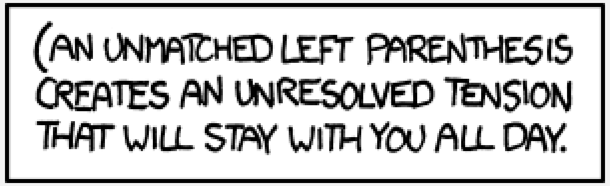
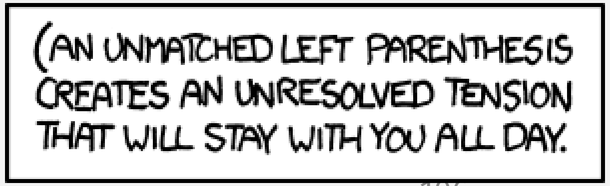
Deterministic CFLs, PDAs, and Parsing

Wednesday, October 6, 2021



Announcements

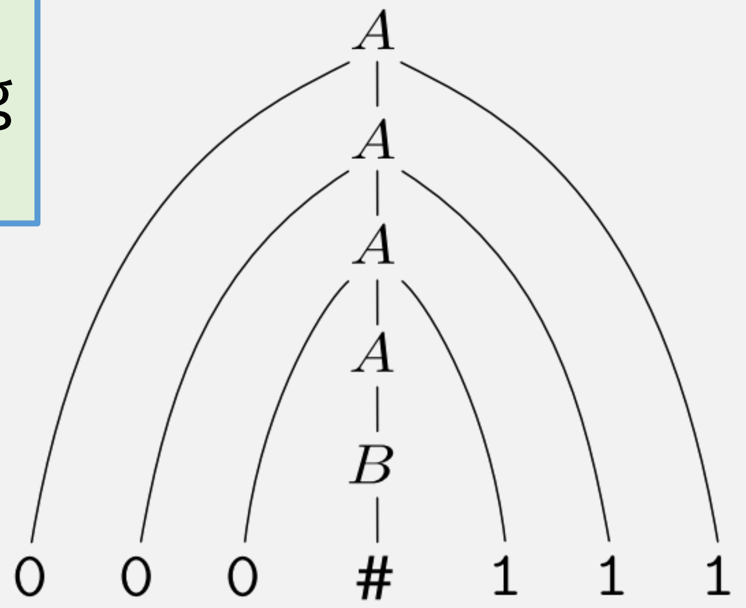
- Reminder: no class next Monday 10/11
- HW4 due Sunday 10/17 11:59pm
  - second Sunday from today



Previously: CFLs, CFGs, and Parse Trees

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

$$A \rightarrow 0A1$$
$$A \rightarrow B$$
$$B \rightarrow \#$$

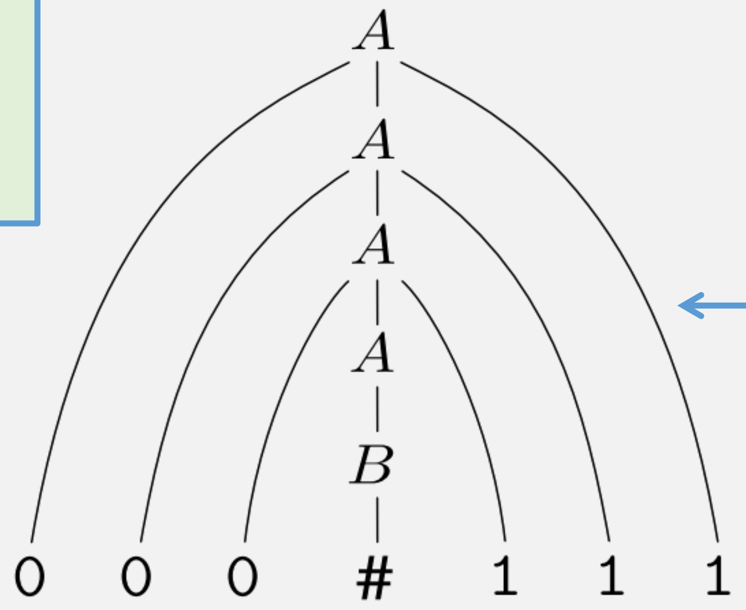


$$A \Rightarrow 0A1 \Rightarrow 00A11 \Rightarrow 000A111 \Rightarrow 000B111 \Rightarrow 000\#111$$

Today: Generating vs Parsing

Generating strings:
start with start variable,
then apply rules to get a
string and parse tree

$$A \rightarrow 0A1$$
$$A \rightarrow B$$
$$B \rightarrow \#$$



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

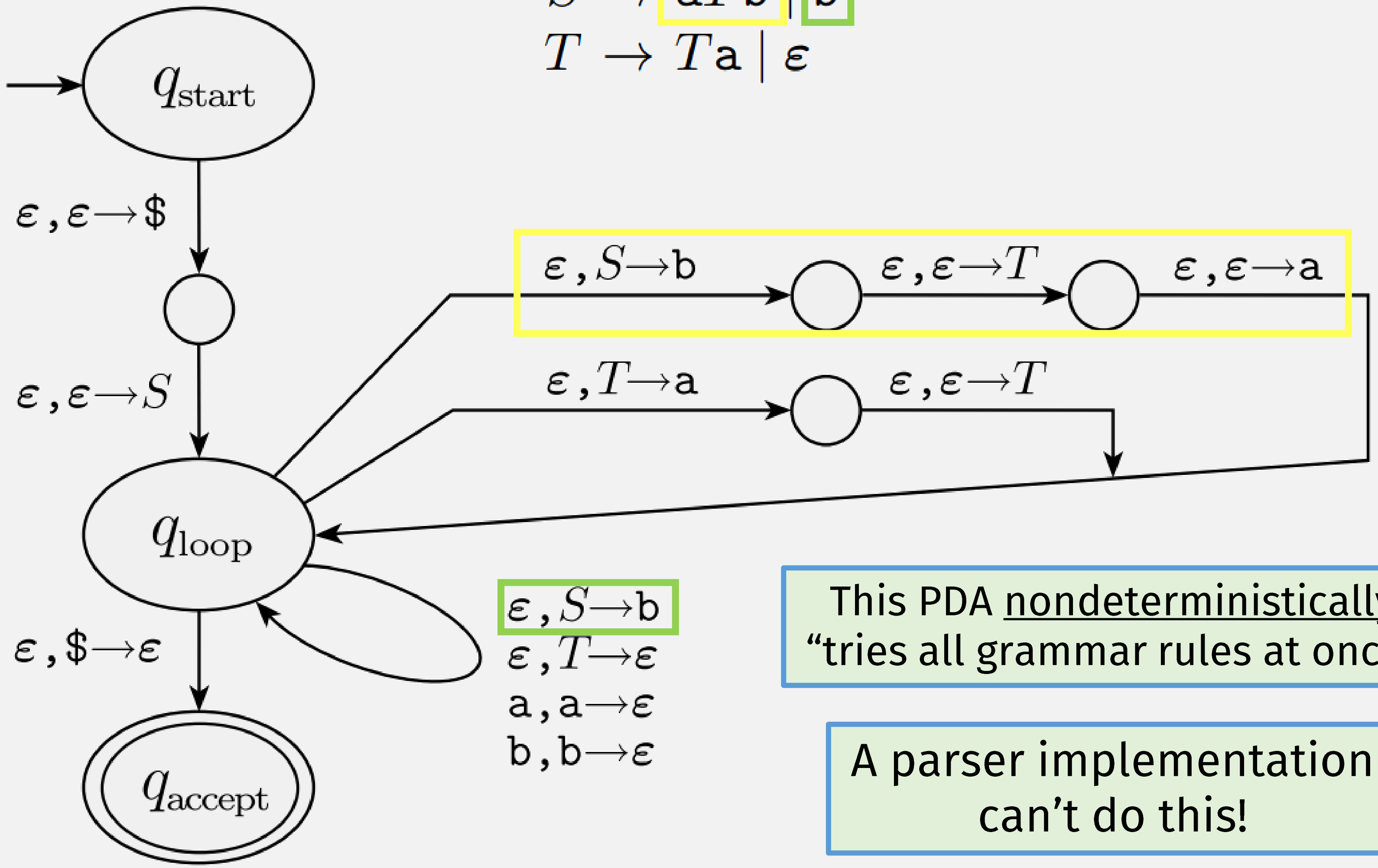
$$A \Rightarrow 0A1 \Rightarrow 00A11 \Rightarrow 000A111 \Rightarrow 000B111 \Rightarrow 000\#111$$

Generating vs Parsing

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

Last time: (Nondeterministic) PDA

$S \rightarrow aTb \mid b$
$T \rightarrow Ta \mid \varepsilon$



This PDA nondeterministically "tries all grammar rules at once"

A parser implementation can't do this!

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 $\delta(q, \mathrm{a}, X)$ is valid, then $\delta(q, \varepsilon, X)$ must not be

DPDAs are Not Equivalent to PDAs!

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

Should use S rule

Parsing is deriving reversed: start with string, end with parse tree

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

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

Should use T rule

$$\mathtt{aa}\underline{\mathtt{abb}}\mathtt{bbb} \rightarrowtail \mathtt{a}\underline{\mathtt{a}T\mathtt{bb}}\mathtt{bb}$$

Don't know which rule to use because we can't see rest of the input!

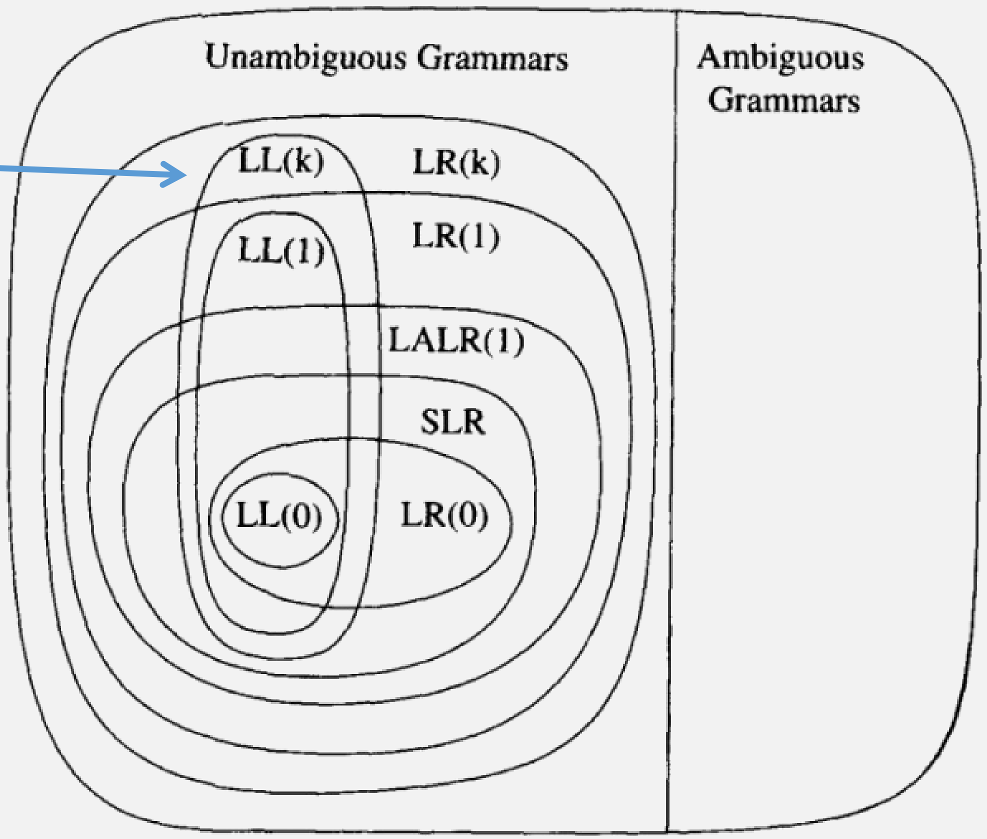
A PDA non-deterministically "tries all rules" (abandons failed attempts) but a DPDA gets only one try!

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

Subclasses of CFLs

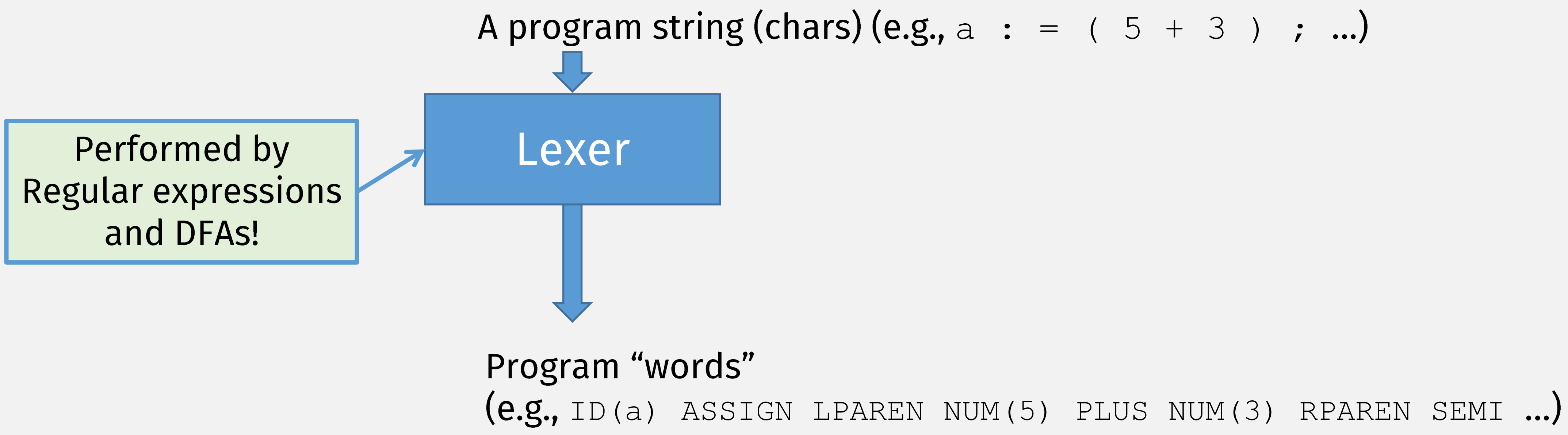
DCFLs

Programming language parsers / compilers are ideally in here



All CFLS

Compiler Stages



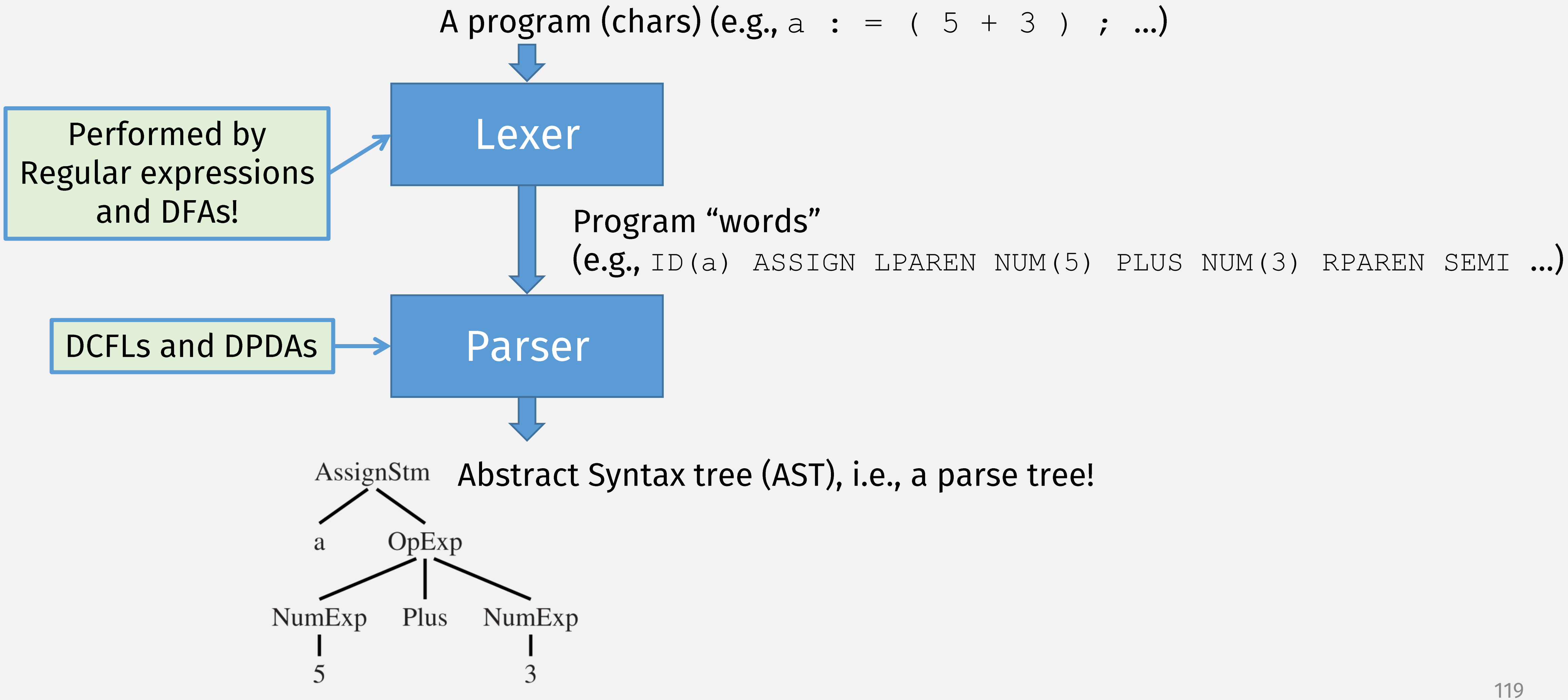
A Lexer Implementation

```
%{
/* C Declarations: */
#include "tokens.h"    /* definitions of IF, ID, NUM, ... */
#include "errormsg.h"
union {int ival; string sval; double fval;} yylval;
int charPos=1;
#define ADJ  (EM_tokPos=charPos, charPos+=yyleng)
%}
/* Lex Definitions: */
digits  [0-9]+
%%
/* Regular Expressions and Actions: */
if                       {ADJ; return IF;}
[a-z][a-z0-9]*           {ADJ; yylval.sval=String(yytext);
                           return ID;}
{digits}                 {ADJ; yylval.ival=atoi(yytext);
                           return NUM;}
({digits}"."[0-9]*)|([0-9]*"."{digits})     {ADJ;
                           yylval.fval=atof(yytext);
                           return REAL;}
("--"[a-z]*"\n")|(" "|"\n"|"\t")+    {ADJ;}
.                        {ADJ; EM_error("illegal character");}
```

Regular expressions!

A "`lex`" tool translates this to a (C program) implementation of a lexer

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

$$
\begin{aligned}
R &\to S \mid T \\
S &\to \mathrm{a}S\mathrm{b} \mid \mathrm{ab} \\
T &\to \mathrm{a}T\mathrm{bb} \mid \mathrm{abb}
\end{aligned}
$$

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

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

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

LL parsing

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

"You're the Parser" Game:
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 $S \rightarrow \text{if } E \text{ then } S \text{ else } S$

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
```

LL parsing

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

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

2 $S \to$ begin S L

3 $S \to$ print E

4 $L \to$ end

5 $L \to$; S L

6 $E \to$ num = num

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

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

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 a plus

Need to save input somewhere, like a stack: this is a job for a (D)PDA!!

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 $\text{id}_4\ :=_6$	7 ; b := c + (d := 5 + 6 , d) $	*shift*
1 $\text{id}_4\ :=_6\ \text{num}_{10}$	; b := c + (d := 5 + 6 , d) $	*reduce* $E \rightarrow \text{num}$
1 $\text{id}_4\ :=_6\ E_{11}$	; 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

$$
\begin{array}{ll}
S \to S \,;\, S & E \to \text{id} \\
S \to \text{id} := E & E \to \text{num} \\
S \to \text{print} \,(\, L \,)\, & E \to E + E
\end{array}
$$

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* $E \to$ 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 $E \rightarrow E + E$

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* $E \rightarrow$ 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 $S \rightarrow S \,;\, S$
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$_4$:=$_6$	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*

Can determine (rightmost) rule

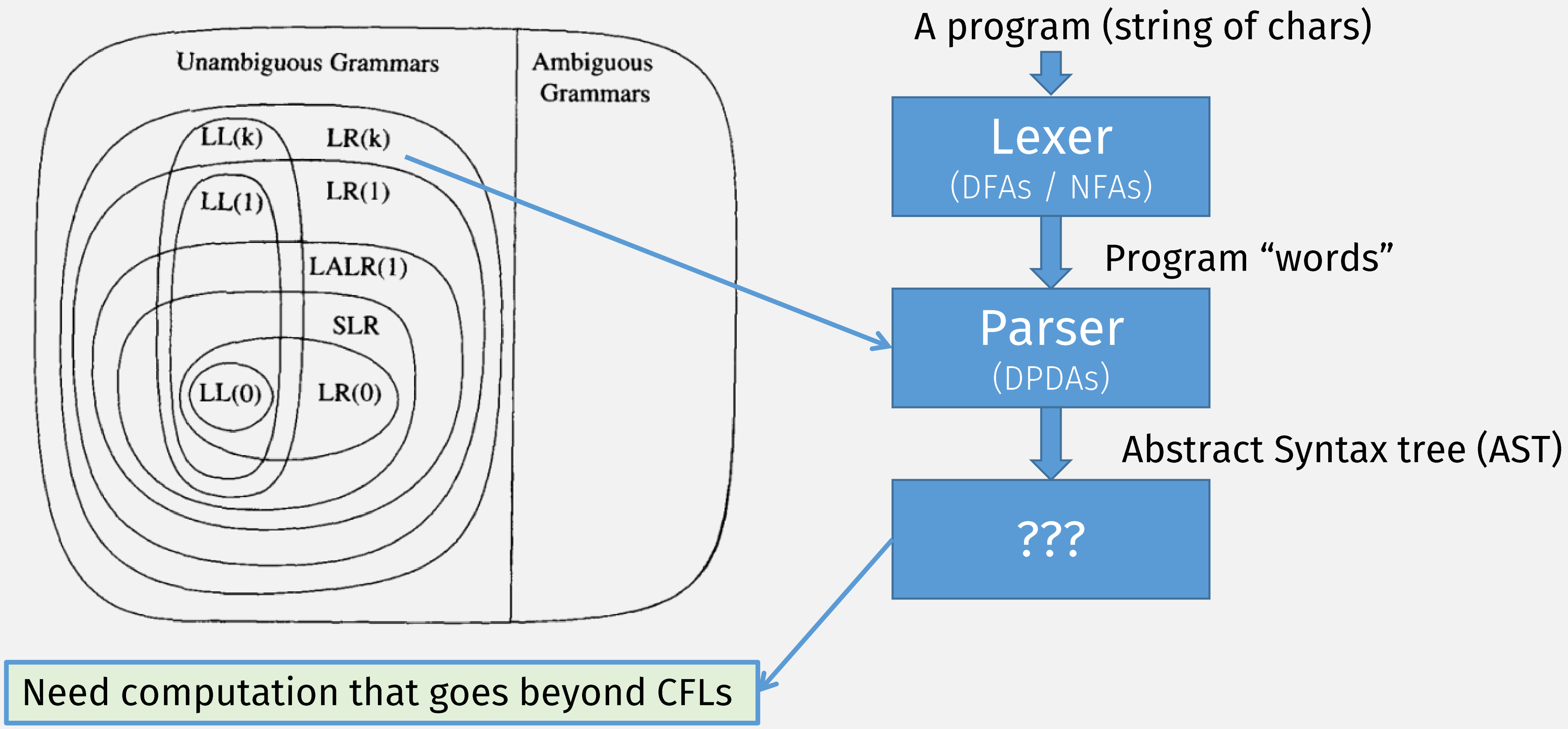
LR parsing

- 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!

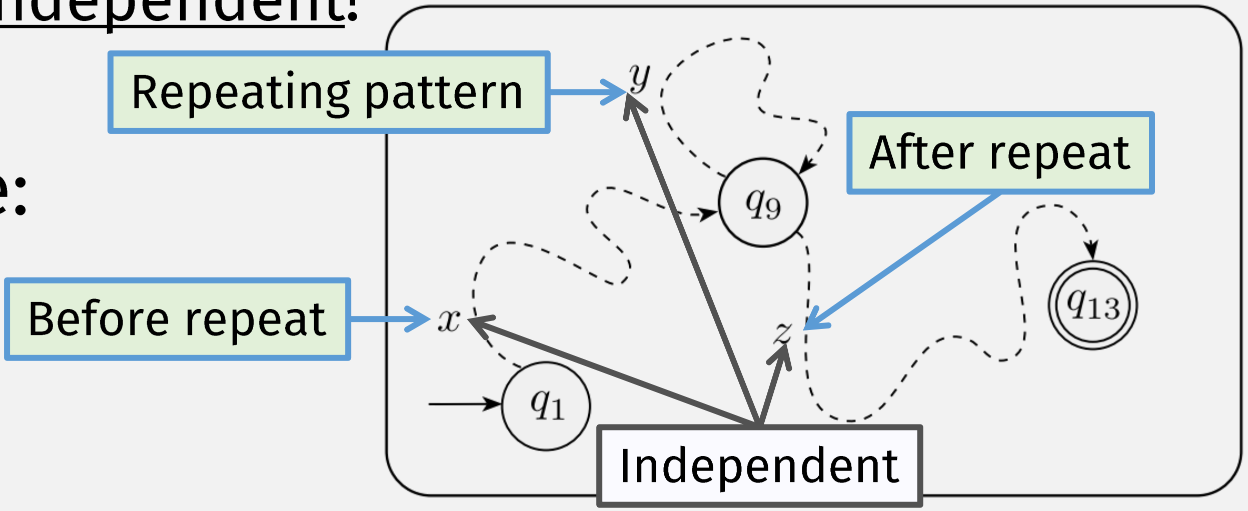


Non-CFLs

Flashback: Pumping Lemma for Reg Langs

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

- Regular language strings can (only) repeat using Kleene pattern
  - But the <u>substrings are independent</u>!



- A non-regular language:

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

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

- How do CFLs repeat?

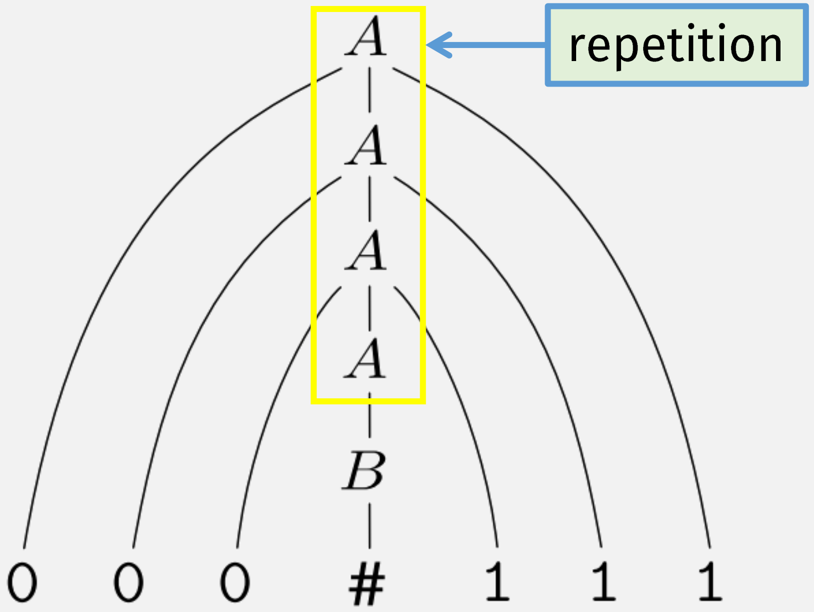
Repetition and Dependency in CFLs

Parts before/after repetition point are linked

Repetition

$$A \rightarrow 0A1$$
$$A \rightarrow B$$
$$B \rightarrow \#$$

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

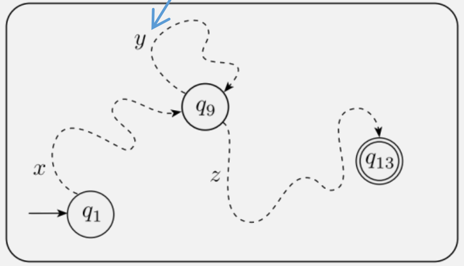


$$A \Rightarrow 0A1 \Rightarrow 00A11 \Rightarrow 000A111 \Rightarrow 000B111 \Rightarrow 000\#111$$

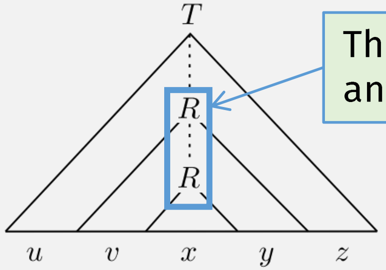
How Do Strings in CFLs Repeat?

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

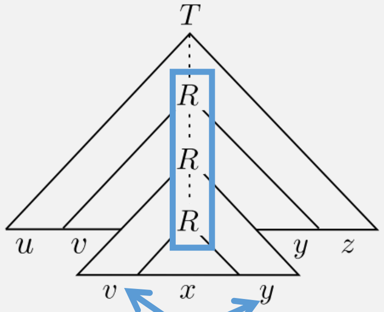
- Strings in regular languages repeat states

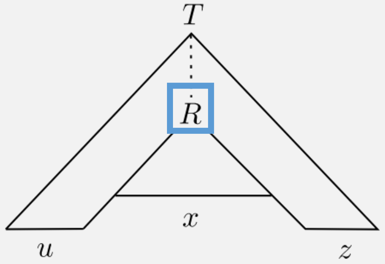


- Strings in CFLs <u>repeat subtrees</u> in the parse tree



This subtree can be repeated any number of times





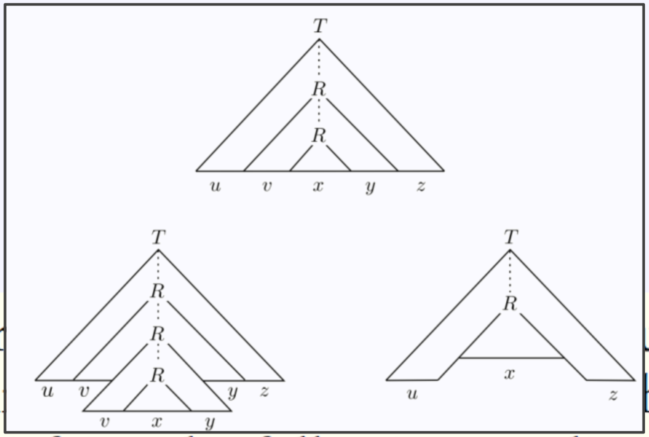
Linked parts

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 two pumpable parts.
But they must be pumped together!

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 ... umber p (the pumping length) where if s is any stri... 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. $|xy| \leq p$.

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

Previously: D is nonregular: unpumpable counterexample s: 0^p10^p1

Now: this s **can** be pumped according to CFL pumping lemma:

$$\overbrace{\underbrace{000\cdots000}_{u}\ \underbrace{0}_{v}\ \underbrace{1}_{x}}^{0^p1}\ \overbrace{\underbrace{0}_{y}\ \underbrace{000\cdots0001}_{z}}^{0^p1}$$

Pumping v and y (together) produces string still in D

- 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 the attempt to prove that the language is not a CFL failed.

Non CFL example: $D = \{ww|\ w \in \{0,1\}^*\}$

- Choose another 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. $|vxy| \leq p$.

This language is not a CFL!

CFL Pumping Lemma is Weird?

???

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. $|vxy| \leq p$.

Review: Regular Language Pumping Lemma

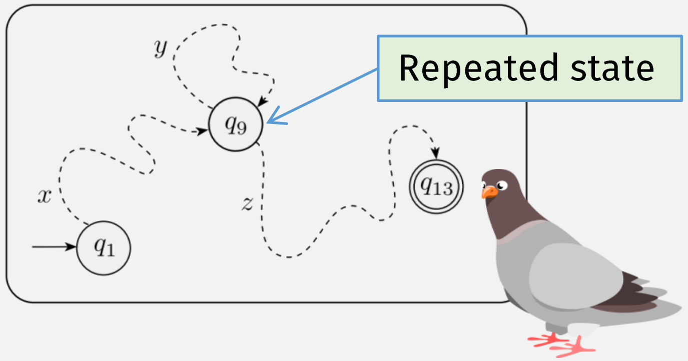
- The pumping length p for a language L is …

  … the # of states in that language's NFA!

Pumping lemma If A is a regular 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 three pieces, $s = xyz$, satisfying the following conditions:

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

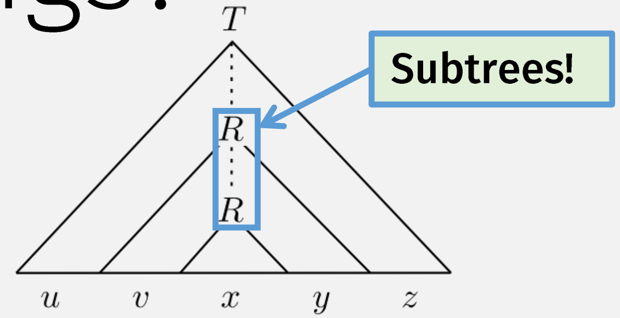
- If string length > # of states, then some state must repeat



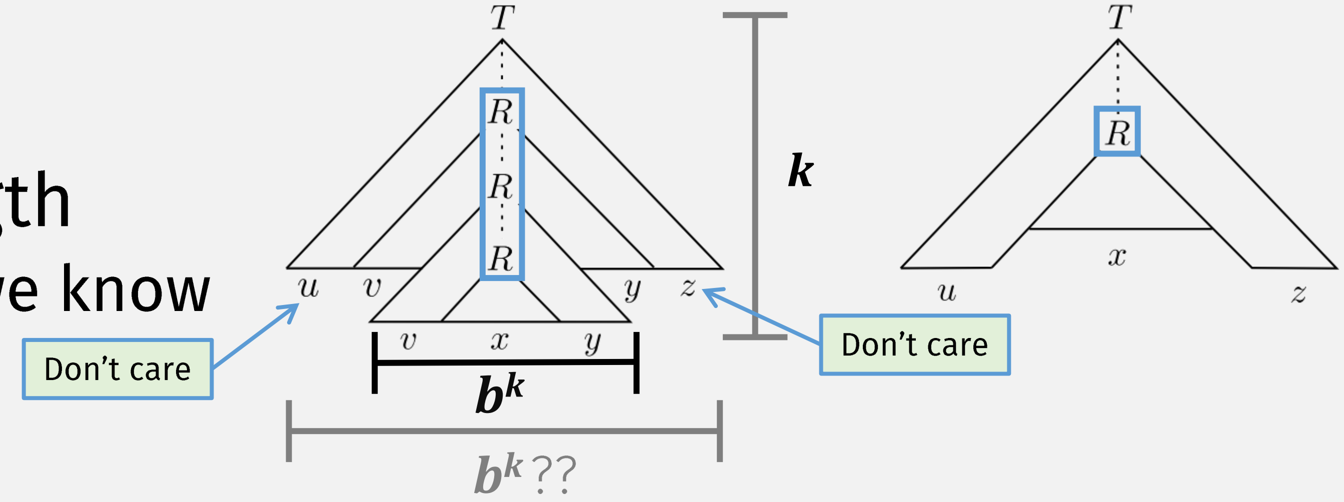
- If a state is repeated once, then it can repeat multiple times

Repeating Pattern in CFL Strings?

- When are we <u>guaranteed</u> to have a repeated subtree?
  - When <u>height</u> of parse tree > # of rules!



- Let $\boldsymbol{k}$ = # of rules and $\boldsymbol{b}$ = longest rule RHS length
  - Then the length string where we know there's a repeat is $\boldsymbol{b^k}$
  - I.e., pumping length = $\boldsymbol{b^k}$???



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. $|vxy| \leq p$.

Pumping Length could be too short!

A Pumpable Non-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. $|vxy| \leq p$.

- CFL Pumping Lemma says:
  - *"All CFLs are pumpable"*
  - So if we find a non-pumpable language … it's not a CFL!
- Pumping Lemma does not say:
  - *"All nonCFLs are not pumpable"*
  - (statement != it's inverse)
  - So Pumping Lemma might not be able to prove some non-CFLs!

Example:

$$L = \{\mathtt{a}^i\mathtt{b}^j\mathtt{c}^k\mathtt{d}^l \mid i = 0 \text{ or } j = k = l\}$$

- For any counterexample, split into *uvxyz* where,
  - v = first char
  - z = remaining chars
  - $u = x = y = \varepsilon$
- If there are **a**s …
  - … it's pumpable bc # of **a**s is arbitrary
- There there are no **a**s
  - … it's pumpable bc # of other chars is arbitrary

This language is pumpable … but not a CFL!
(can't come up with a CFG)

Ogden's Lemma (generalizes pumping lemma)

Ogden's lemma is: If L is a CFL, then there is a constant n, such that if z is any string of length at least n in L, in which we select at least n positions to be *distinguished*, then we can write $z = uvwxy$, such that:

1. vwx has at most n distinguished positions.
2. vx has at least one distinguished position.
3. For all i, uv^iwx^iy is in L.

Says that <u>every</u> long enough segment must be pumpable

<u>Example</u>:

$$L = \{\mathtt{a}^i\mathtt{b}^j\mathtt{c}^k\mathtt{d}^l \mid i = 0 \text{ or } j = k = l\}$$

This language is not a CFL because it doesn't satisfy Ogden's Lemma

<u>Counterexample</u>: $\mathtt{ab}^n\mathtt{c}^n\mathtt{d}^n$

- n "distinguished" positions must include non-$\mathtt{a}$ character
  - Impossible to pump no matter which n chars are chosen

A Practical Non-CFL

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

- XML also looks like this <u>non-CFL</u>: $D = \{ww|\ w \in \{0,1\}^*\}$

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

- <u>In practice</u>:
  - XML is <u>parsed</u> 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 10/6

See gradescope