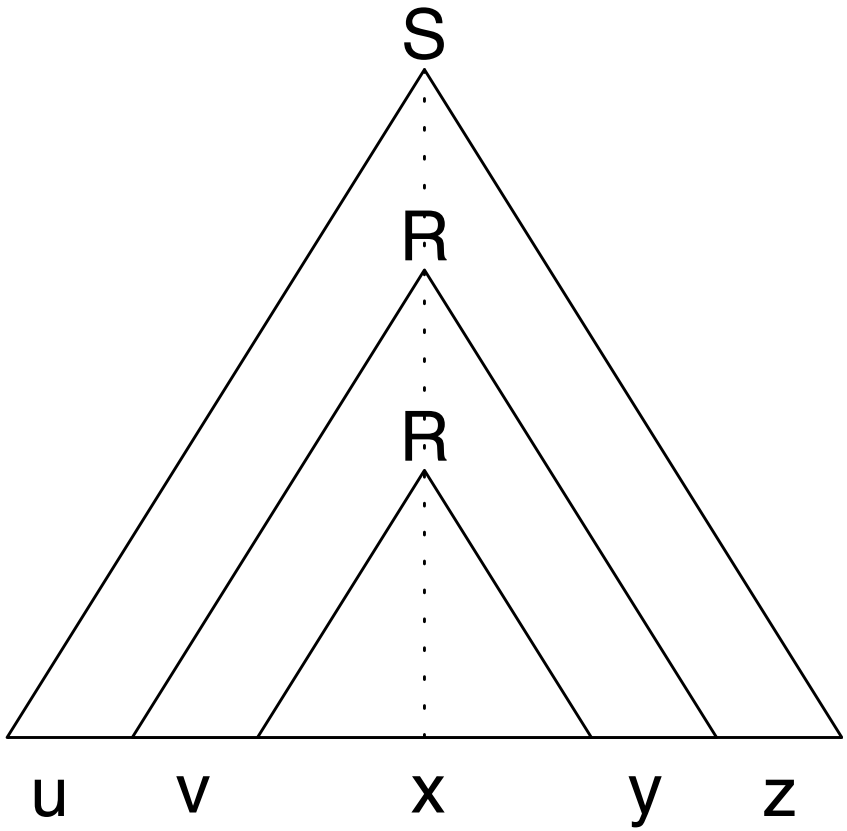
$$s = uvxyz \quad \text{with} \quad vy \neq \varepsilon \ \wedge \ |vxy| \leqslant p.$$

- ✓ Oops…this string *can* be pumped! Tells us nothing.
- ✓ None of them stay in L when we pump. (Why?)
- ✓ Contradiction.

So L is not context-free. QED.

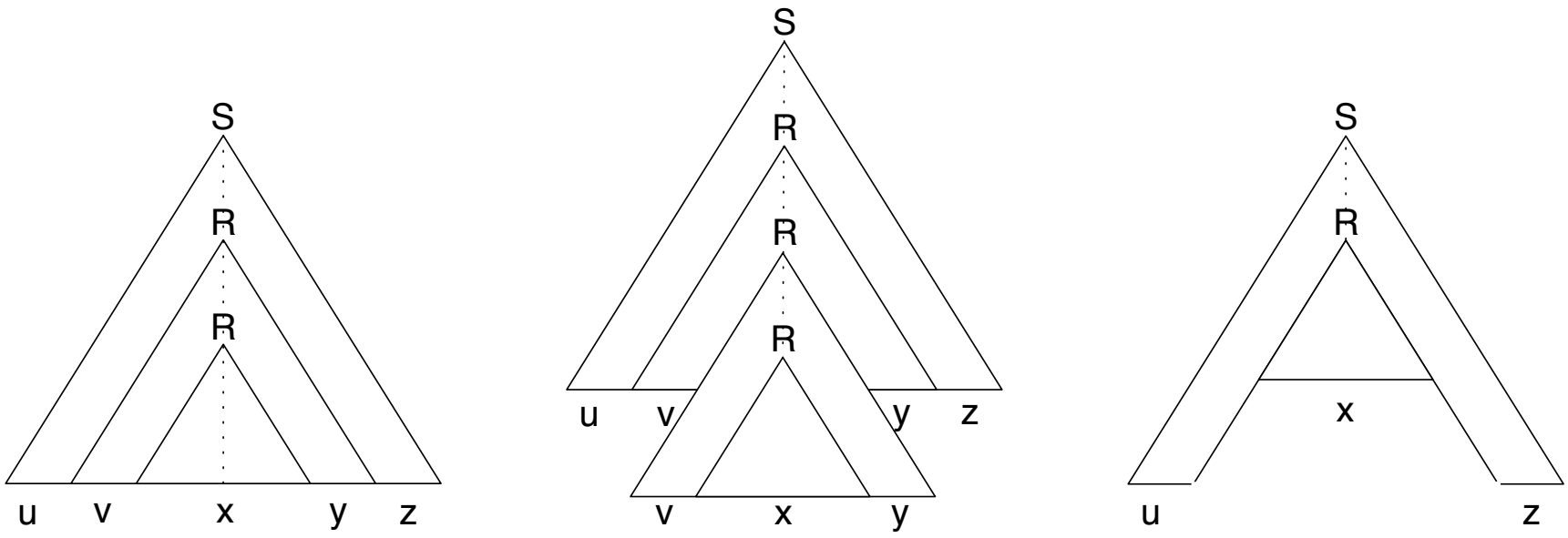
Intutition

If our parse tree is tall enough, there must be a path with a repeating nonterminal.



Consequence?

Pumping



- ✓ How can we be sure that v and y aren't empty?
- ✓ How can we ensure vxy is short enough?

Regular Grammars

A *grammar* is *regular* if its rules are all of the forms:

$$
\begin{aligned}
X &\rightarrow a \\
X &\rightarrow aY \\
X &\rightarrow \varepsilon
\end{aligned}
$$

Example Regular Grammar

$$
\begin{aligned}
S &\rightarrow 1B \\
B &\rightarrow 1B \\
B &\rightarrow 0C \\
C &\rightarrow 0S \\
S &\rightarrow 0S \\
C &\rightarrow 1B \\
C &\rightarrow
\end{aligned}
$$

How could we turn this into a finite state machine?

The Parsing Problem

Given a grammar and a string,

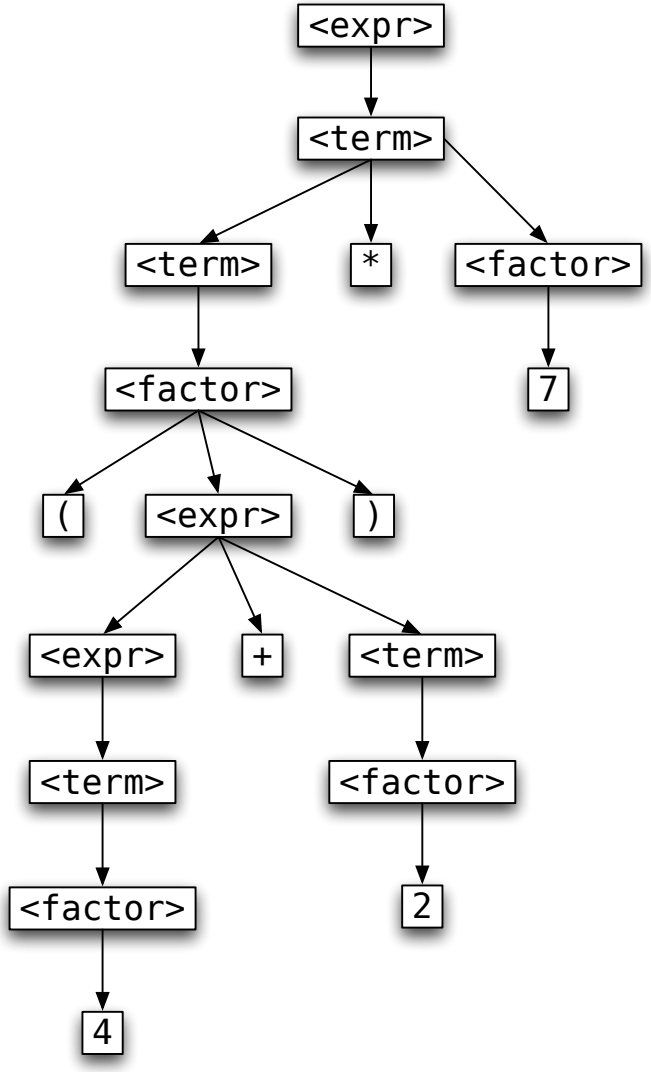
1. Is the string in the language?
2. Why? (Parse tree or other evidence)

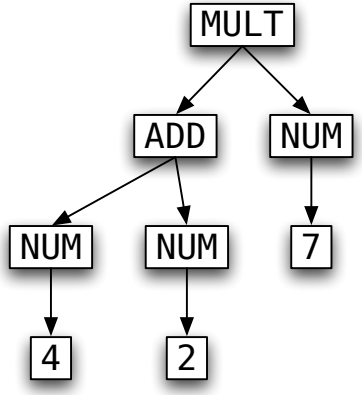
Evidence

```
<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.

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

Bogus Backtracking

```
S -> Ac | Bc
A -> a | c | ab
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 b
```

[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**.

Then

- ✓ It would be easy to write correct `Consume` functions
- ✓ Our parser would run in linear time (in the length of the input).

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?

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.
- ✓ Accept if in accept state *and* [Rich] stack is empty

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

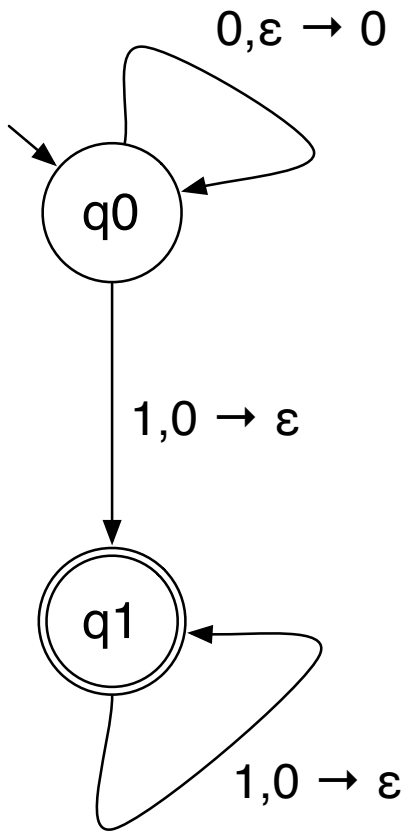
- ✓ A finite set K of states
- ✓ A finite alphabet Σ
- ✓ A finite "stack alphabet" Γ
- ✓ A start state $q_0 \in Q$
- ✓ Accepting states $A \subseteq Q$
- ✓ Transitions, each from the set

$$\left(K \times (\Sigma \cup \{\varepsilon\}) \times (\Gamma \cup \{\varepsilon\})\right) \times \left(K \times (\Gamma \cup \{\varepsilon\})\right)$$

PDA Example

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

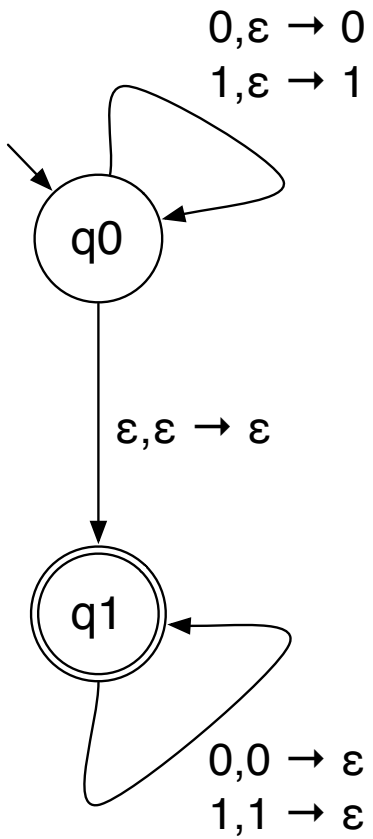
$$\{ 0^n 1^n \mid n > 0 \}$$



PDA Example

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

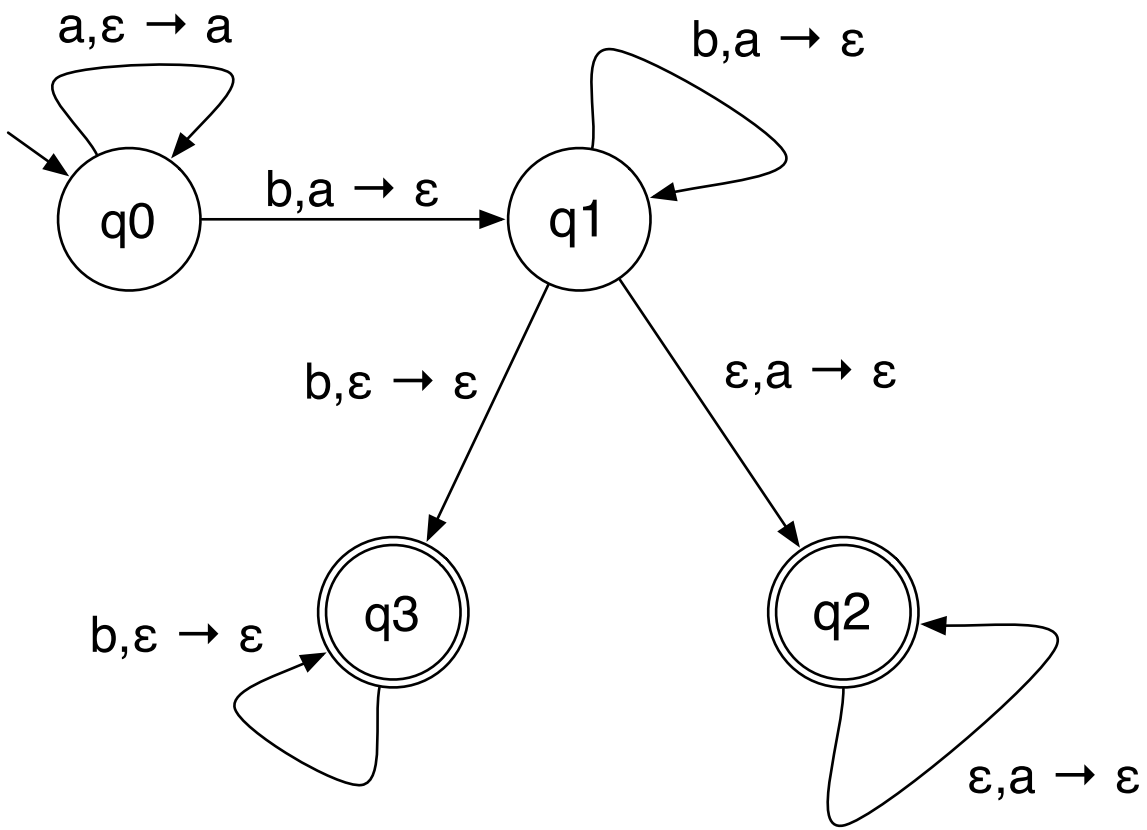
$$\{ ww^R \mid w \in \{0, 1\}^* \}$$



PDA Example

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

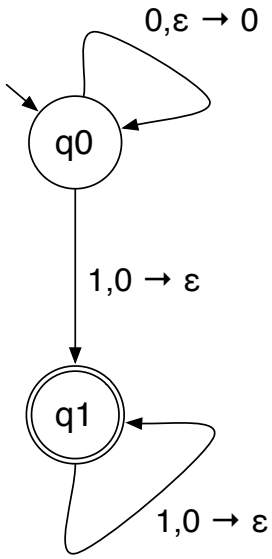
$$\{ a^i b^j \mid i, j > 0 \ \wedge \ i \neq j \}$$

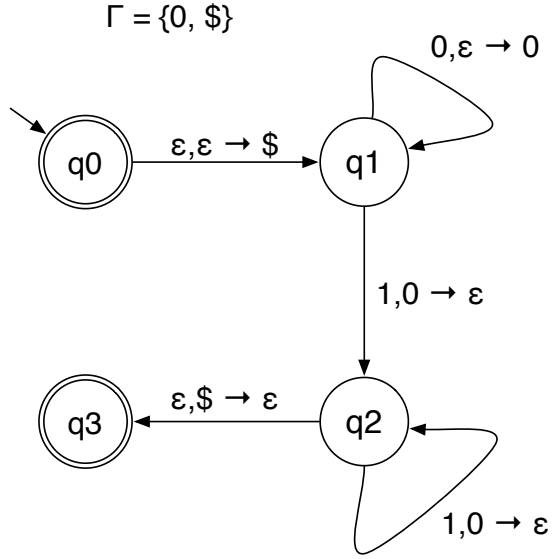


PDA Variations

There are many equivalent ways of defining PDAs.

Some authors say a PDA accepts *whenever* it's in an accept state.
How would we fix the previous PDAs?





Other authors (e.g., Rich) let transitions push or pop any finite string from the stack.

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 rule (No backtracking required!)

2. PDAs recognize only CFLs

- Turn a PDA into a grammar that simulates it
- See the book: for every two pair of states $\mathbf{p}, \mathbf{q}$ and each nonterminal $\mathbf{x}$, define a nonterminal $\langle \mathbf{p}, \mathbf{x}, \mathbf{q} \rangle$, strings that get you from state $\mathbf{p}$ to $\mathbf{q}$ in the PDA with the same stack except for popping $\mathbf{x}$.

Closure Properties

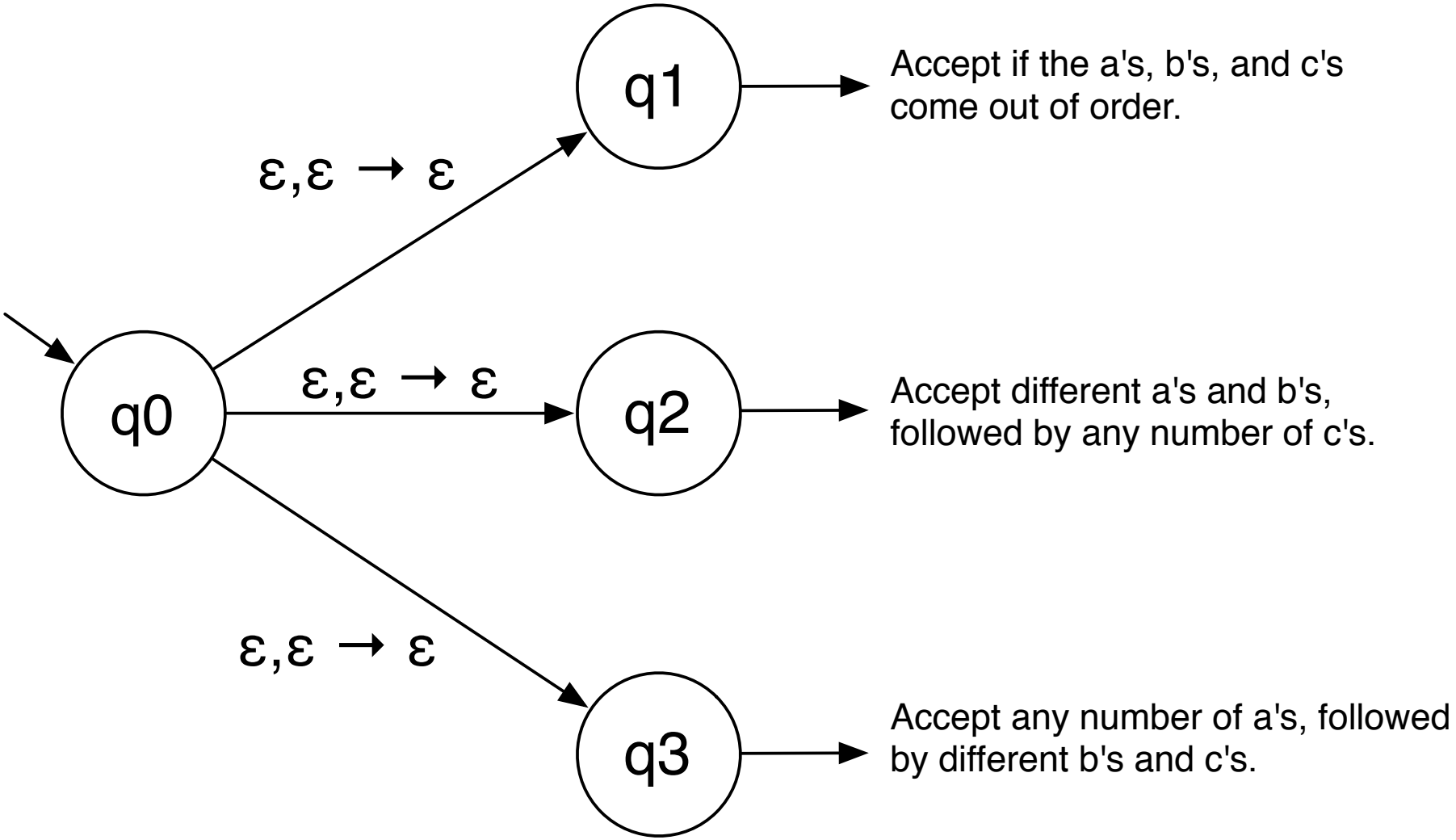
Context-Free Languages are:

- ✓ 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 $O(n^3)$ (n = length of input)
- ✓ Requires a grammar in "Chomsky Normal Form"
  - All rules of the form $A \to a$ or $A \to BC$.
- ✓ Key ideas:
  - For every substring, what nonterminals produce it?
  - Dynamic programming for efficiency

Dynamic Programming

Recursive expressions, such as

$$\begin{array}{lcll} f(n) & := & 1 & \text{if } n < 3 \\ f(n) & := & f(n-1) + f(n-3) & \text{otherwise} \end{array}$$

are very clear, but inefficient if taken literally. Two roughly-equivalent solutions:

- ✓ Memoization: keep track of what f values have been computed
- ✓ Dynamic Programming: Compute all f values 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 $S \in a(1,n)$

$$\begin{aligned} a(i,i) &= \{ C \mid C \to x_i \} \\ a(i,k) &= \{ C \mid C \to AB \wedge A \in a(i,j) \wedge B \in a(j+1,k) \} \end{aligned}$$

CYK "Wavefront" Matrix

a(1, 1)	a(1, 2)	a(1, 3)	a(1, 4)		a(1, n-1)	a(1, n)
	a(2, 2)	a(2, 3)	a(2, 4)		a(2, n-1)	a(2, n)
		a(3, 3)	a(3, 4)		a(3, n-1)	a(3, n)
			a(4, 4)			
					a(n-1, n-1)	a(n-1, n)
						a(n, n)

Each entry is computed from entries in its same row and column, e.g. a(1, 4) from a(1,1) and a(2, 4), a(1, 2) and a(3, 4), a(1, 3) and a(4, 4).

CYK Example

Let's parse (( ) ( )) using the grammar

$$
\begin{aligned}
S &\to LT \\
T &\to SR \\
S &\to LR \\
S &\to SS \\
L &\to (\\
R &\to)
\end{aligned}
$$