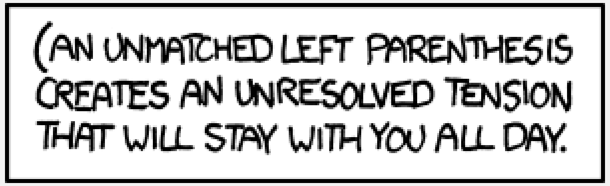
UMB CS 420

Non-CFLs

Tuesday, October 25, 2022



Announcements

- HW 5 in
  - ~~Due 10/23 11:59pm EST~~

- HW 6 out
  - Due 10/30 11:59pm EST

Last Time: Generating vs Parsing

- In practice, **parsing** a string more important than **generating** one
  - E.g., a **compiler** (first step) parses source code into a parse tree
  - (Actually, *any* program with string inputs must first parse it)

But:

- PDAs are <u>non-deterministic</u> (like NFAs)
- Compiler's parsing algorithm must be <u>deterministic</u>

- <u>So</u>: to model parsers, we need a **Deterministic PDA** (DPDA)

Last Time: DPDA: Formal Definition

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

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. $\delta\colon Q \times \Sigma_\varepsilon \times \Gamma_\varepsilon \longrightarrow (Q \times \Gamma_\varepsilon) \cup \{\emptyset\}$ is the transition function
5. $q_0 \in Q$ is the start state, and
6. $F \subseteq Q$ is the set of accept states.

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. $\delta\colon Q \times \Sigma_\varepsilon \times \Gamma_\varepsilon \longrightarrow \mathcal{P}(Q \times \Gamma_\varepsilon)$
5. $q_0 \in Q$ is the start state, and
6. $F \subseteq Q$ is the set of accept states.

Difference: DPDA has only **one possible action**, for any given **state**, **input**, and **stack op** (similar to **DFA** vs **NFA**)

This must take into account ε reads or stack ops! E.g., if $\delta(q, \text{a}, X)$ is valid, then $\delta(q, \varepsilon, X)$ must not be

DPDAs are <u>Not</u> Equivalent to PDAs!

Parsing = generating reversed:
- start with string
- end with parse tree

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

- **PDA:** can non-deterministically "<u>try all</u> rules" (abandoning failed attempts);
- **DPDA:** must <u>choose one</u> rule at each step!

Should use S rule

aaabbb ⟶ aaSbb

aaa

aaabbbbb ⟶ aaTbbbb

Should use T rule

When parsing reaches this input position, which rule to use, S or T?

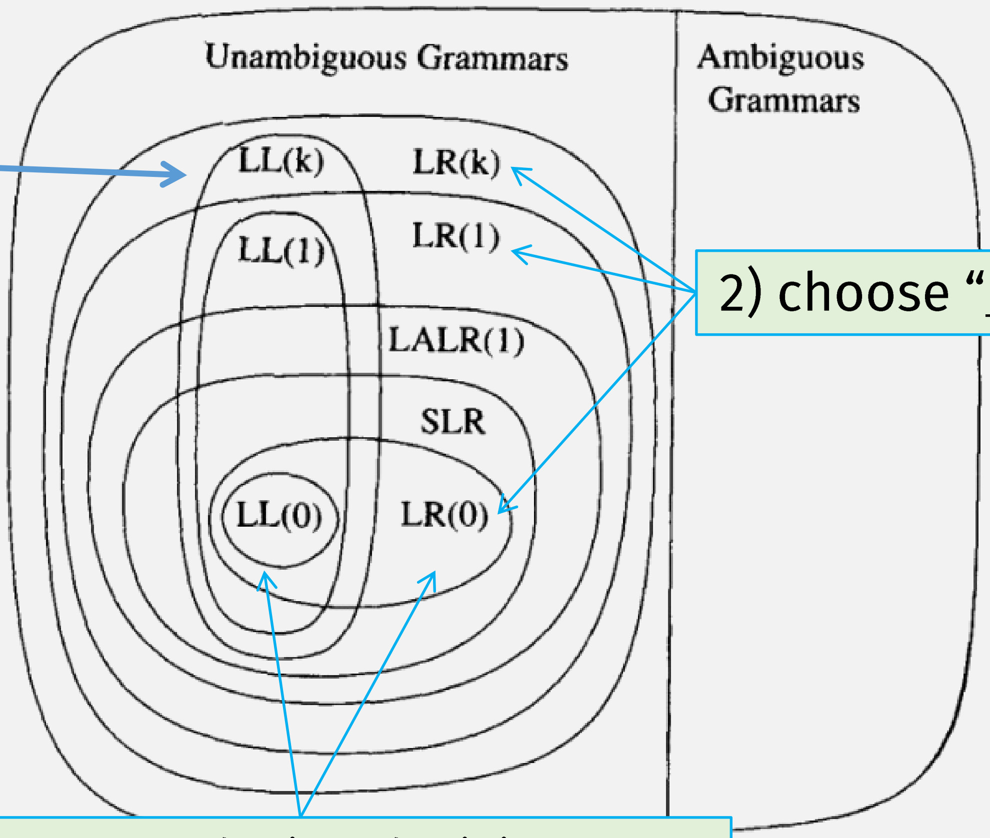
To choose "correct" rule, need to "<u>look ahead</u>" at rest of the input!

PDAs recognize CFLs, but <u>DPDAs only recognize DCFLs!</u> (a <u>subset</u> of CFLs)

Subclasses of CFLs

DCFLs

Programming language parsers / compilers are ideally in here



2) choose "look ahead" amount

2 parser design decisions:
1) Parse from left, or from right

All CFLS

LL parsing

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

Game: "You're the Parser":
Guess which rule applies?

1 $S \rightarrow$ if E then S else S

2 $S \rightarrow$ begin S L

3 $S \rightarrow$ print E

4 $L \rightarrow$ end

5 $L \rightarrow$; S L

6 $E \rightarrow$ num = num

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

LL parsing

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

1 $S \to \text{if } E \text{ then } S \text{ else } S$

2 $S \to \text{begin } S\ L$

3 $S \to \text{print } E$

4 $L \to \text{end}$

5 $L \to ;\ S\ L$

6 $E \to \text{num} = \text{num}$

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

LL parsing

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

1 $S \rightarrow \text{if } E \text{ then } S \text{ else } S$

2 $S \rightarrow \text{begin } S\ L$

3 $S \rightarrow \text{print } E$

4 $L \rightarrow \text{end}$

5 $L \rightarrow ;\ S\ L$

6 $E \rightarrow \text{num} = \text{num}$

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

LL parsing

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

1 $S \rightarrow \text{if } E \text{ then } S \text{ else } S$

2 $S \rightarrow \text{begin } S\ L$

3 $S \rightarrow \text{print } E$

4 $L \rightarrow \text{end}$

5 $L \rightarrow \ ;\ S\ L$

6 $E \rightarrow \text{num} = \text{num}$

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

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

LR parsing

- **L** = left-to-right
- **R** = rightmost derivation

1 $S \rightarrow S \; ; \; S$
2 $S \rightarrow \text{id} := E$
3 $S \rightarrow \text{print} \; (\; L \;)$
4 $E \rightarrow \text{id}$
5 $E \rightarrow \text{num}$
6 $E \rightarrow E + E$

```
a := 7;
b := c + (d := 5 + 6, d)
```

When parse is here, can't determine whether it's an assign (: =) or addition (+)

Need to save input (lookahead) to some memory, like a **stack**! this is a job for a (D)PDA!

push

State name

Stack	*Input*	*Action*
1	a := 7 ; b := c + (d := 5 + 6 , d) $	*shift* "push"
1 id_4	:= 7 ; b := c + (d := 5 + 6 , d) $	*shift*
1 id_4 $:=_6$	7 ; b := c + (d := 5 + 6 , d) $	*shift*
1 id_4 $:=_6$ num_{10}	; b := c + (d := 5 + 6 , d) $	*reduce* $E \rightarrow \text{num}$
1 id_4 $:=_6$ E_{11}	; b := c + (d := 5 + 6 , d) $	*reduce* $S \rightarrow \text{id} := E$
1 S_2	; b := c + (d := 5 + 6 , d) $	*shift*

LR parsing

- **L** = left-to-right
- **R** = rightmost derivation

$$S \rightarrow S\ ;\ S \qquad E \rightarrow \text{id}$$
$$S \rightarrow \text{id} := E \qquad E \rightarrow \text{num}$$
$$S \rightarrow \text{print}\ (\ L\ ) \qquad E \rightarrow E + E$$

Stack	*Input*	*Action*
$_1$	a := 7 ; b := c + (d := 5 + 6 , d) $	*shift*
$_1$ id$_4$	:= 7 ; b := c + (d := 5 + 6 , d) $	*shift*
$_1$ id$_4$:=$_6$	7 ; b := c + (d := 5 + 6 , d) $	*shift*
$_1$ id$_4$:=$_6$ num$_{10}$	; b := c + (d := 5 + 6 , d) $	*reduce* $E \rightarrow$ num
$_1$ id$_4$:=$_6$ E_{11}	; b := c + (d := 5 + 6 , d) $	*reduce* $S \rightarrow$ id:=E
$_1$ S_2	; b := c + (d := 5 + 6 , d) $	*shift*

LR parsing

- **L** = left-to-right
- **R** = rightmost derivation

$$S \rightarrow S\ ;\ S \qquad E \rightarrow \text{id}$$
$$S \rightarrow \text{id} := E \qquad E \rightarrow \text{num}$$
$$S \rightarrow \text{print}\ (\ L\ ) \qquad E \rightarrow E + E$$

Stack	*Input*	*Action*
$_1$	a := 7 ; b := c + (d := 5 + 6 , d) $	*shift*
$_1$ id$_4$	:= 7 ; b := c + (d := 5 + 6 , d) $	*shift*
$_1$ id$_4$:=$_6$	7 ; b := c + (d := 5 + 6 , d) $	*shift*
$_1$ id$_4$:=$_6$ num$_{10}$	; b := c + (d := 5 + 6 , d) $	*reduce* $E \rightarrow$ num
$_1$ id$_4$:=$_6$ E_{11}	; b := c + (d := 5 + 6 , d) $	*reduce* $S \rightarrow$ id:=E
$_1$ S_2	; b := c + (d := 5 + 6 , d) $	*shift*

LR parsing

- **L** = left-to-right
- **R** = rightmost derivation

1 $S \to S\,;\,S$		4 $E \to \text{id}$
2 $S \to \text{id} := E$		5 $E \to \text{num}$
3 $S \to \text{print}\,(\,L\,)$		6 $E \to E + E$

Can determine (rightmost) rule

Stack	*Input*	*Action*
$_1$	a := 7 ; b := c + (d := 5 + 6 , d) \$	*shift*
$_1$ id$_4$	:= c + (d := 5 + 6 , d) \$	*shift*
$_1$ id$_4$:=$_6$	:= c + (d := 5 + 6 , d) \$	*shift*
$_1$ id$_4$:=$_6$ num$_{10}$	; b := c + (d := 5 + 6 , d) \$	*reduce* $E \to$ num
$_1$ id$_4$:=$_6$ E_{11}	b := c + (d := 5 + 6 , d) \$	*reduce* $S \to$ id:=E
$_1$ S_2	; b := c + (d := 5 + 6 , d) \$	*shift*

LR parsing

- **L** = left-to-right
- **R** = rightmost derivation

1 $S \rightarrow S\ ;\ S$

2 $S \rightarrow \text{id} := E$

3 $S \rightarrow \text{print}\ (\ L\ )$

4 $E \rightarrow \text{id}$

5 $E \rightarrow \text{num}$

6 $E \rightarrow E + E$

Stack	*Input*	*Action*
$_1$	a := 7 ; b := c + (d := 5 + 6 , d) \$	*shift*
$_1$ id_4	:= 7 ; b := c + (d := 5 + 6 , d) \$	*shift*
$_1$ id_4 $:=_6$	= c + (d := 5 + 6 , d) \$	*shift*
$_1$ id_4 $:=_6$ num_{10}	= c + (d := 5 + 6 , d) \$	*reduce* $E \rightarrow \text{num}$
$_1$ id_4 $:=_6$ E_{11}	; b := c + (d := 5 + 6 , d) \$	*reduce* $S \rightarrow \text{id} := E$
$_1$ S_2	b := c + (d := 5 + 6 , d) \$	*shift*

Can determine (rightmost) rule

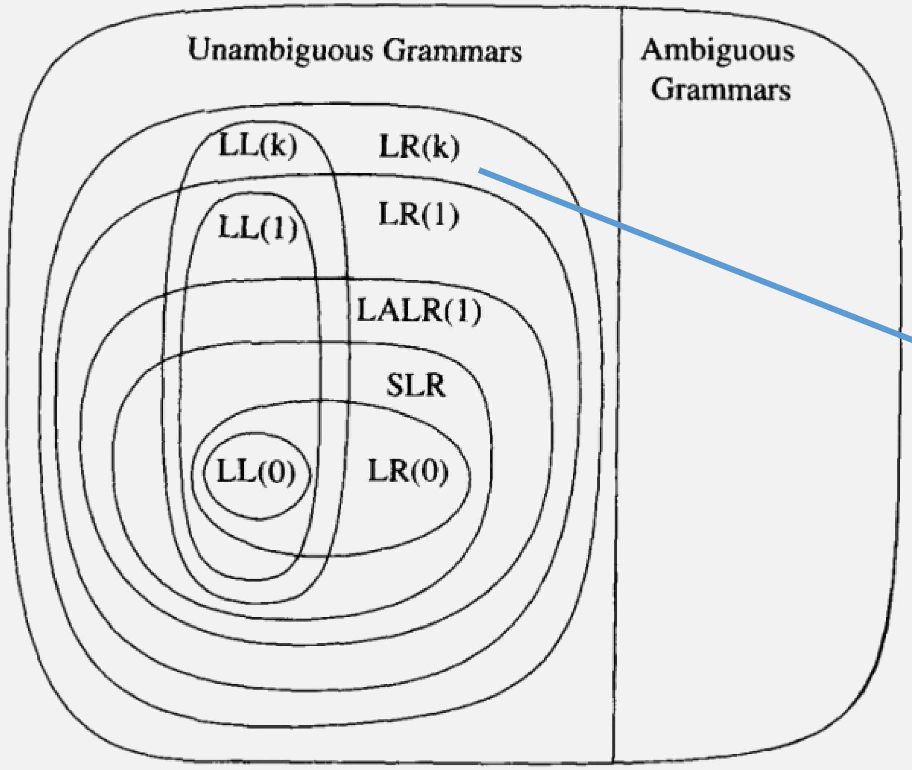
LR parsing

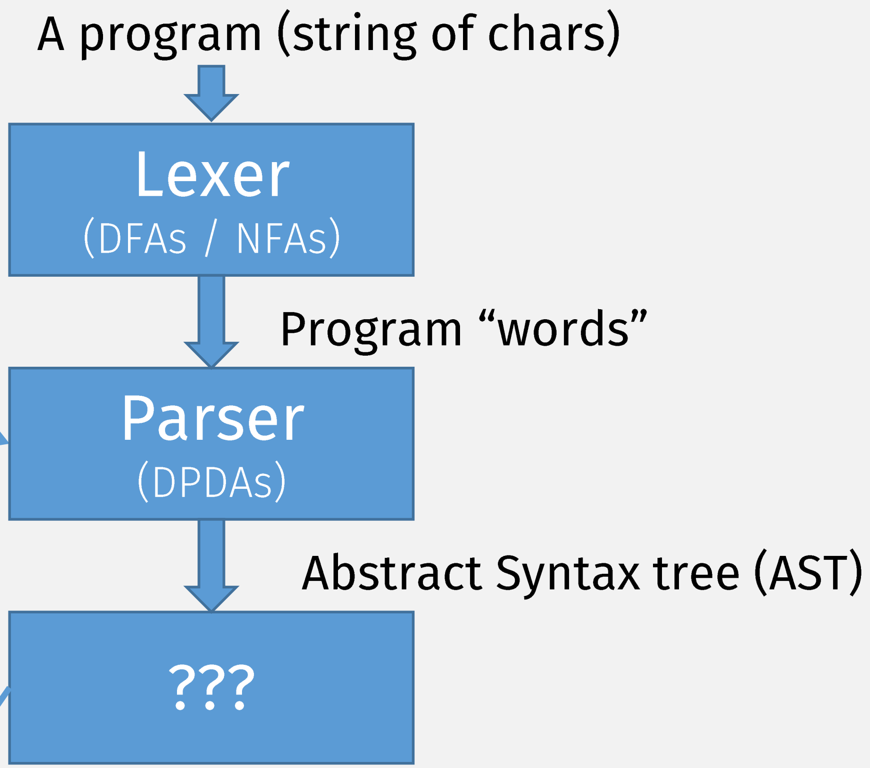
- **L** = left-to-right
- **R** = rightmost derivation

$$S \rightarrow S \;;\; S \qquad E \rightarrow \text{id}$$
$$S \rightarrow \text{id} := E \qquad E \rightarrow \text{num}$$
$$S \rightarrow \text{print} \; (\; L \;) \qquad E \rightarrow E + E$$

Stack	*Input*	*Action*
$_1$	`a := 7 ; b := c + ( d := 5 + 6 , d ) $`	*shift*
$_1$ id$_4$	`:= 7 ; b := c + ( d := 5 + 6 , d ) $`	*shift*
$_1$ id$_4$:=$_6$	`7 ; b := c + ( d := 5 + 6 , d ) $`	*shift*
$_1$ id$_4$:=$_6$ num$_{10}$	`; b := c + ( d := 5 + 6 , d ) $`	*reduce* $E \rightarrow$ num
$_1$ id$_4$:=$_6$ E_{11}	`; b := c + ( d := 5 + 6 , d ) $`	*reduce* $S \rightarrow$ id:=E
$_1$ S_2	`; b := c + ( d := 5 + 6 , d ) $`	*shift*

To learn more, take a Compilers Class!





This phase needs computation that goes beyond CFLs

Flashback: Pumping Lemma for Regular Langs

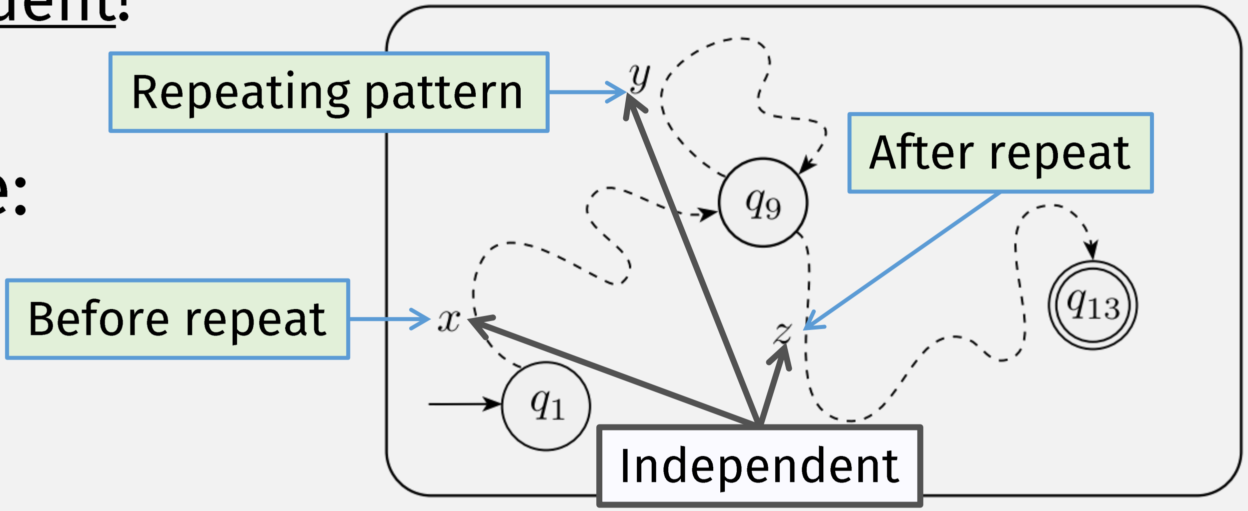
- Pumping Lemma <u>describes how strings **repeat**</u>

- Regular language strings repeat using Kleene start operation
  - <u>substrings are independent</u>!

- A non-regular language:

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

Kleene star can't express this pattern:
2nd part <u>depends</u> on (length of) 1st part



- Q: How do CFLs repeat?

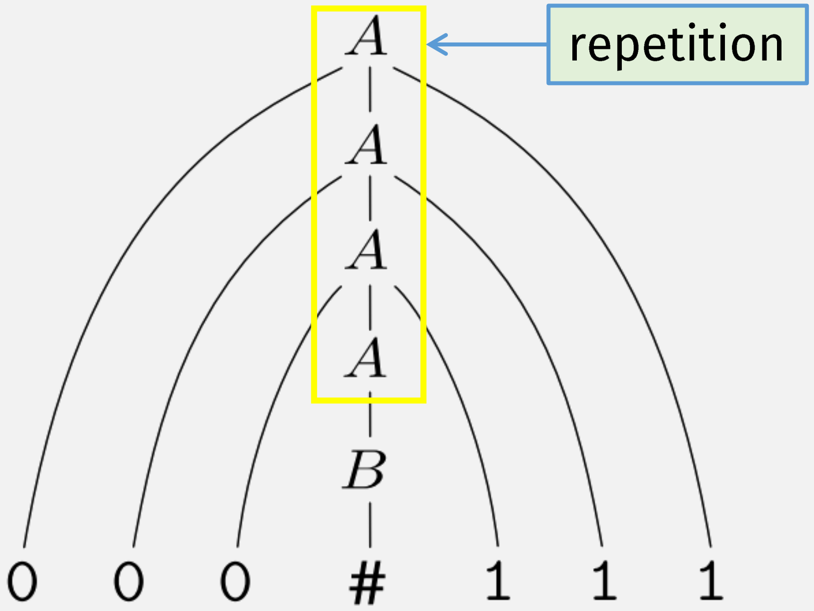
Repetition and Dependency in CFLs

Parts before/after repetition point are linked

Repetition

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

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

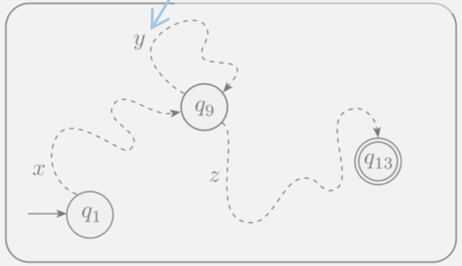


$$A \Rightarrow 0A1 \Rightarrow 00A11 \Rightarrow 000A111 \Rightarrow 000B111 \Rightarrow 000\#111$$

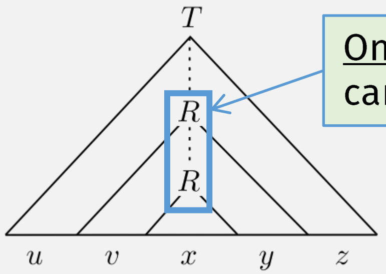
How Do Strings in CFLs Repeat?

NFA can take loop transition any number of times, to process repeated y in input

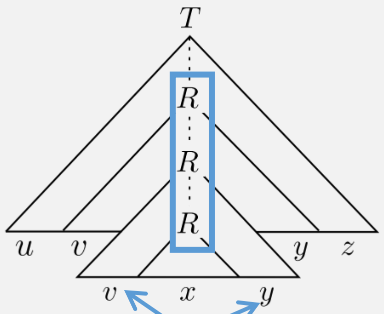
- Strings in regular languages <u>repeat states</u>

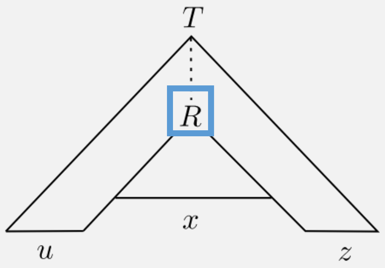


- Strings in CFLs <u>repeat subtrees</u> in the parse tree



<u>One</u> repeated subtree means that it can be repeated <u>any</u> number of times





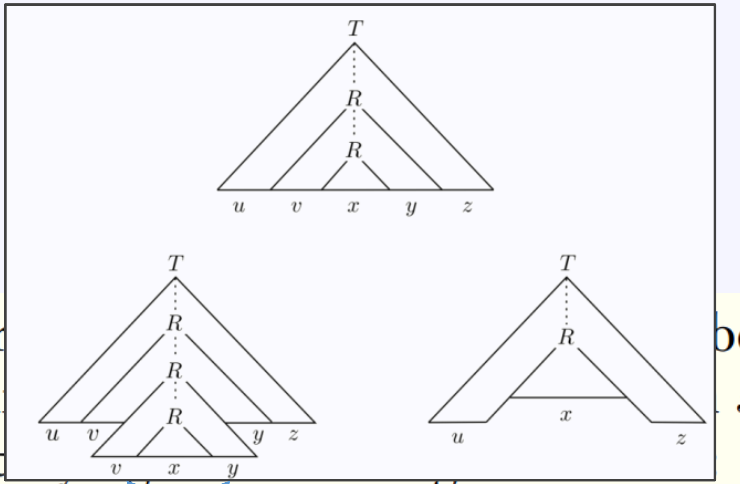
Linked parts

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 ... ber p (the pumping length) where if s is any stri... s may be divided into three pieces, $s = xyz$, sat...

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

Two pumpable parts, pumped together

A Non CFL example

$$\text{language } B = \{\mathtt{a}^n\mathtt{b}^n\mathtt{c}^n \mid n \geq 0\} \text{ is not context free}$$

Intuition

- Strings in CFLs can have two parts that are "pumped" together
- This language requires three parts to be "pumped" together
- So it's not a CFL!

Want to prove: $a^n b^n c^n$ **is not** a CFL

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

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

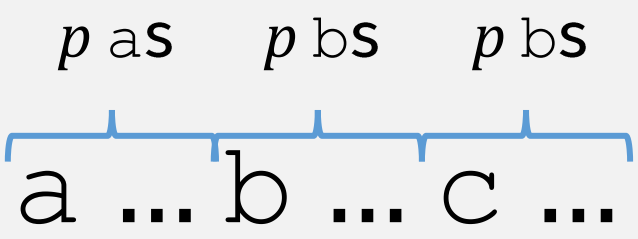
Reminder: CFL Pumping lemma says: all strings $a^n b^n c^n \geq$ length p **are splittable** into $uvxyz$ where v and y are pumpable

Proof (by contradiction):

Now we must find a contradiction …

- Assume: $a^n b^n c^n$ **is** a CFL
  - So it must satisfy the pumping lemma for CFLs
  - I.e., all strings $\geq$ length p are pumpable
- Counterexample = $a^p b^p c^p$

Contradiction if: string $\geq$ length p that is **not splittable** into $uvxyz$ where v and y are pumpable



Want to prove: $a^n b^n c^n$ **is not** a CFL

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

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

Possible Splits

Proof (by contradiction):

contradiction

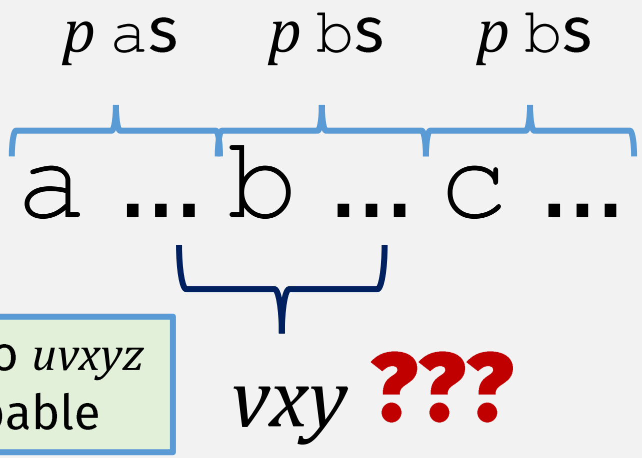
- **Assume:** $a^n b^n c^n$ **is** a CFL
  - So it must satisfy the pumping lemma for CFLs
  - I.e., all strings ≥ length p are pumpable
- **Counterexample** = $a^p b^p c^p$

Contradiction if: string ≥ length p that is **not splittable** into $uvxyz$ where v and y are pumpable

- Possible Splits (using condition # 3: $|vxy| \leq p$)

Not pumpable

- ☒ vyx is all as
- ☒ vyx is all bs
- ☒ vyx is all cs
- ☒ vyx has as and bs
- ☒ vyx has bs and cs



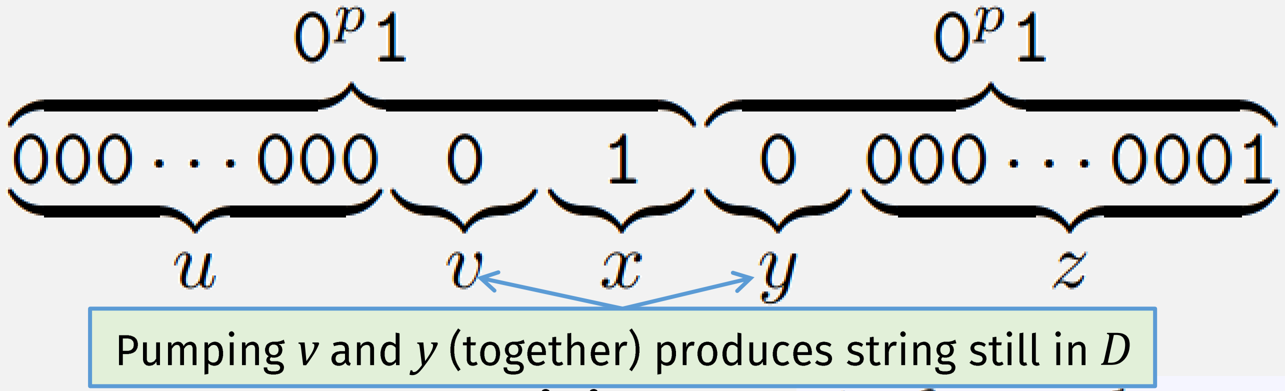
$a^p b^p c^p$ **cannot be split** into $uvxyz$ where v and y are pumpable

So $a^n b^n c^n$ **is not a CFL**
(justification:
contrapositive of CFL pumping lemma)

Another Non-CFL $D = \{ww \mid w \in \{0,1\}^*\}$

<u>Be careful when choosing **counterexample**</u> s: 0^p10^p1

This s **can** be pumped according to <u>CFL pumping lemma</u>:



- CFL Pumping Lemma conditions:
  - ☑ **1.** for each $i \geq 0$, $uv^ixy^iz \in A$,
  - ☑ **2.** $|vy| > 0$, and
  - ☑ **3.** $|vxy| \leq p$.

This doesn't prove that the language is a CFL!
It only means that <u>this attempt</u> to prove that the language is not a CFL <u>failed</u>.

Another Non-CFL $D = \{ww \mid w \in \{0,1\}^*\}$

- Need another counterexample string *s*:

If *vyx* is contained in first or second half, then any pumping will break the match

$$\overbrace{0^p 1^p}\overbrace{0^p 1^p}$$

$$0^p \underbrace{1^p 0^p} 1^p$$

So *vyx* must straddle the middle

But any pumping still breaks the match because order is wrong

- CFL Pumping Lemma conditions:

1. for each $i \geq 0$, $uv^ixy^iz \in A$,

2. $|vy| > 0$, and

3. $|vxy| \leq p$.

Now we have proven that **this language is not a CFL!**

A Practical Non-CFL

- **XML**
  - ELEMENT → <TAG>CONTENT</TAG>
  - Where TAG is any string

- XML also looks like this non-CFL: $D = \{ww \mid w \in \{0,1\}^*\}$

- This means XML is not context-free!
  - Note: HTML *is* context-free because …
  - … there are only a finite number of tags,
  - so they can be embedded into a finite number of rules.

- In practice:
  - XML is parsed as a CFL, with a CFG
  - Then matching tags checked in a 2nd pass with a more powerful machine …

Next Time: A More Powerful Machine …

M_1 accepts its input if it is in language: $B = \{w\#w \mid w \in \{0,1\}^*\}$

M_1 = "On input string w:

1. Zig-zag across the tape to corresponding positions on either side of the # symbol to check whether these positions contain the same symbol. If they do not, or if no # is found, *reject*. Cross off symbols as they are checked to keep track of which symbols correspond.

Infinite memory, initially starts with input

Can move to, and read/write from, arbitrary memory locations!

In-class quiz 10/25

See gradescope