Deterministic CFLs, DPDAs, and Parsing

Wed, October 14, 2020

HW4 Questions?

HW3 Presentations

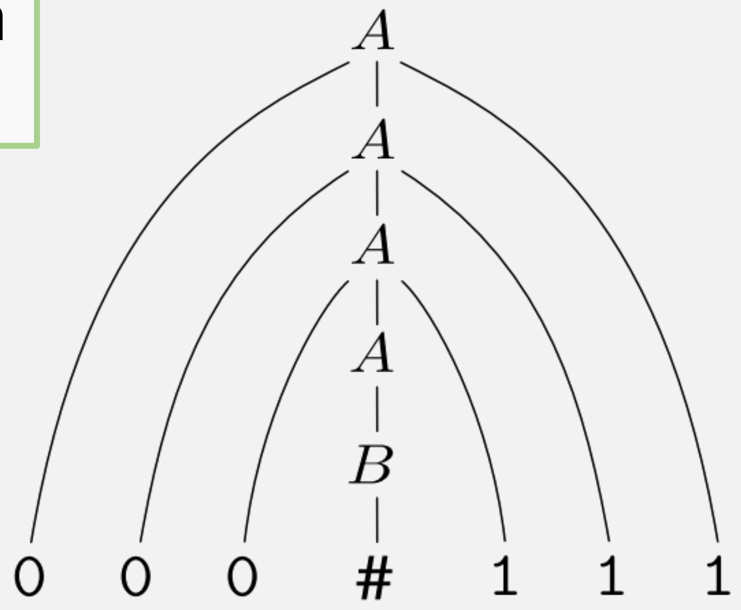
Yash/Scott (Java)

Luke (Python)

Previously: CFLs, CFGs, and Parse Trees

Generating a string creates parse tree from the start variable

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



In practice, the opposite is more interesting: **parsing** a string into parse tree

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

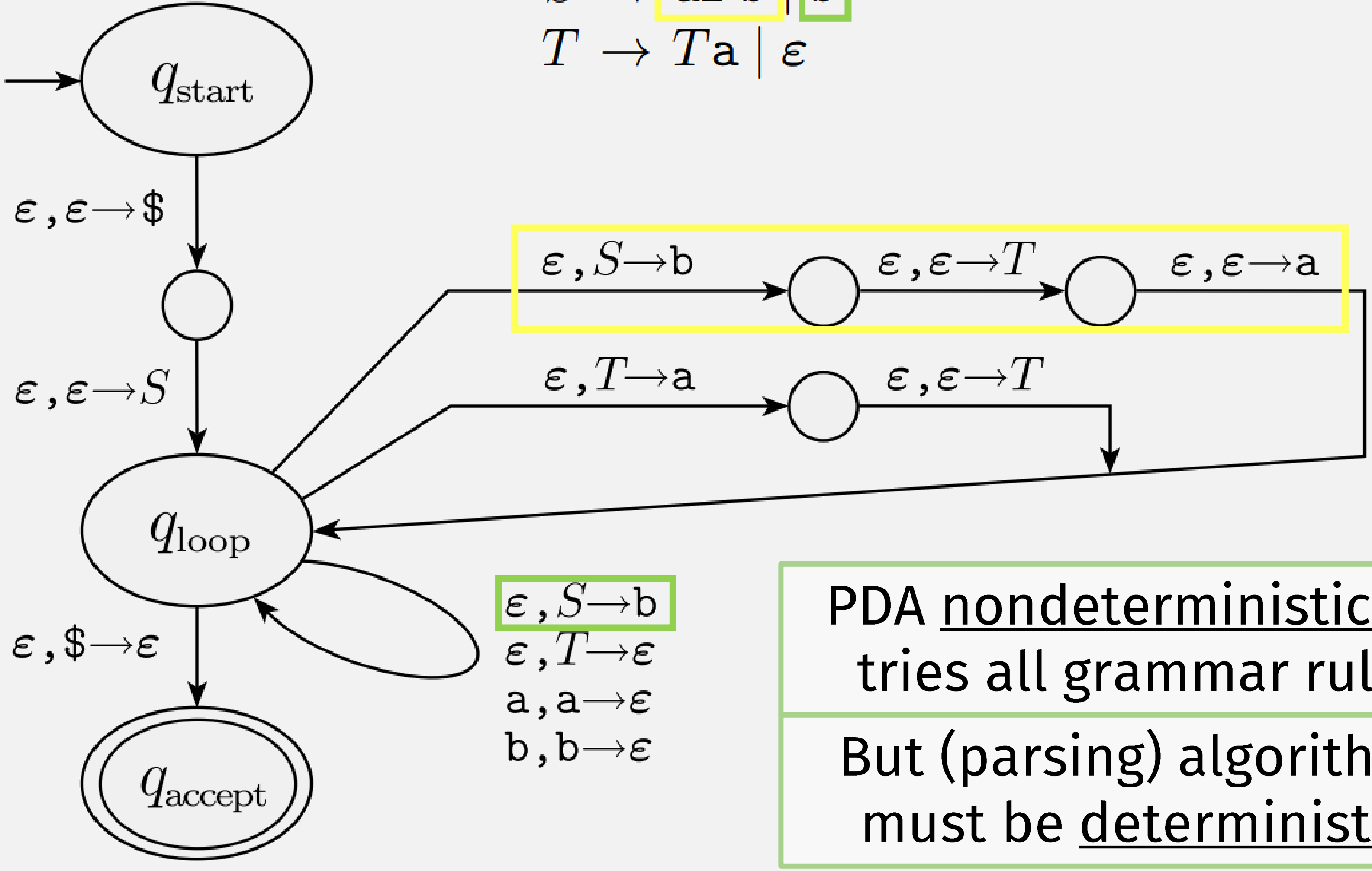
Generating vs Parsing

- Parsing is practically more interesting
  - E.g., an algorithm for parsing source code

- But we don’t have a machine that can do it yet.

Last time: Nondeterministic PDA

$$S \to \boxed{\mathtt{a}T\mathtt{b}} \mid \boxed{\mathtt{b}}$$
$$T \to T\mathtt{a} \mid \varepsilon$$



q_{start}
$\varepsilon,\varepsilon\to\$$
$\varepsilon,\varepsilon\to S$
q_{loop}
$\varepsilon,S\to\mathtt{b}$
$\varepsilon,\varepsilon\to T$
$\varepsilon,\varepsilon\to\mathtt{a}$
$\varepsilon,T\to\mathtt{a}$
$\varepsilon,\varepsilon\to T$
$\varepsilon,\$\to\varepsilon$
$\boxed{\varepsilon,S\to\mathtt{b}}$
$\varepsilon,T\to\varepsilon$
$\mathtt{a},\mathtt{a}\to\varepsilon$
$\mathtt{b},\mathtt{b}\to\varepsilon$
q_{accept}

PDA nondeterministically tries all grammar rules

But (parsing) algorithms must be deterministic!

Generating vs Parsing

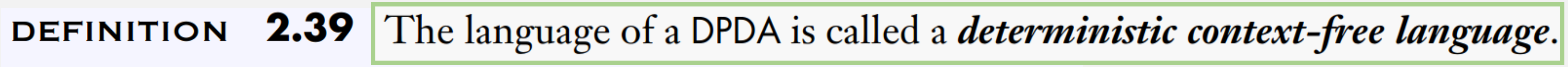
- Parsing is practically more interesting
  - E.g., an algorithm for parsing source code

- But we don’t have a machine that can do it yet.

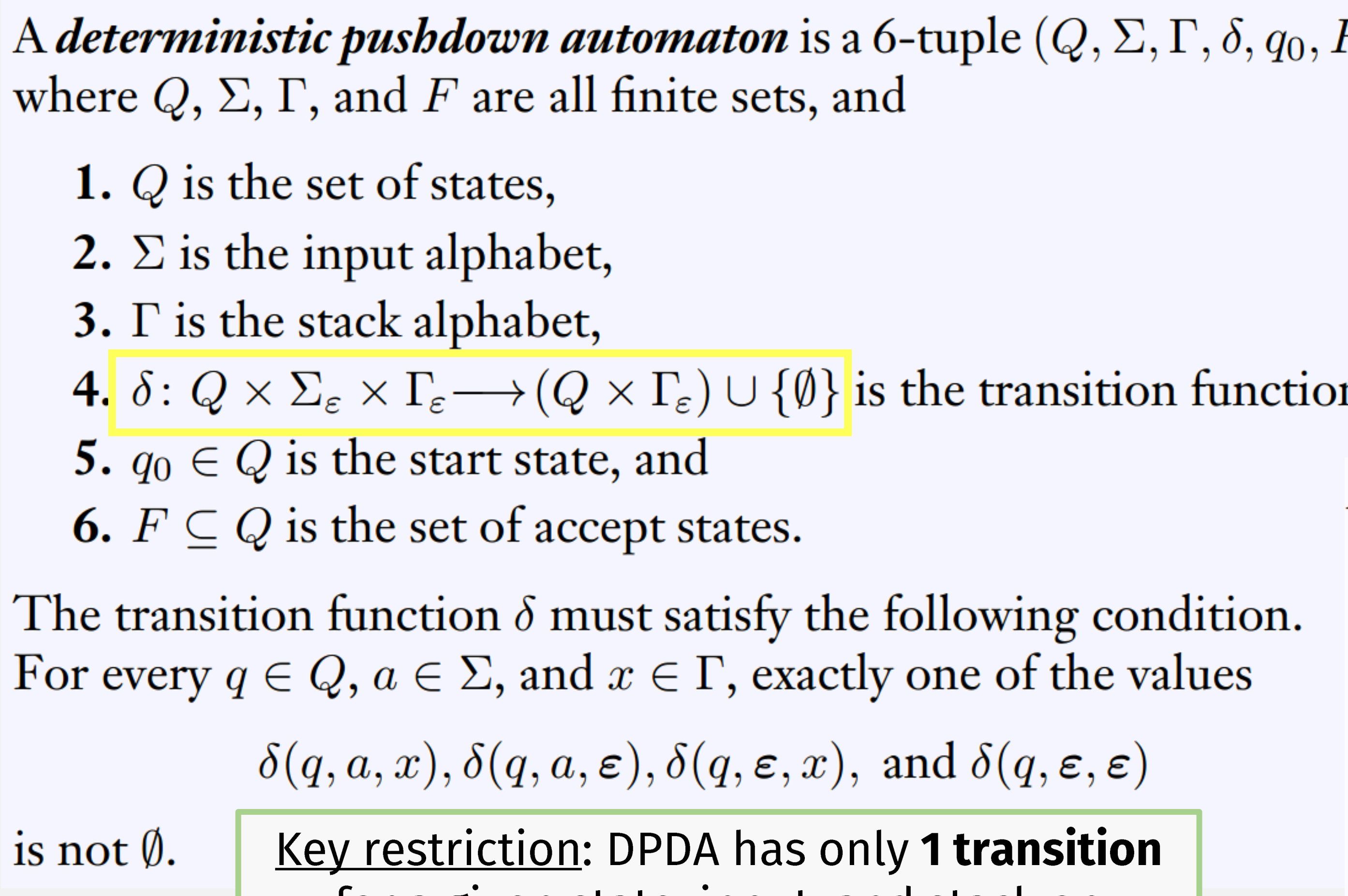
- PDAs are non-deterministic, like NFAs
  - But algorithms must be deterministic

- Need a **Deterministic** PDA (DPDA)

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

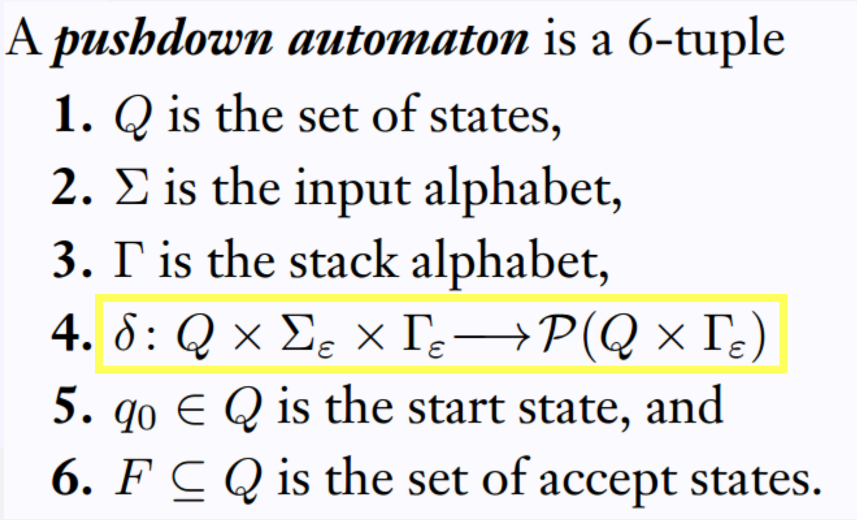
1. Q is the set of states,
2. Σ is the input alphabet,
3. Γ is the stack alphabet,
4. $\delta\colon Q \times \Sigma_\varepsilon \times \Gamma_\varepsilon \longrightarrow (Q \times \Gamma_\varepsilon) \cup \{\emptyset\}$ is the transition function,
5. $q_0 \in Q$ is the start state, and
6. $F \subseteq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

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

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

is not $\emptyset$.

Key restriction: DPDA has only **1 transition** for a given state, input, and stack op



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!

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

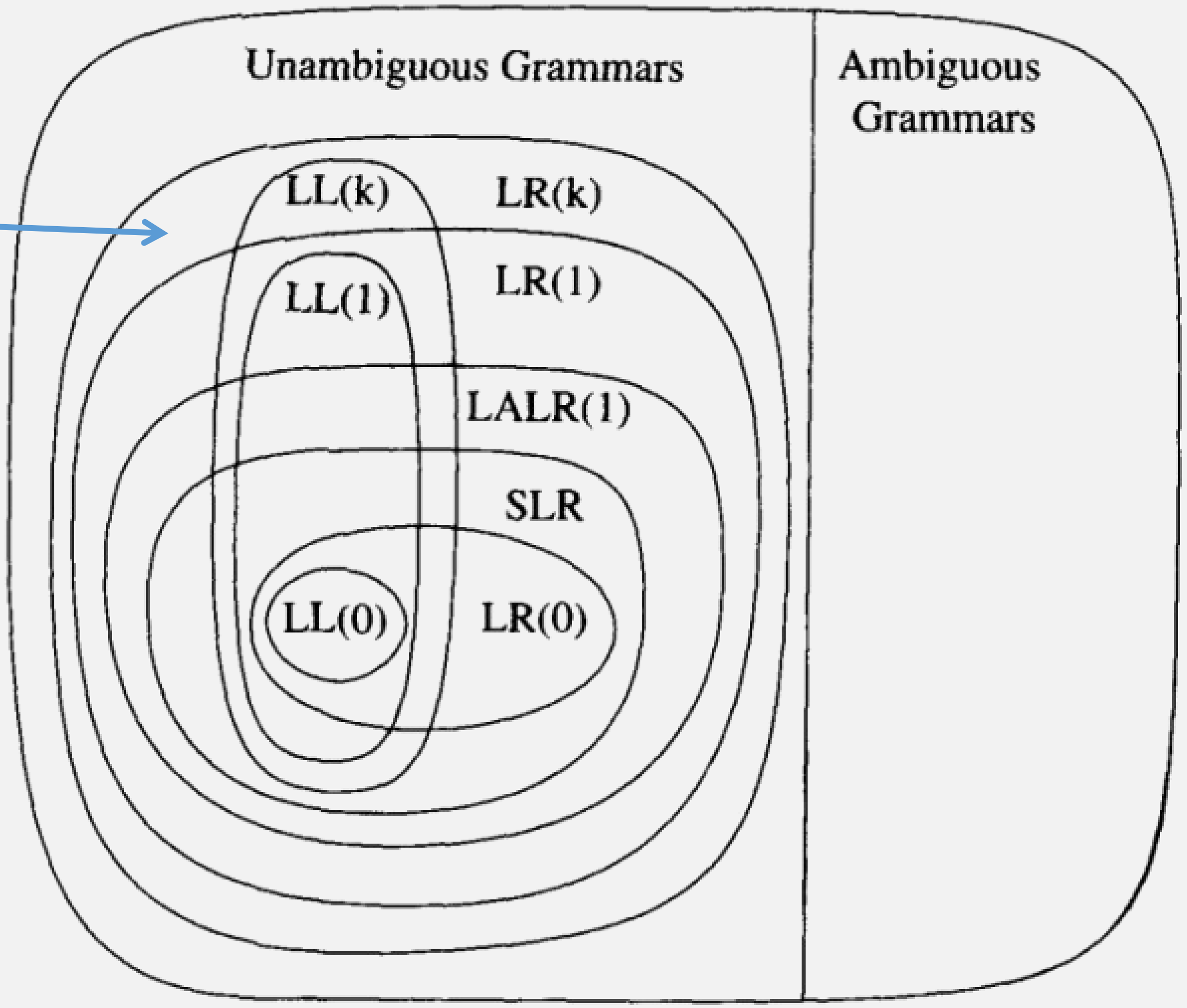
$$\mathrm{aa\underline{ab}bb} \rightarrowtail \mathrm{a\underline{aSb}b}$$

At this input char, PDA can non-deterministically “try all rules”, but a DPDA must guess one

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

PDAs recognize CFGs, but DPDA can only recognize a subset of CFGs, DCFGs!

Subclasses of CFLs

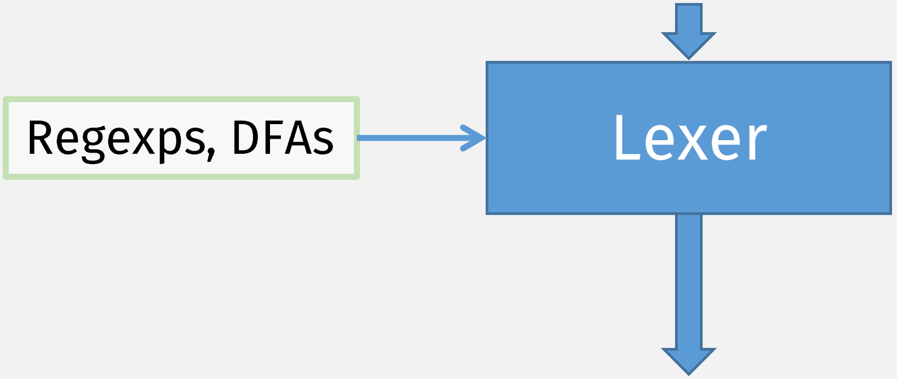


DCFGs

PLs typically in here

Compiler Stages

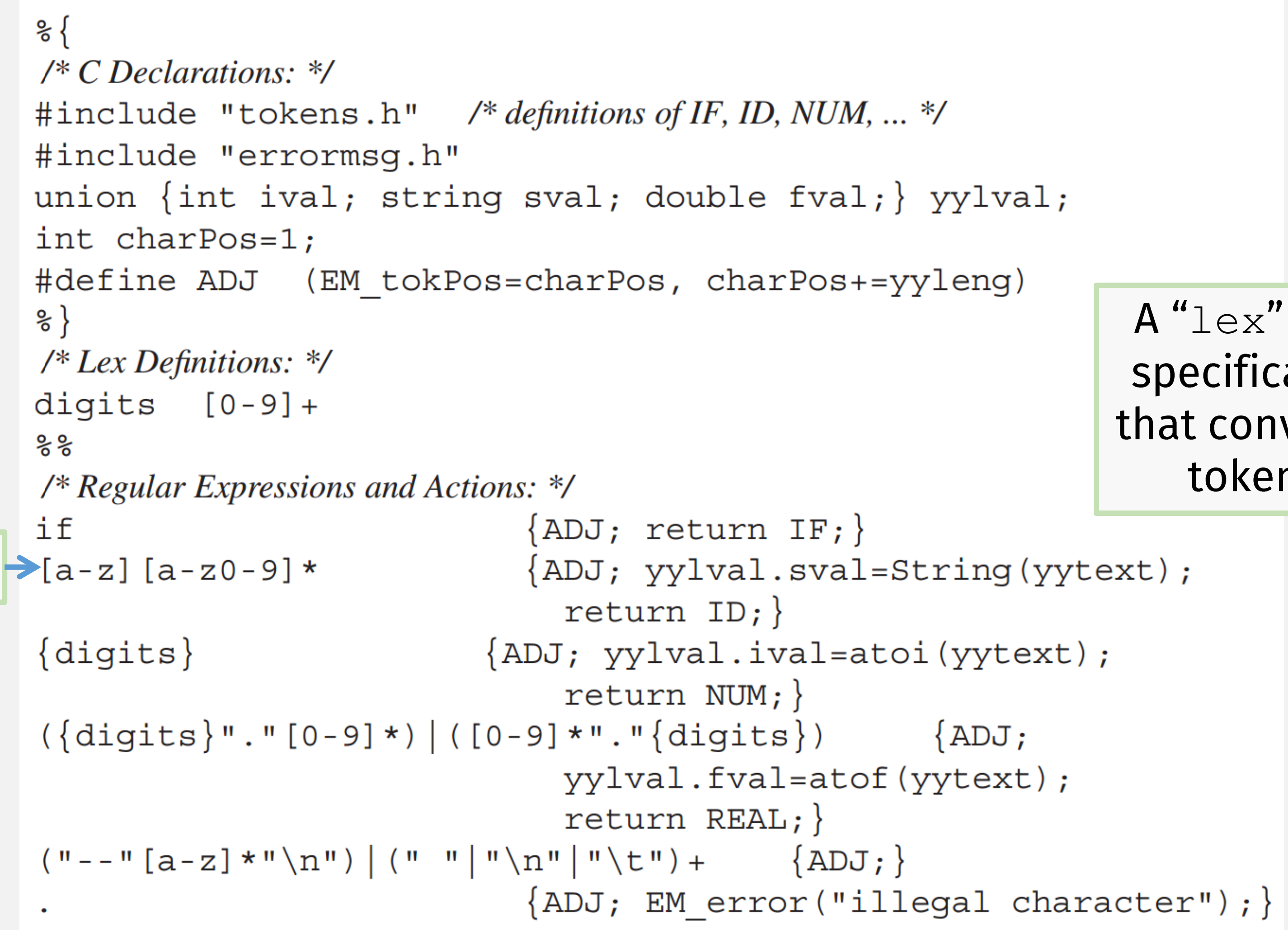
A program (chars) (e.g., `a : = ( 5 + 3 ) ;` ...)



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

A Lexer Specification

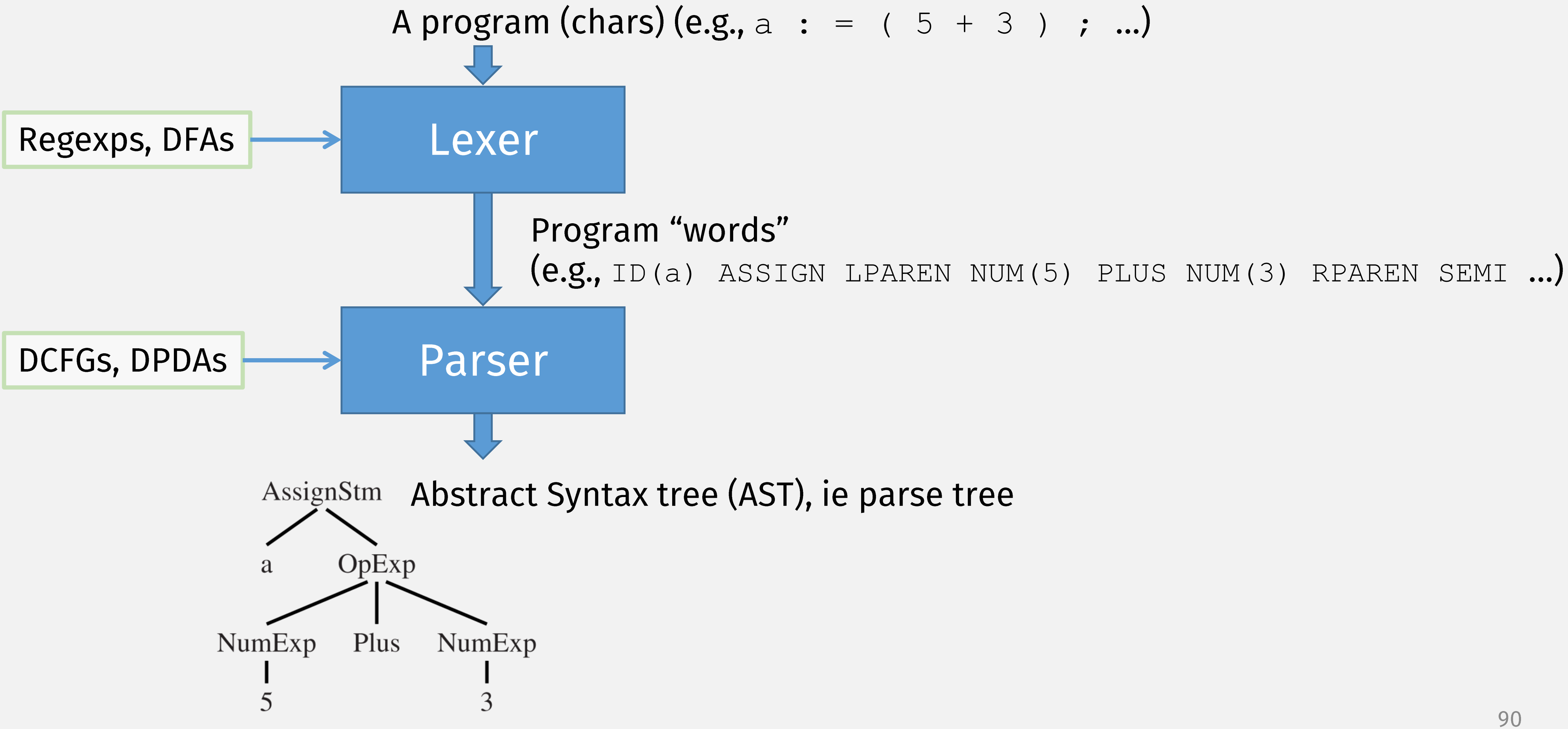
Just write Regexps



```
%{
/* 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 compiles this specification to a program that converts programs into tokens (i.e., “words”)

Compiler Stages



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

Regexps, DFAs

Lexer

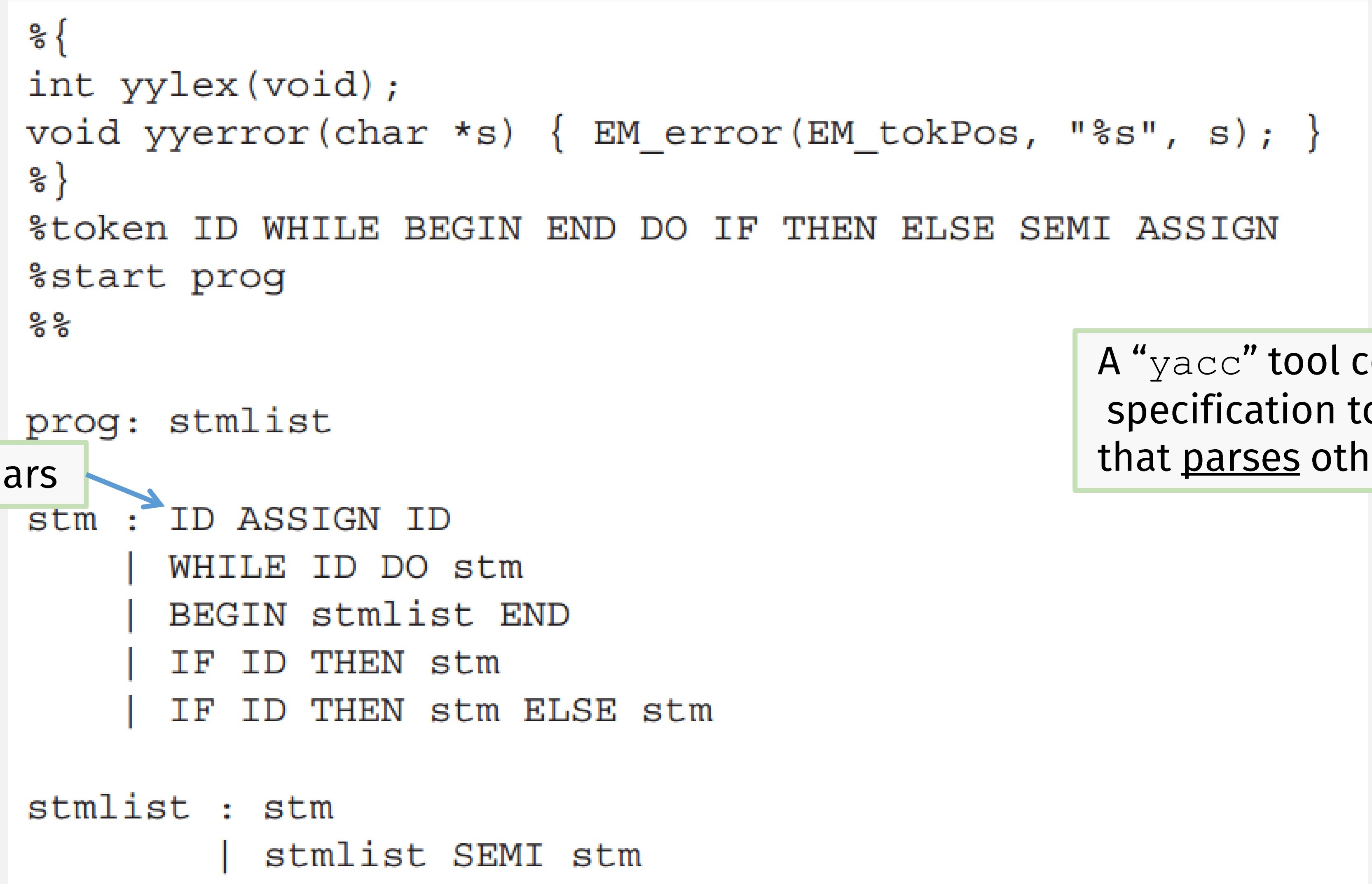
Program “words”
(e.g., `ID(a) ASSIGN LPAREN NUM(5) PLUS NUM(3) RPAREN SEMI` ...)

DCFGs, DPDAs

Parser

Abstract Syntax tree (AST), ie parse tree

A Parser Specification



```
%{
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 Grammars

A "yacc" tool compiles this specification to a program that parses other programs

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

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

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
↑
```

LL parsing

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

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

$$
\begin{aligned}
L &\rightarrow \text{end} \\
L &\rightarrow \ ;\ S\ L \\
\\
E &\rightarrow \text{num} = \text{num}
\end{aligned}
$$

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

LL parsing

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

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

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

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

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

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

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

```
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

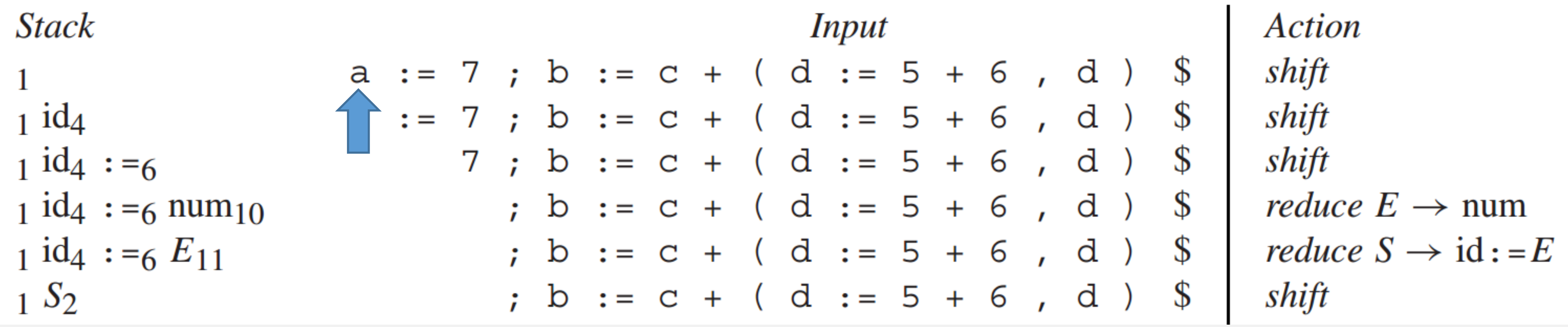
- L = left-to-right
- R = rightmost derivation

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

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

When parse is here, cant determine whether it’s an assign or a plus

Need to save input somewhere, like a stack!



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* $E \rightarrow \mathrm{num}$
$_1$ 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

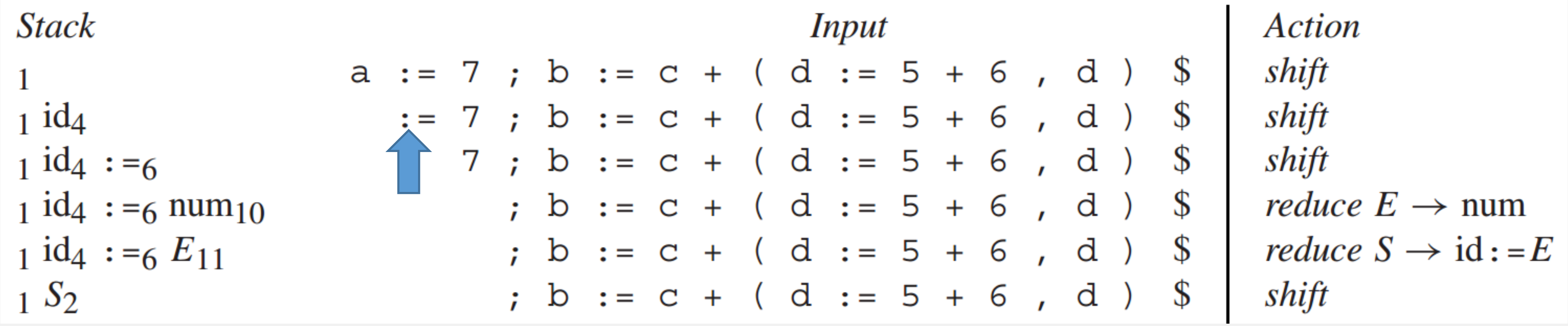
- L = left-to-right
- R = rightmost derivation

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

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

When parse is here, cant determine whether it’s an assign or a plus

Need to save input somewhere, like a stack!



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

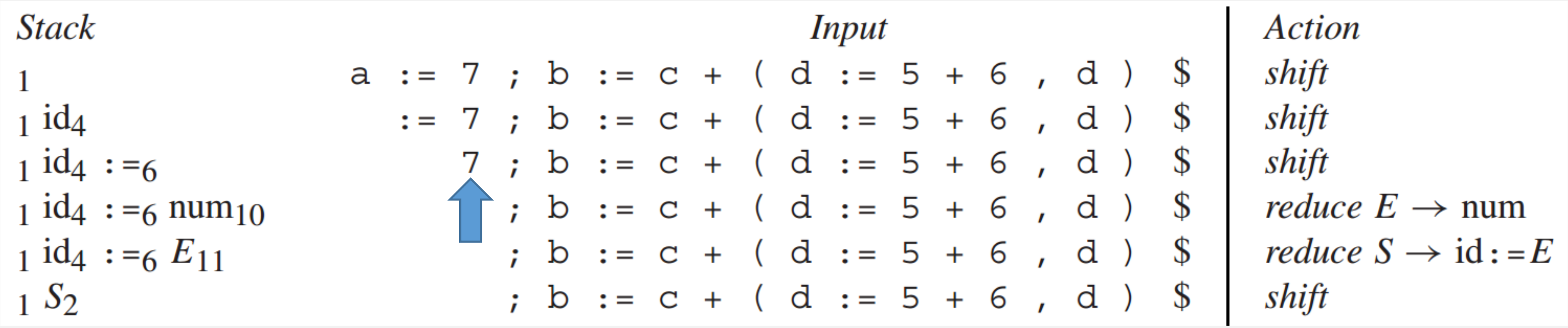
- L = left-to-right
- R = rightmost derivation

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

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

When parse is here, cant determine whether it’s an assign or a plus

Need to save input somewhere, like a stack!



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

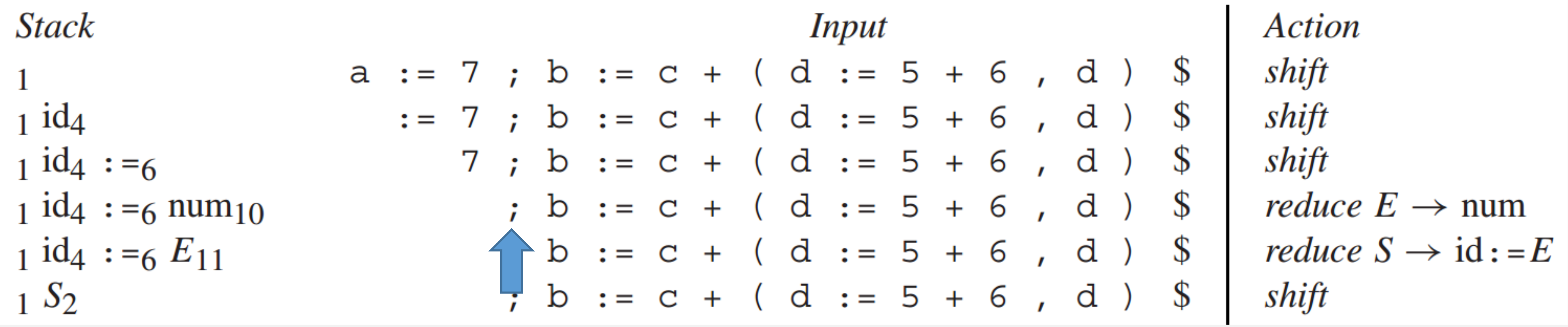
- L = left-to-right
- R = rightmost derivation

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

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

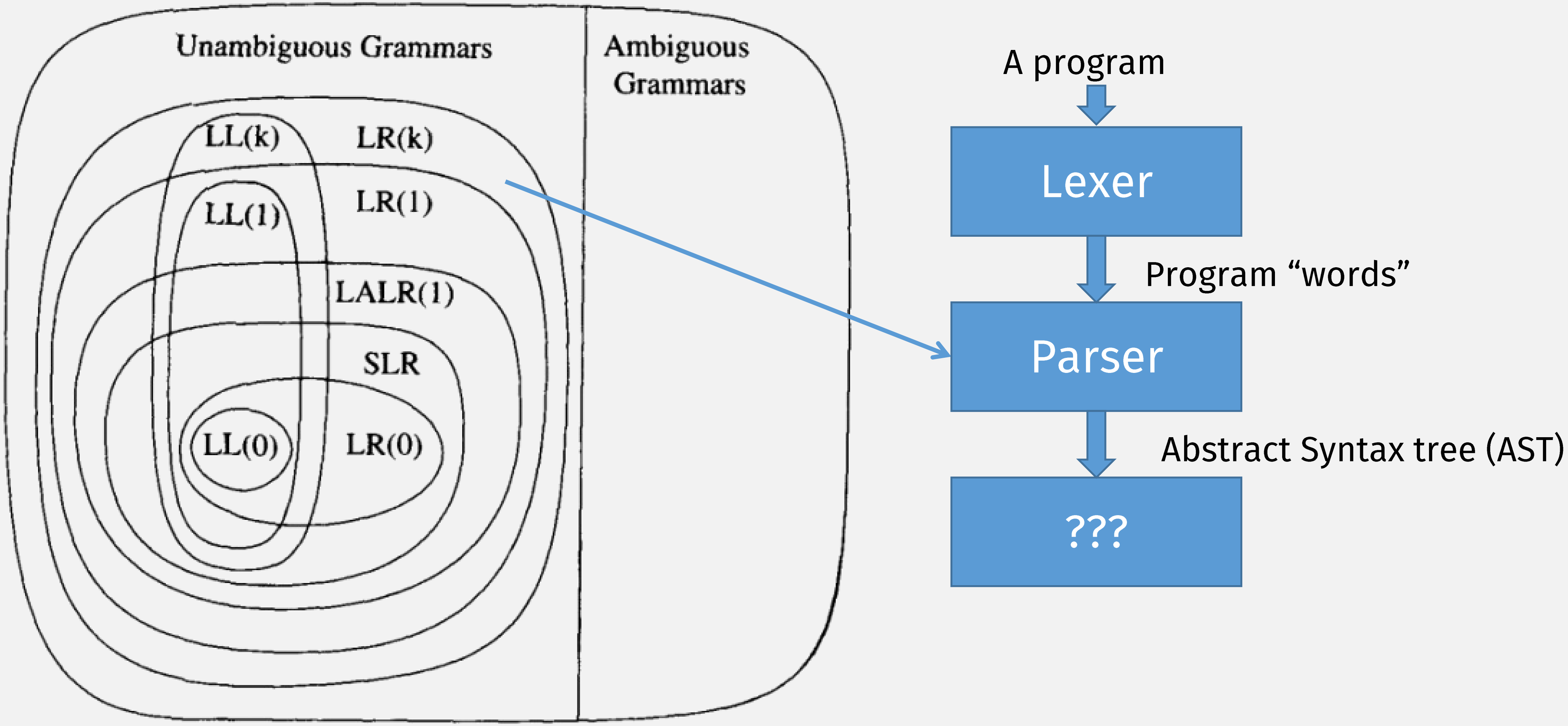
When parse is here, cant determine whether it’s an assign or a plus

Need to save input somewhere, like a stack!



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*

Take a Compilers Class!



A program

Lexer

Program “words”

Parser

Abstract Syntax tree (AST)

???

Check-in Quiz 10/14

On Gradescope

End of Class Survey 10/14

See course website