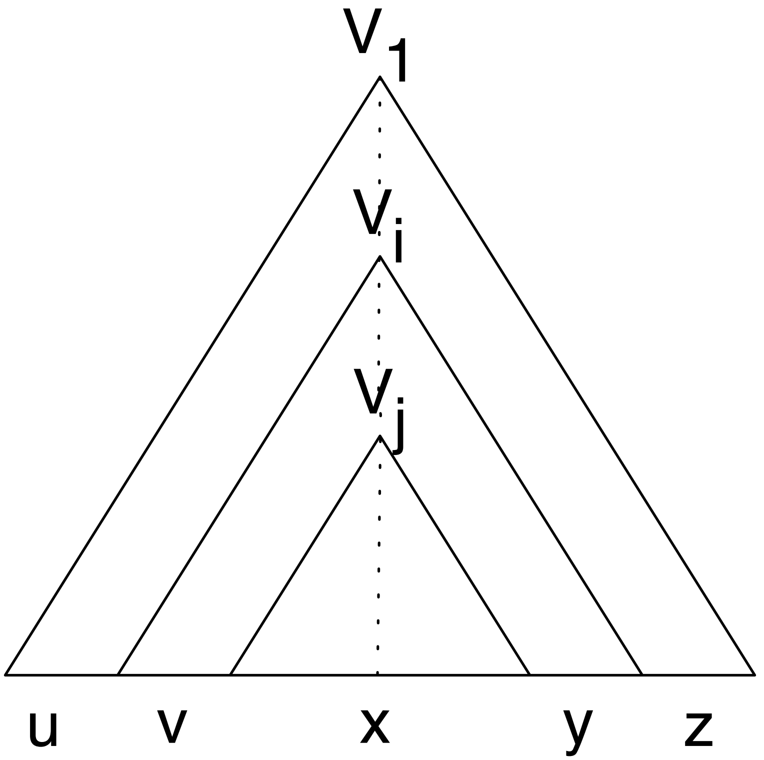
Parsing & PDA

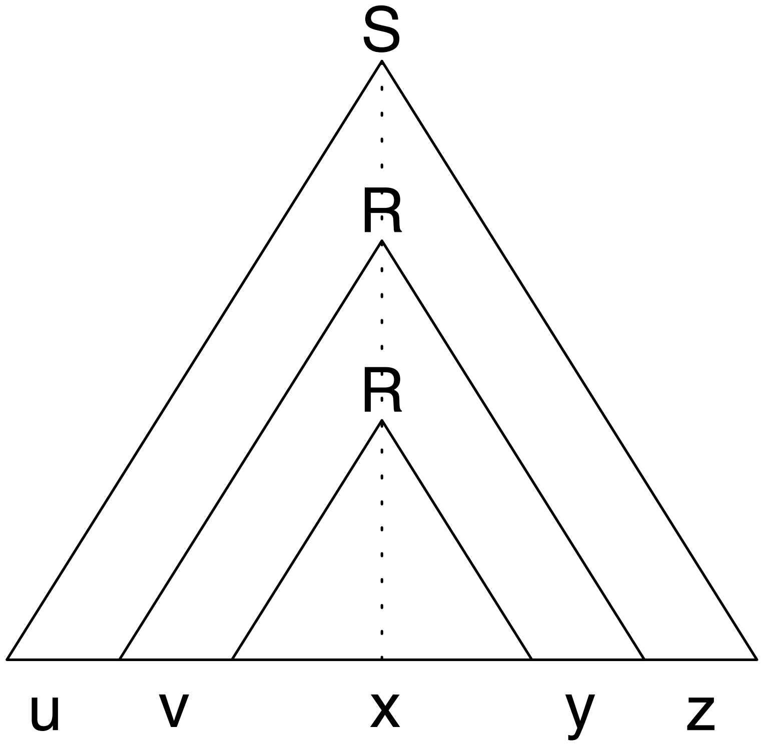
No, the other PDA.
No, not that one either.

November 4, 2010

Applying Pigeonhole

- Assume we have a string in L whose <u>shortest</u> parse tree has height ≥ |V|+1. [height = edges]





Pumping

Final questions:

(1) How do we know v and y aren't both ε?

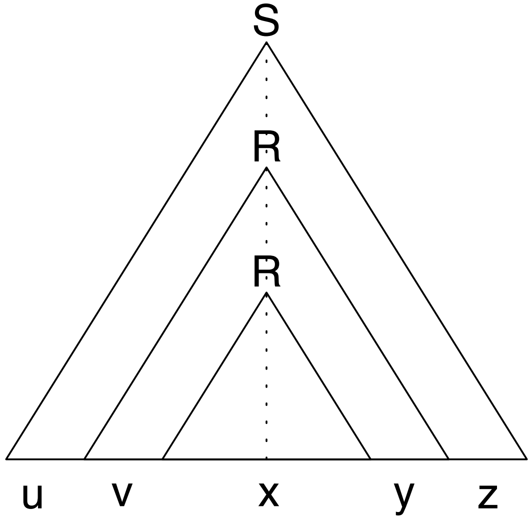
E.g.,

$R \Rightarrow P$

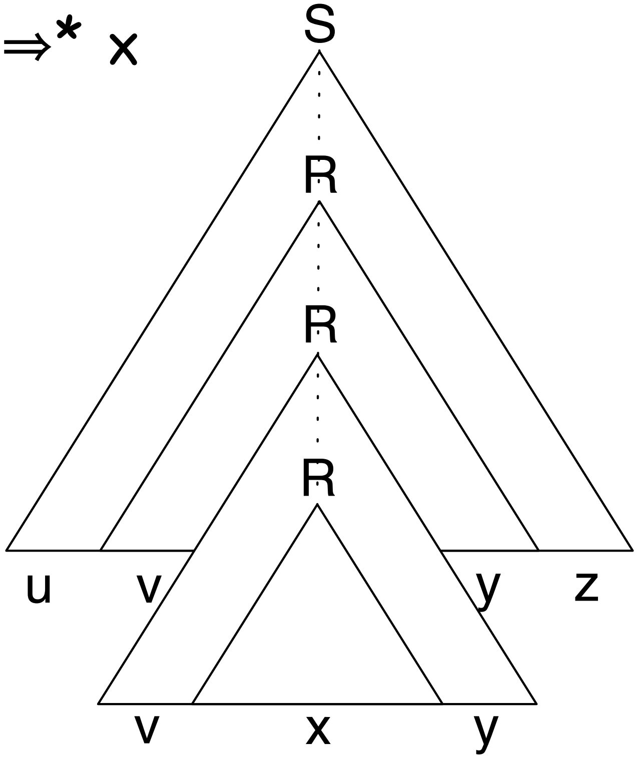
$P \Rightarrow Q$

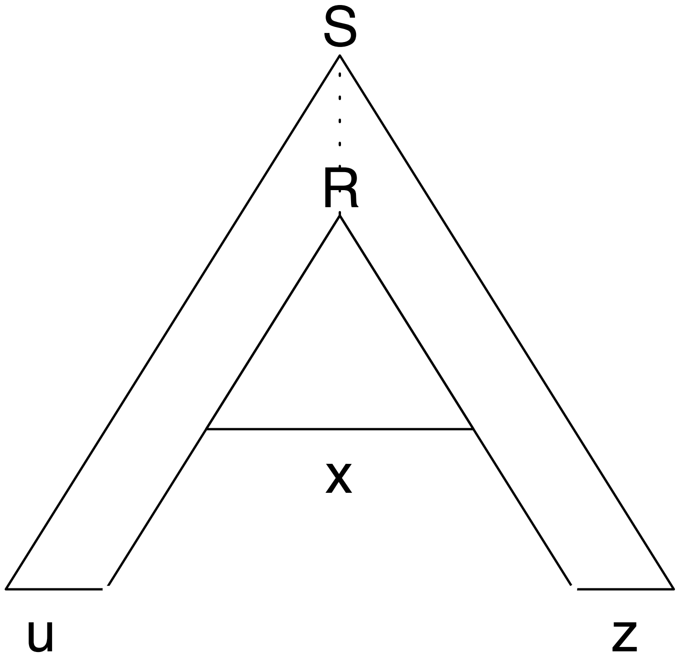
$Q \Rightarrow R$

$R \Rightarrow^* x$



(2) Is there a point beyond which all strings have "tall" parse trees?





Regular Grammars

- A grammar is said to be **regular** if its rules are of the following forms:

$$X \rightarrow a$$

$$X \rightarrow \varepsilon$$

$$X \rightarrow aY$$

Example

S → 1B

B → 1B

B → 0C

C → 0S

S → 0S

C → 1B

C → ε

Context-Free Grammars

- ✓ An unrestricted grammar consists of
  1. A set V of **variables** (a.k.a. **nonterminals**)
  2. A disjoint set Σ (of **terminals**)
  3. A set of rules of the form LEFT → RIGHT where
     LEFT $\in (V \cup \Sigma)^+$ and RIGHT $\in (V \cup \Sigma)^*$
  4. One designated $S \in V$, called the **start variable** (a.k.a. **start symbol**

- ✓ If every LEFT is a single variable, the grammar is said to be **context-free**.

Rewriting Strings

- If we have a rule LEFT → RIGHT, we can replace LEFT by RIGHT <u>inside</u> any string:

$$\alpha\text{LEFT}\beta \Rightarrow \alpha\text{RIGHT}\beta$$

- The language of a grammar G is the set

$$L(G) = \{\ w \in \Sigma^* \mid S \Rightarrow^* w\ \}$$

- The same language L might be the language of many different grammars.
  - L is said to be a **context-free language** if it can be generated by **at least one** context-free grammar.

CFG Example

$S \rightarrow \varepsilon$

$S \rightarrow 0B$

$S \rightarrow 1A$

$A \rightarrow 0S$

$A \rightarrow 1AA$

$B \rightarrow 1S$

$B \rightarrow 0BB$

- What is the start symbol?
- What are the terminals and nonterminals?
- What strings does this grammar generate?

The Parsing Problem

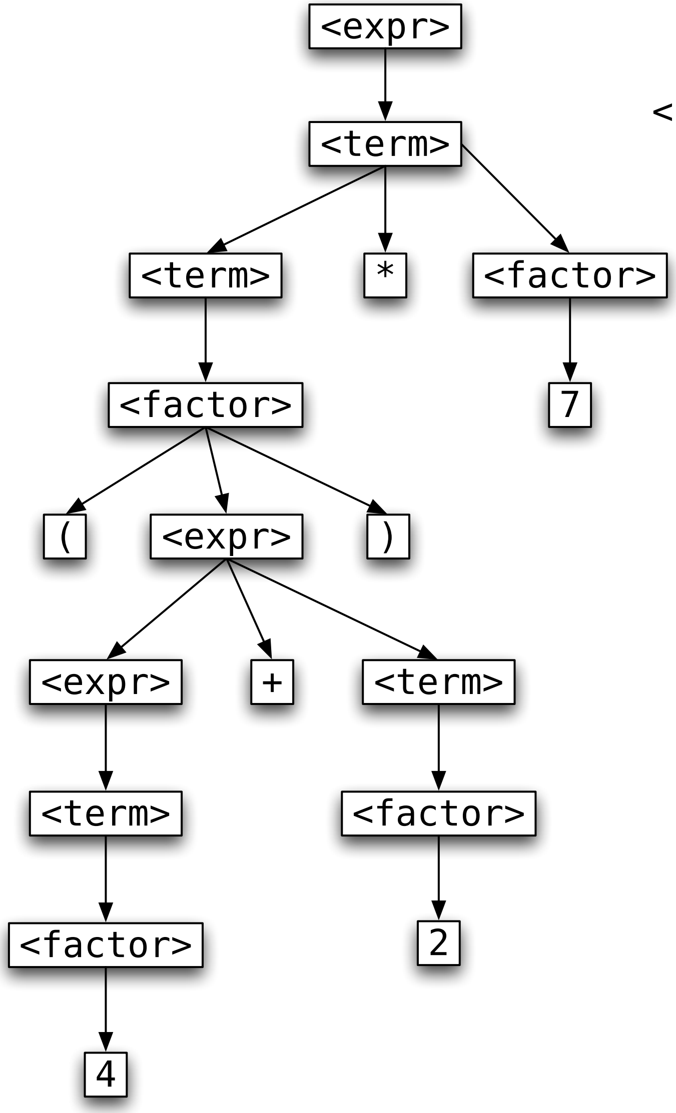
- Given a grammar and a string
  - Is the string in the language.
  - Usually, produce a “proof”
    - Parse tree or some abstraction thereof.

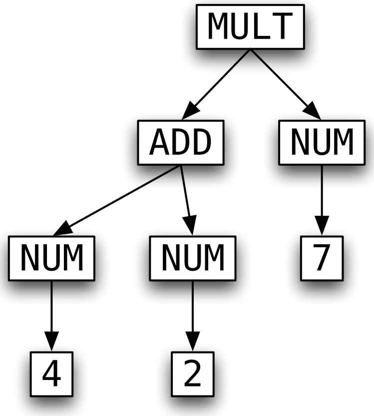
Parse Tree vs. AST

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

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

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





42

Naive Parsing

- Backtracking search.
  - Try all ways to derive the string.
- Inefficient.
- Harder to implement that you might think…

Bogus Backtracking

S -> Aa | Ba
A -> a | c | ac
B -> Bb | b

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

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

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

LL(k) Grammars

- If each Consume function “knew” the right choice
  - we’d never need to backtrack
  - we’d never get tangled in infinite loops
  - It would be easy to write correct Consume functions
  - Our parser would run in linear time.

S -> aA | bB

- We say that a grammar is **LL(k)** if, by “peeking ahead” no more than k tokens, we can guarantee a decision that is
  - correct
  - unique (unambiguous)
- A language is LL(k) if there exists at least one LL(k) grammar.

LL(k) Grammars?

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

Recursive Descent

- ✓ Naive backtracking works for LL(k) grammars!

```
S -> s | a B $
B -> d | c B B
```

```
Consume_S():
    try to consume s
    if this fails, consume a, Consume_B(), then consume $

Consume_B():
    try to consume d
    if this fails, consume c, Consume_B(), then Consume_B()
```

Predictive Parsing

- ✓ Generalization of Recursive Descent.
- ✓ Stack of predictions, initially S.
- ✓ At each step, predict or match.
- ✓ Succeed if we run out of input & predictions at the same time.

Digression

- 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.

Example

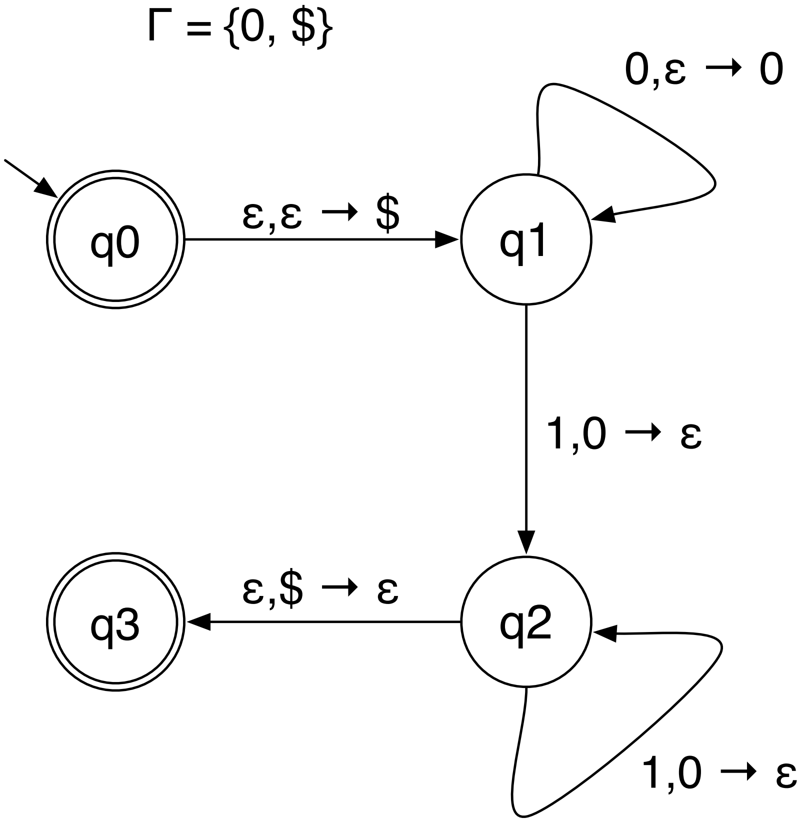
- Using a state machine and a stack, how can we recognize $\{ 0^n1^n \mid n \geq 0 \}$?

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

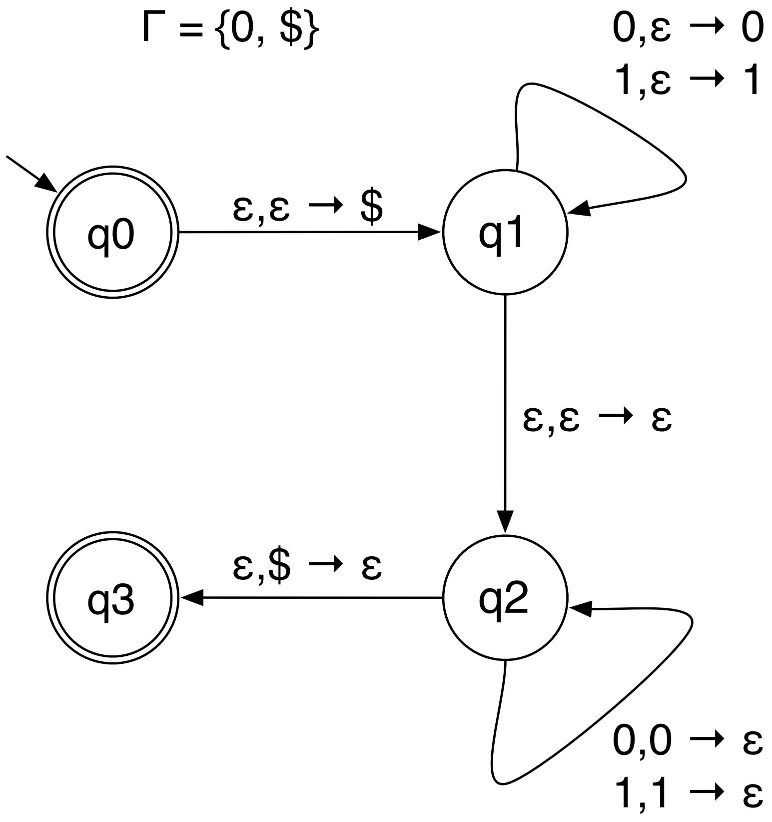
Example

- ✓ Using a state machine and a stack, how can we recognize { 0^n1^n | n ≥ 0 }?



Example

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



PDA vs CFG

- ✓ 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.