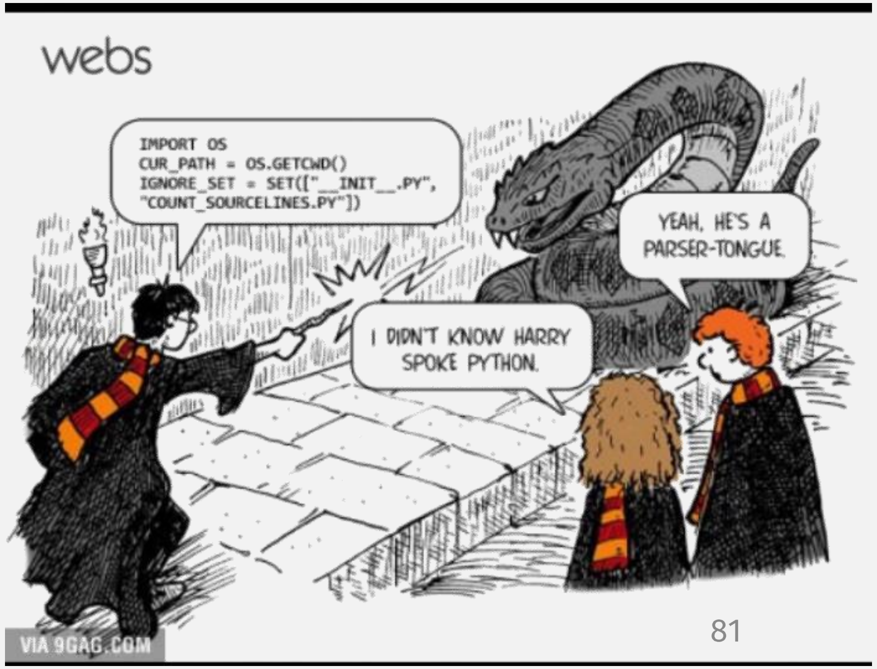
Deterministic CFLs, Deterministic PDAs, and Parsing

Monday, March 8, 2021

Announcements

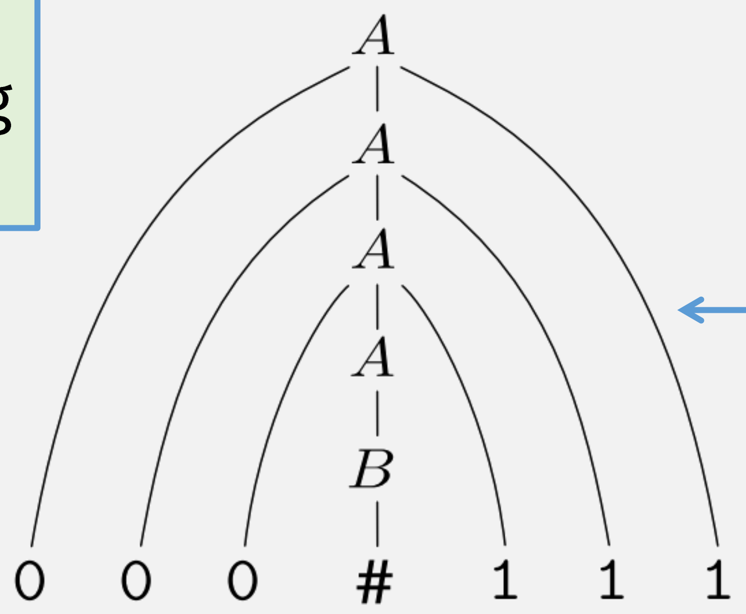
- Reminder: no class next week (Spring Break)
  - 3/15 – 3/19
- HW5 due Wed 3/10 11:59pm EST
- HW6 released soon
  - Due after break



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

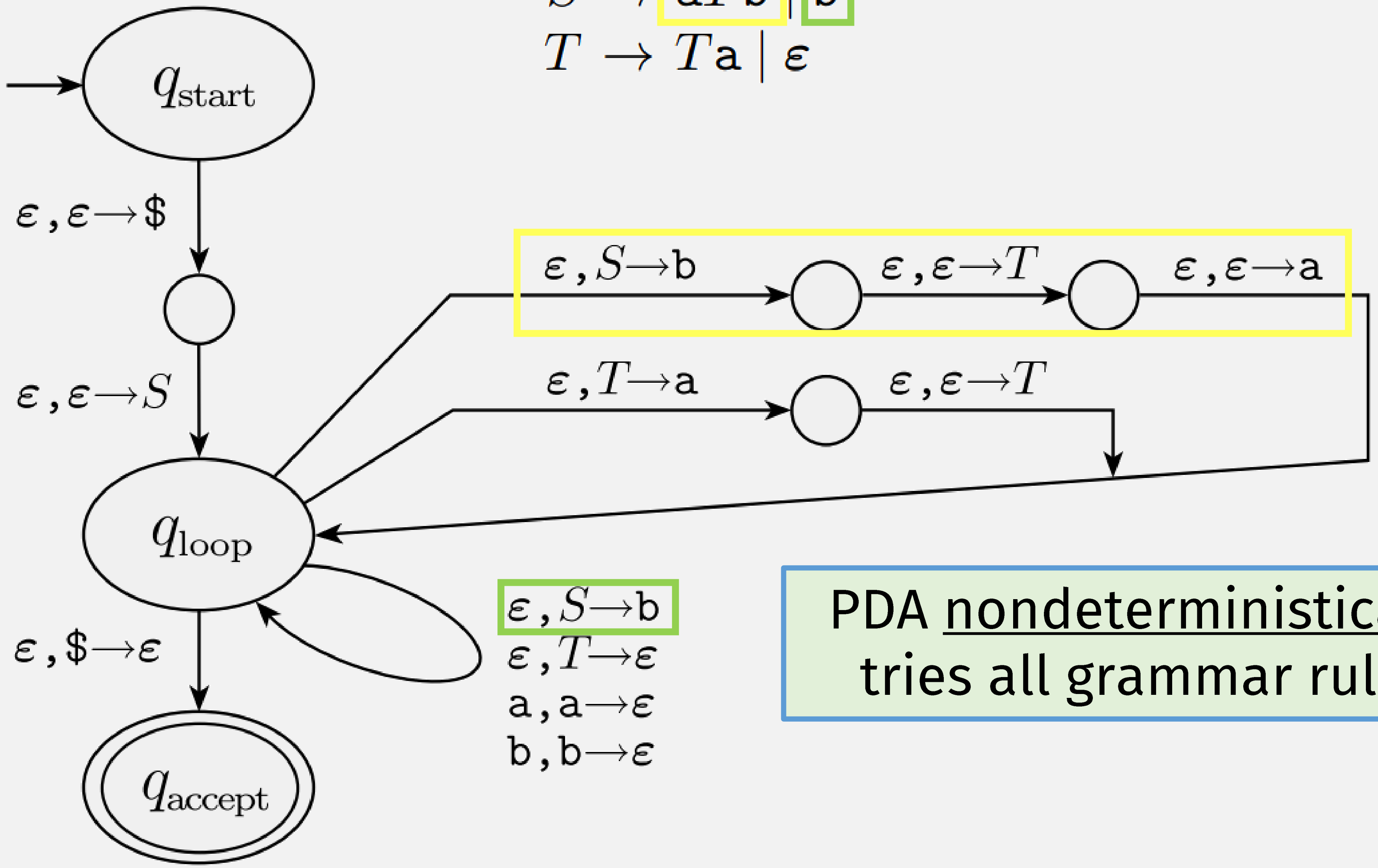
$$A \Rightarrow 0A1 \Rightarrow 00A11 \Rightarrow 000A111 \Rightarrow 000B111 \Rightarrow 000\#111$$

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: (Nondeterministic) PDA

$S \to \texttt{a}T\texttt{b} \mid \texttt{b}$
$T \to T\texttt{a} \mid \varepsilon$



PDA nondeterministically tries all grammar rules

DPDA: Formal Definition

DEFINITION 2.39 — 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.

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

DPDAs are Not Equivalent to PDAs!

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

Follows S rule

Parsing is reverse of deriving

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

When parsing reaches this input position, don't know whether to use rule S or T, because we can't see rest of string

Follows T rule

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

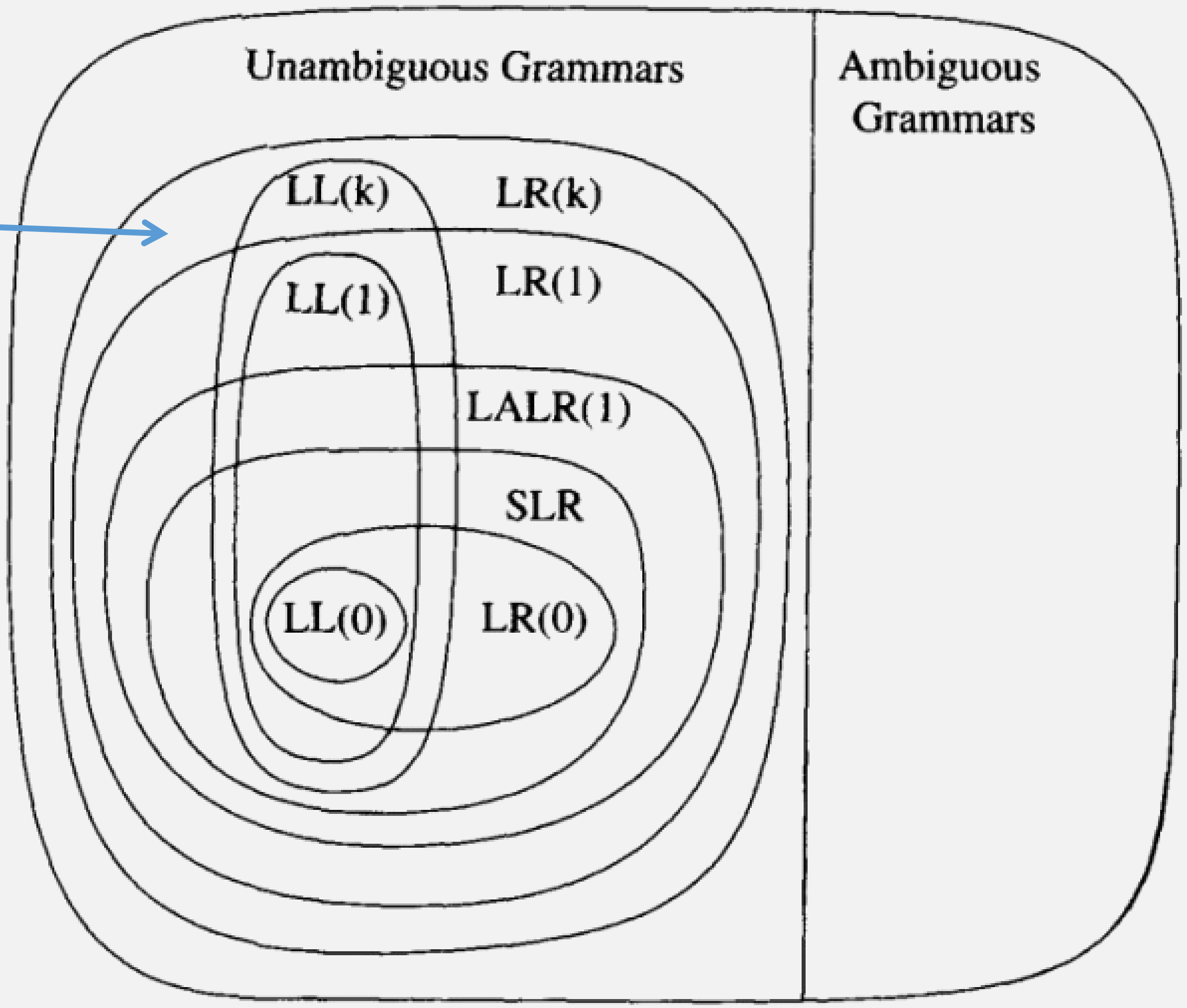
A PDA can non-deterministically "try all rules", but a DPDA must choose one

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

Subclasses of CFLs

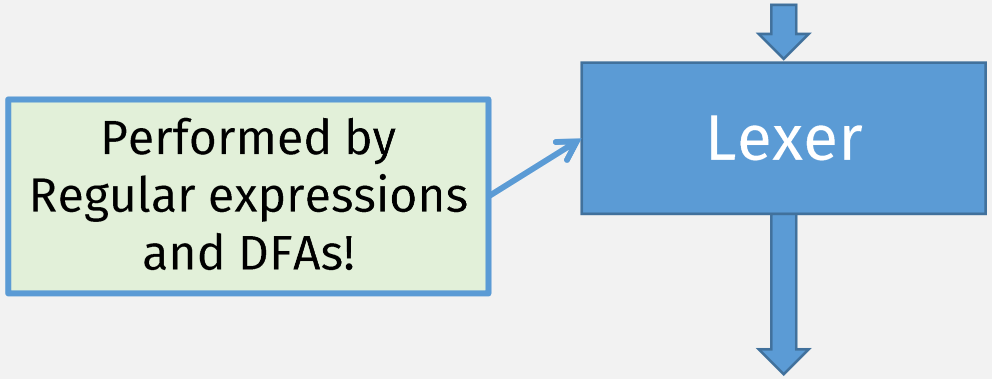
DCFLs

Programming language parsers / compilers are usually in here



Compiler Stages

A program (chars) (e.g., `a := ( 5 + 3 ) ;` ...)



Program “words”
(e.g., `ID(a) ASSIGN LPAREN NUM(5) PLUS NUM(3) RPAREN SEMI` ...)

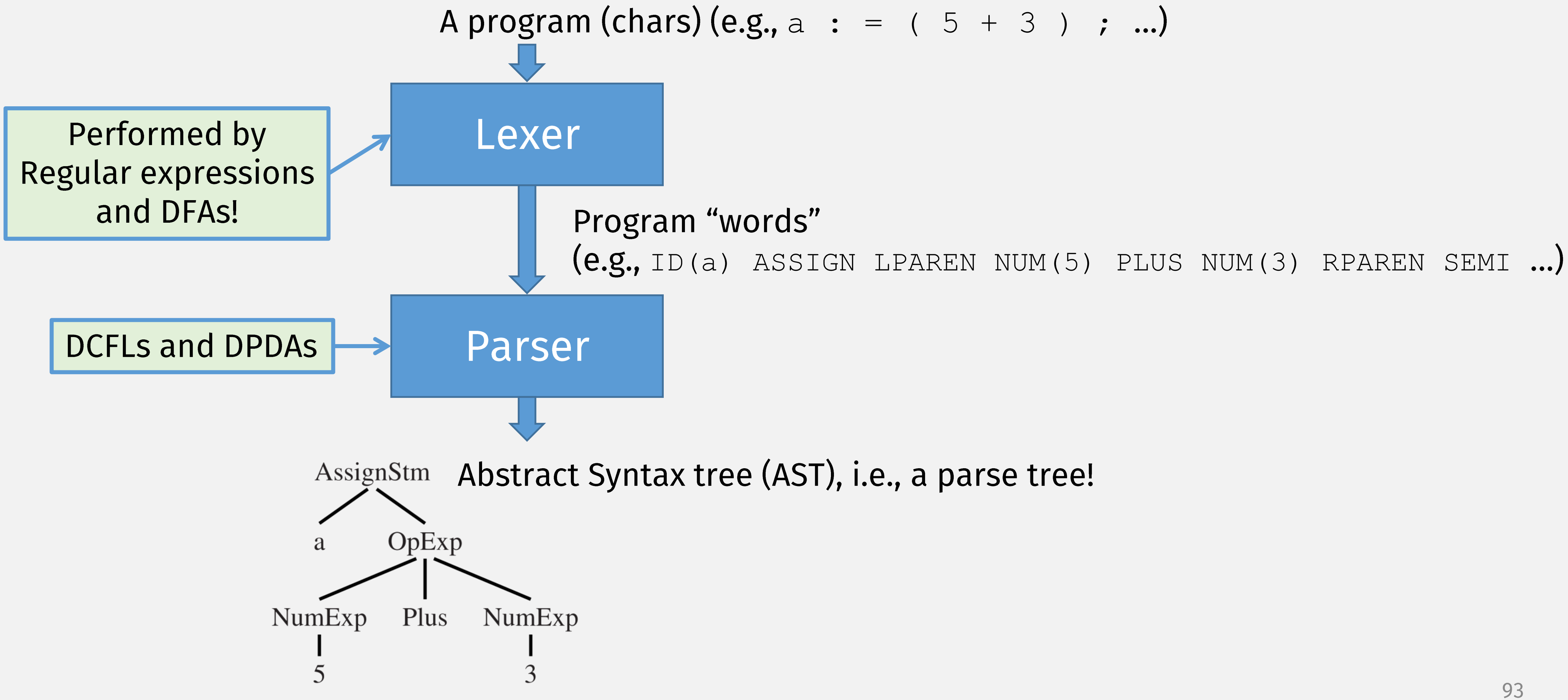
A Lexer Implementation

```
%{
/* 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");}
```

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

Regular expressions!

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

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

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

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

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

LL parsing

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

$S \rightarrow$ if E then S else S
$S \rightarrow$ begin S L
$S \rightarrow$ print E

$L \rightarrow$ end
$L \rightarrow$; S L

$E \rightarrow$ num $=$ 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

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

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

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

```
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

LR parsing

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

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

```
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; i.e., 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*
$_1$ 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

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

```
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; i.e., this is a job for a (D)PDA!!

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

LR parsing

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

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

```
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; i.e., 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*
$_1$ 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

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

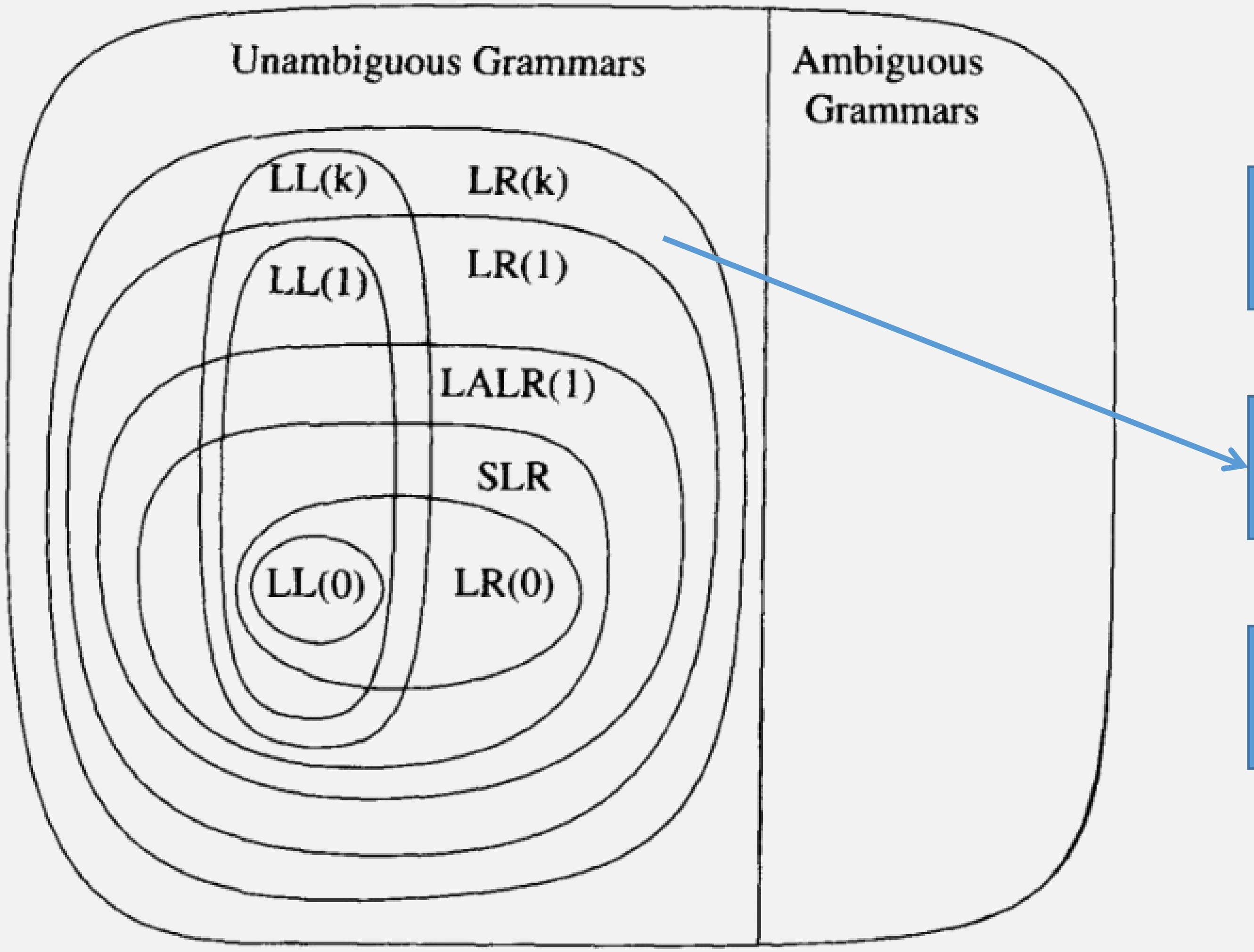
```
a := 7;
b := c + (d := 5 + 6, d)
```

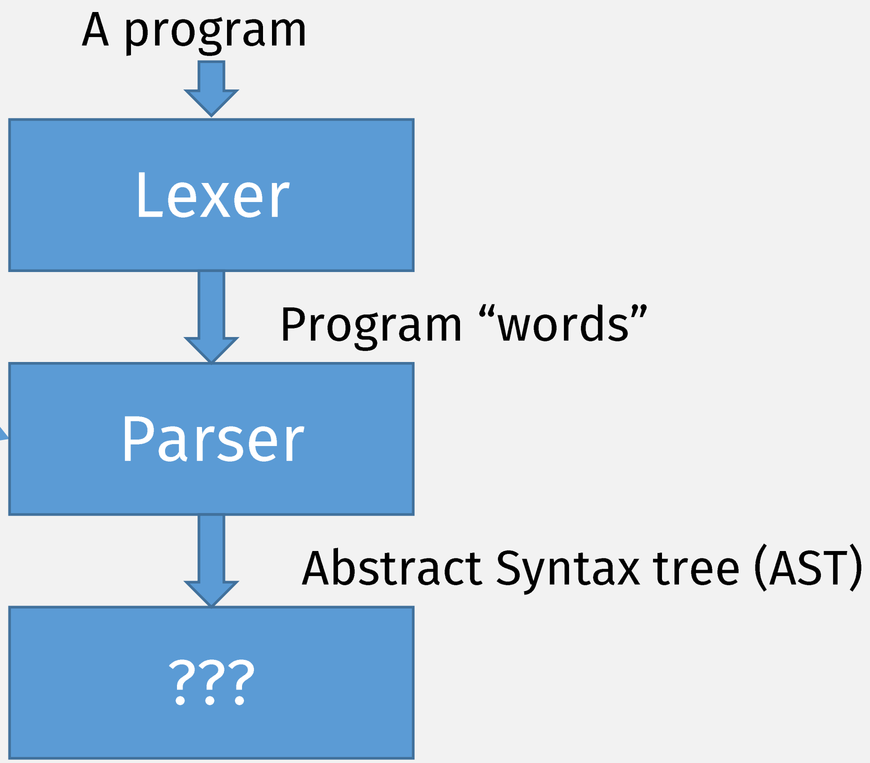
When parse is here, can't determine whether it's an assign or a plus

Need to save input somewhere, like a stack; i.e., 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*
$_1$ 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*

To learn more, take a Compilers Class!





Next time: Pumping Lemma for CFLs

- Pumping Lemma for reg langs identifiers non-regular langs
- How do we know when a language is not a CFL?
- The Pumping Lemma for CFLs!

Check-in Quiz 3/8

On Gradescope