PDA->CFG Example, A Little Parsing, and Pumping

Announcements

- HW5 just removed
- HW6 released soon
- Should be caught back up

Last time: Terminating PDAs?

- P _accepts_ w if reaches an accept state. When we require emptying the stack, then reaches an accept state w/an empty stack.

- Must P either accept or reject every string in Σ*?

Last time: Terminating PDAs?

- P _accepts_ w if reaches an accept state. When we require emptying the stack, then reaches an accept state w/an empty stack.

- Must P either accept or reject every string in Σ*?
  - No! (In fact, stay tuned!!)

- So we have a mismatch between what's true, and what's computed!

Last time: Terminating PDAs?

- Membership for CFLs is /decidable/!

- (It will turn out) there _is_ an algorithm.
  - Cf. CKY algorithm, to solve in a classy manner
  - What's a slower, more obvious way to show it?
  - Generate and test!
  - PDA->CFG, CFG->GNF (Greibach Normal Form) ..?

Terminating PDAs?

- Membership for CFLs is /decidable/!
- (It will turn out) there _is_ an algorithm.
  - Cf. CKY algorithm, to solve in a classy manner
  - What's a slower, more obvious way to show it?
  - Generate and test!
  - PDA->CFG, CFG->GNF (Greibach Normal Form) ..?
- This means a string of length n generated in n steps.
- Generate and test!

Transform PDA->CFG

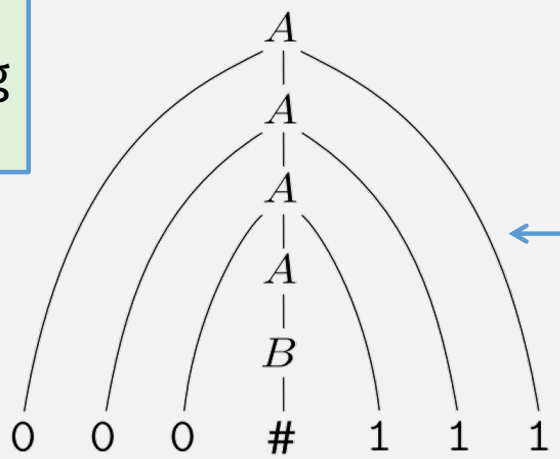
- 3 Kinds of rules:
- all states p, $A_{pp} \to \varepsilon$ (EASY)
- The stack: either the first symbol (meaning ”the” one) pushed is the last one popped, or not.
- p,q,r,s , u, a, b $A_{pq} \to aA_{rs}b$ when delta pushes u on read a and pops u on read b. When they’re matched like parens
- p,q,r $A_{pq} \to A_{pr}A_{rq}$ (every kind of indirection)

Example

Previously: CFLs, CFGs, and Parse Trees

Generating strings:
start with start variable,
Apply rules to get a string
(and parse tree)

$$A \to 0A1$$
$$A \to B$$
$$B \to \#$$



In practice,
opposite is more interesting:
start with a string,
and **parse** it into parse tree

$$A \Rightarrow 0A1 \Rightarrow 00A11 \Rightarrow 000A111 \Rightarrow 000B111 \Rightarrow 000\#111$$

Generating vs Parsing

- In practice, **parsing** a string is more important than **generating** one
  - E.g., a **compiler first** parses source code into a (tree) data representation
  - Actually, *any* program accepting a string input must first parse it
- But a compiler / parser (algorithm) must be deterministic
- The PDAs we've seen are non-deterministic (like NFAs)
- So: to model parsers, we need a **Deterministic** PDA (DPDA)
  - Analogous to DFA vs NFA

Last time: DPDA: Formal Definition

The language of a DPDA is called a ***deterministic context-free language***.

DEFINITION 2.39

A ***deterministic pushdown automaton*** is a 6-tuple $(Q, \Sigma, \Gamma, \delta, q_0, F)$, where Q, Σ, Γ, and F are all finite sets, and

1. Q is the set of states,
2. Σ is the input alphabet,
3. Γ is the stack alphabet,
4. $\delta: Q \times \Sigma_\varepsilon \times \Gamma_\varepsilon \longrightarrow (Q \times \Gamma_\varepsilon) \cup \{\emptyset\}$ is the transition function,
5. $q_0 \in Q$ is the start state, and
6. $F \subseteq Q$ is the set of accept states.

Key restriction:
DPDA has only **1 transition** for a given state, input, and stack op
(just like DFA vs NFA)

The transition function δ must satisfy the following condition. For every $q \in Q$, $a \in \Sigma$, and $x \in \Gamma$, exactly one of the values

$$\delta(q, a, x), \delta(q, a, \varepsilon), \delta(q, \varepsilon, x), \text{ and } \delta(q, \varepsilon, \varepsilon)$$

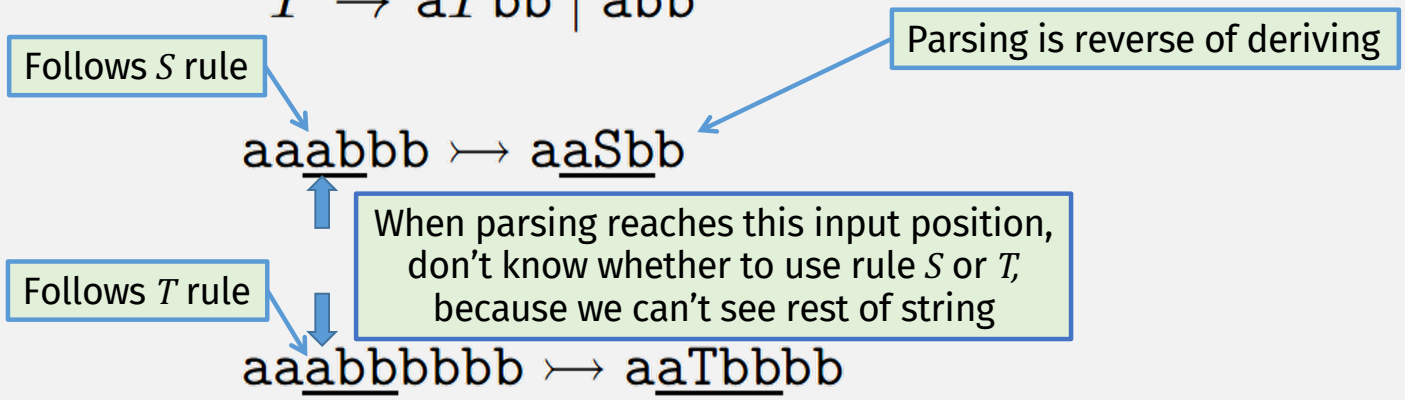
is not $\emptyset$.

A ***pushdown automaton*** is a 6-tuple

1. Q is the set of states,
2. Σ is the input alphabet,
3. Γ is the stack alphabet,
4. $\delta: Q \times \Sigma_\varepsilon \times \Gamma_\varepsilon \longrightarrow \mathcal{P}(Q \times \Gamma_\varepsilon)$
5. $q_0 \in Q$ is the start state, and
6. $F \subseteq Q$ is the set of accept states.

Last time: DPDAs are Not Equivalent to PDAs!

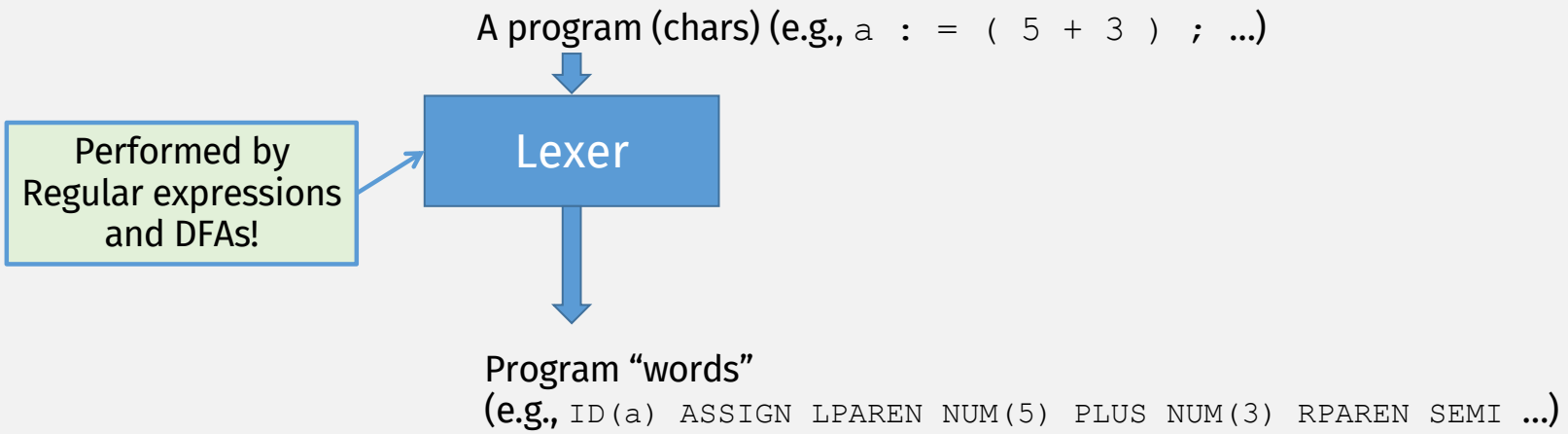
$$R \to S \mid T$$
$$S \to \mathtt{a}S\mathtt{b} \mid \mathtt{ab}$$
$$T \to \mathtt{a}T\mathtt{bb} \mid \mathtt{abb}$$



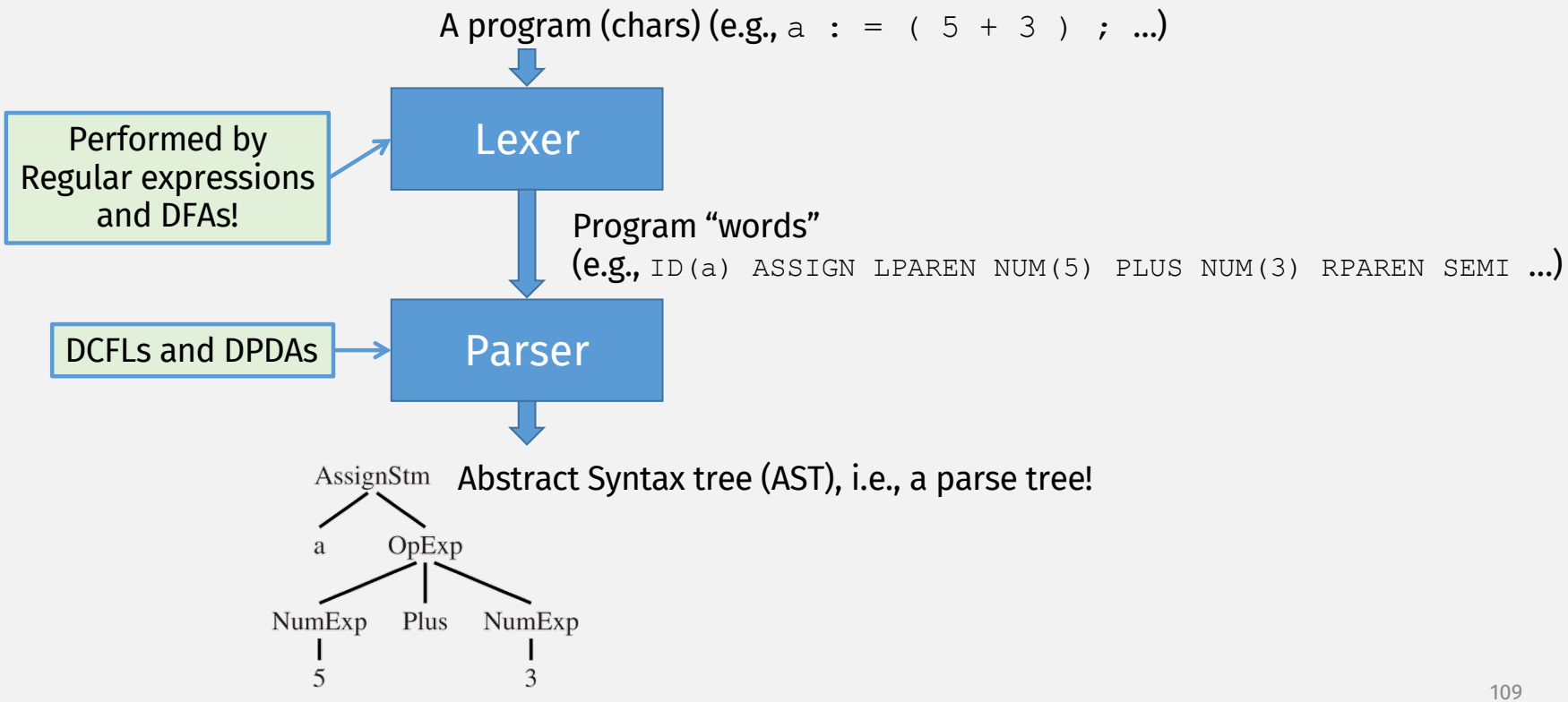
A PDA can non-deterministically "try all rules", but a DPDA must choose one

PDAs recognize CFLs, but a DPDA only recognizes DCFLs! (a subset of CFLs)

Compiler Stages



Compiler Stages



A Parser Implementation

```
%{
int yylex(void);
void yyerror(char *s) { EM_error(EM_tokPos, "%s", s); }
%}
%token ID WHILE BEGIN END DO IF THEN ELSE SEMI ASSIGN
%start prog
%%

prog: stmlist

stm : ID ASSIGN ID
    | WHILE ID DO stm
    | BEGIN stmlist END
    | IF ID THEN stm
    | IF ID THEN stm ELSE stm

stmlist : stm
        | stmlist SEMI stm
```

Just write the CFG!

A "`yacc`" tool translates this to a C program implementation of a parser

Parsing

$$R \to S \mid T$$
$$S \to \mathtt{a}S\mathtt{b} \mid \mathtt{ab}$$
$$T \to \mathtt{a}T\mathtt{bb} \mid \mathtt{abb}$$

$\mathtt{aa\underline{ab}bb} \rightarrowtail \mathtt{a\underline{aSb}b}$

A parser must be able to choose one correct rule, when reading input left-to-right

$\mathtt{aa\underline{abb}bbbb} \rightarrowtail \mathtt{a\underline{aTbb}bb}$

LL parsing

- L = left-to-right
- L = leftmost derivation

$$S \rightarrow \text{if } E \text{ then } S \text{ else } S$$
$$S \rightarrow \text{begin } S\ L$$
$$S \rightarrow \text{print } E$$

$$L \rightarrow \text{end}$$
$$L \rightarrow ;\ S\ L$$

$$E \rightarrow \text{num} = \text{num}$$

```
if 2 = 3 begin print 1; print 2; end else print 0
```

Parsing Game: Guess which rule applies?

LL parsing

- L = left-to-right
- L = leftmost derivation

$$S \rightarrow \text{if } E \text{ then } S \text{ else } S$$
$$S \rightarrow \text{begin } S\ L$$
$$S \rightarrow \text{print } E$$

$$L \rightarrow \text{end}$$
$$L \rightarrow ;\ S\ L$$

$$E \rightarrow \text{num} = \text{num}$$

```
if 2 = 3 begin print 1; print 2; end else print 0
```

LL parsing

- L = left-to-right
- L = leftmost derivation

$$S \rightarrow \text{if } E \text{ then } S \text{ else } S$$
$$S \rightarrow \text{begin } S\ L$$
$$S \rightarrow \text{print } E$$

$$L \rightarrow \text{end}$$
$$L \rightarrow ;\ S\ L$$

$$E \rightarrow \text{num} = \text{num}$$

```
if 2 = 3 begin print 1; print 2; end else print 0
```

LL parsing

- L = left-to-right
- L = leftmost derivation

$$S \rightarrow \text{if } E \text{ then } S \text{ else } S$$
$$S \rightarrow \text{begin } S\ L$$
$$S \rightarrow \text{print } E$$

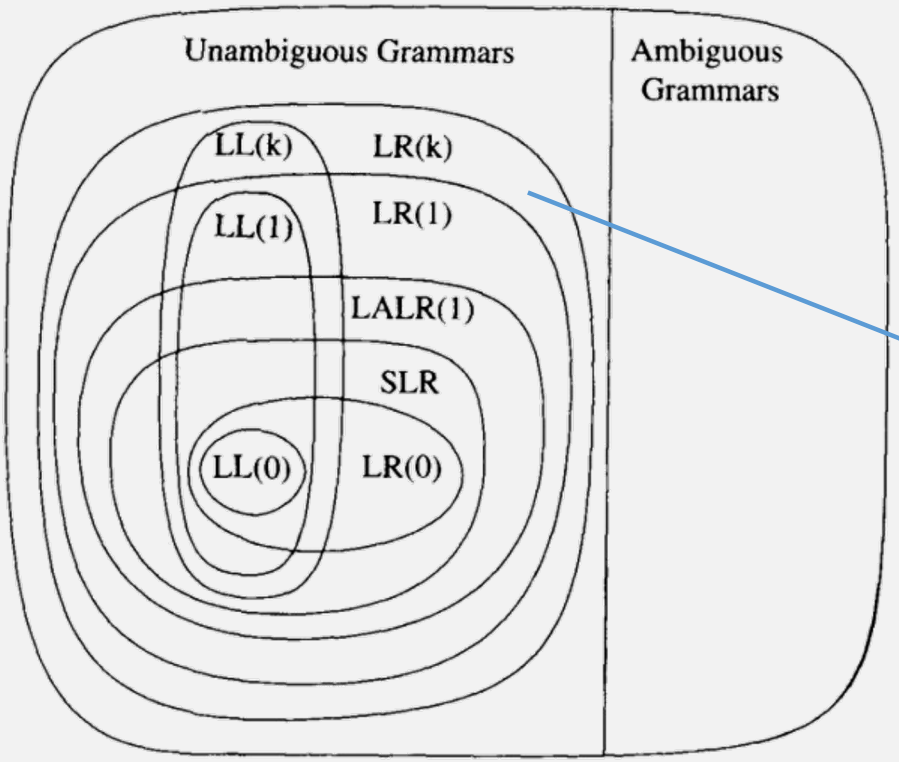
$$L \rightarrow \text{end}$$
$$L \rightarrow ;\ S\ L$$

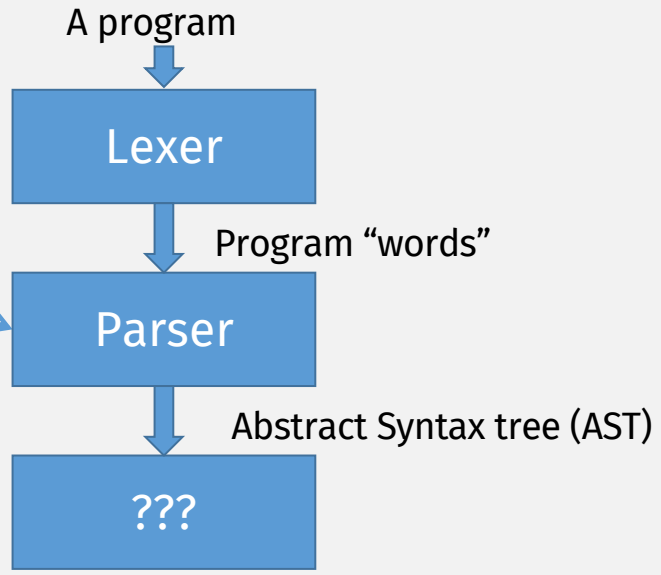
$$E \rightarrow \text{num} = \text{num}$$

```
if 2 = 3 begin print 1; print 2; end else print 0
```

"Prefix" languages (like Scheme/Lisp) are easily parsed with LL parsers

To learn more, take a Compilers Class!





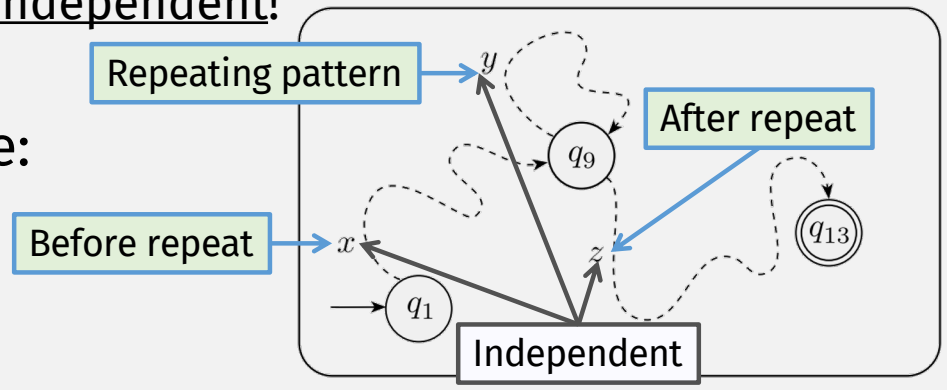
The Pumping Lemma for CFLs

Flashback: Pumping Lemma for Reg Langs

- The Pumping Lemma describes how strings **repeat**
- Strs in a regular lang can (only) repeat using Kleene pattern
  - But the substrings are independent!
- A non-regular language:

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

Kleene can't express this pattern:
2nd part depends on (length of) 1st part



- How do CFLs repeat?

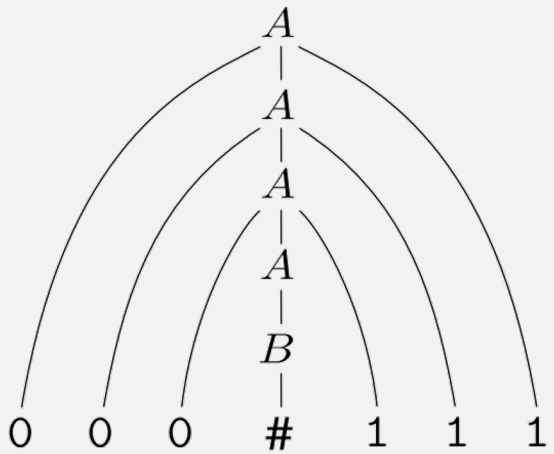
Repetition and Dependency in CFLs

Parts before/after repetition are linked

Repetition

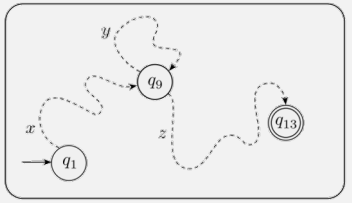
$$A \to 0A1$$
$$A \to B$$
$$B \to \#$$

$$\{0^n\#1^n \mid n \geq 0\}$$

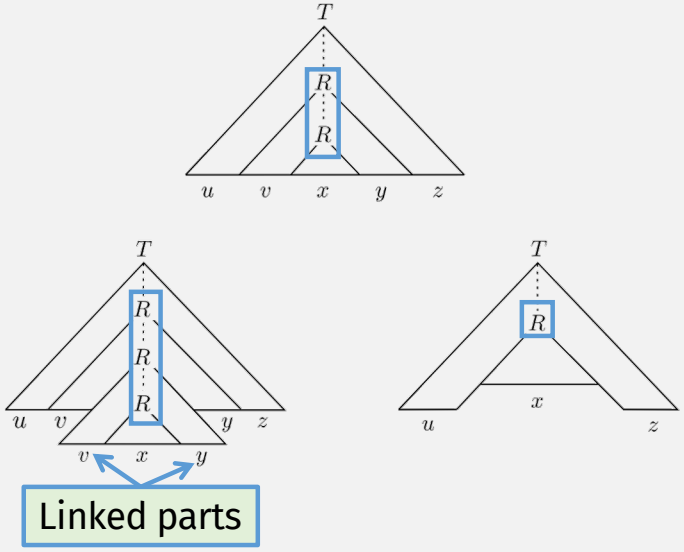


$$A \Rightarrow 0A1 \Rightarrow 00A11 \Rightarrow 000A111 \Rightarrow 000B111 \Rightarrow 000\#111$$

How Can Strings in CFLs Repeat?



- Strings in regular languages repeat states
- Strings in CFLs repeat subtrees in the parse tree

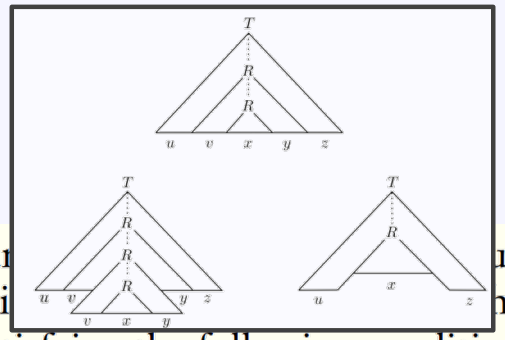


Pumping Lemma for CFLS

Pumping lemma for context-free languages If A is a context-free language, then there is a number p (the pumping length) where, if s is any string in A of length at least p, then s may be divided into five pieces $s = uvxyz$ satisfying the conditions

Now there are <u>two</u> pumpable parts.
But they must be <u>pumped together</u>!

1. for each $i \geq 0$, $uv^ixy^iz \in A$,
2. $|vy| > 0$, and
3. $|vxy| \leq p$.



Pumping lemma If A is a regular [illegible] number p (the pumping length) where if s is any stri[illegible]hen s may be divided into three pieces, $s = xyz$, satisfying the following conditions:

1. for each $i \geq 0$, $xy^iz \in A$,
2. $|y| > 0$, and
3. $|xy| \leq p$.

Check-in Quiz

On Gradescope