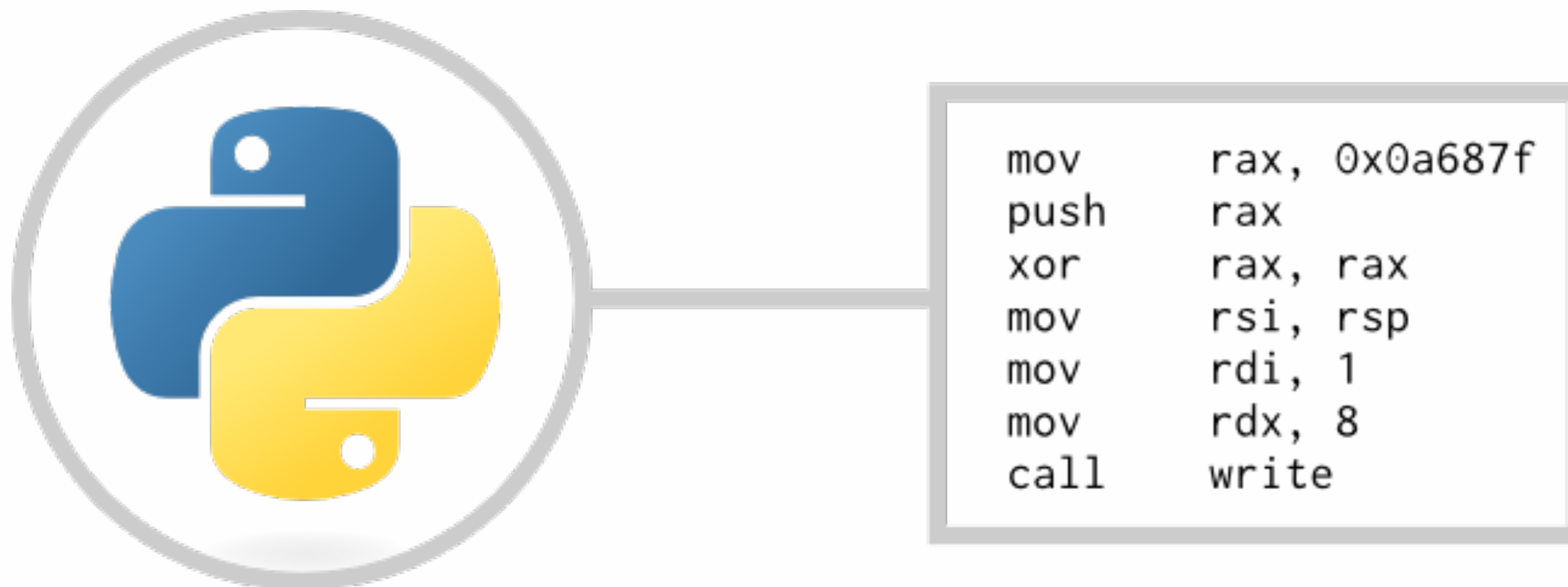
CS 480

Translators (Compilers)



extra slides on LR(0), SLR(1), and LR(1) parsing

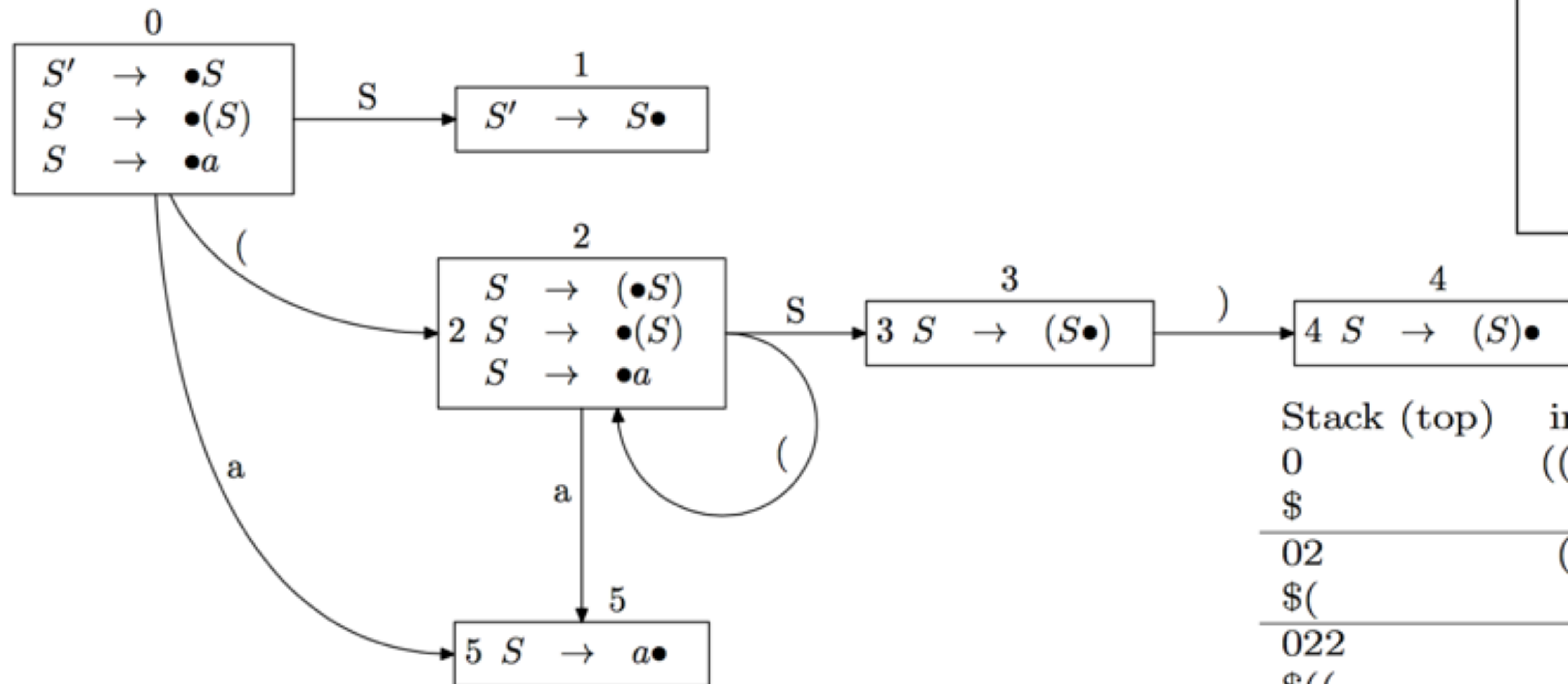
Instructor: Liang Huang

(materials extracted from David L. Hill's notes from Stanford CS 143)

LR(0) Parsing (no lookahead)

- very few grammars are in LR(0)

$$
\begin{aligned}
S' &\rightarrow \bullet S \\
S &\rightarrow \bullet(S) \\
S &\rightarrow \bullet a
\end{aligned}
$$



Stack (top)	input	action
0 $	((a))$	shift 2
02 $(	(a))$	shift 2
022 $((	a))$	shift 5
0225 $((a	))$	reduce $S \rightarrow a$
0223 $((S	))$	shift 4
02234 $((S)	))$	reduce $S \rightarrow (S)$
023 $(S	)$	shift 4
0234 $(S)	$	reduce $S \rightarrow (S)$
01 $S	$	accept

Unlike the example of shift-reduce parsing, an LR(0) parser does not actually shift symbols onto the stack. Instead, it shifts states. No information is lost because of the special structure of the LR(0) machine. Notice that every transition into a state has exactly the same symbol labelling it. If you see a state q on the stack, it is as though a were shifted onto the stack. The stack will also have states corresponding to nonterminal symbols.

LR(0) Parsing Algorithm

shift If the next input is a and there is a transition on a from the top state on the stack (call it q_i) to some state q_j, push q_j on the stack and remove a from the input.

reduce If the state has a reduce item $A \to \alpha \bullet$

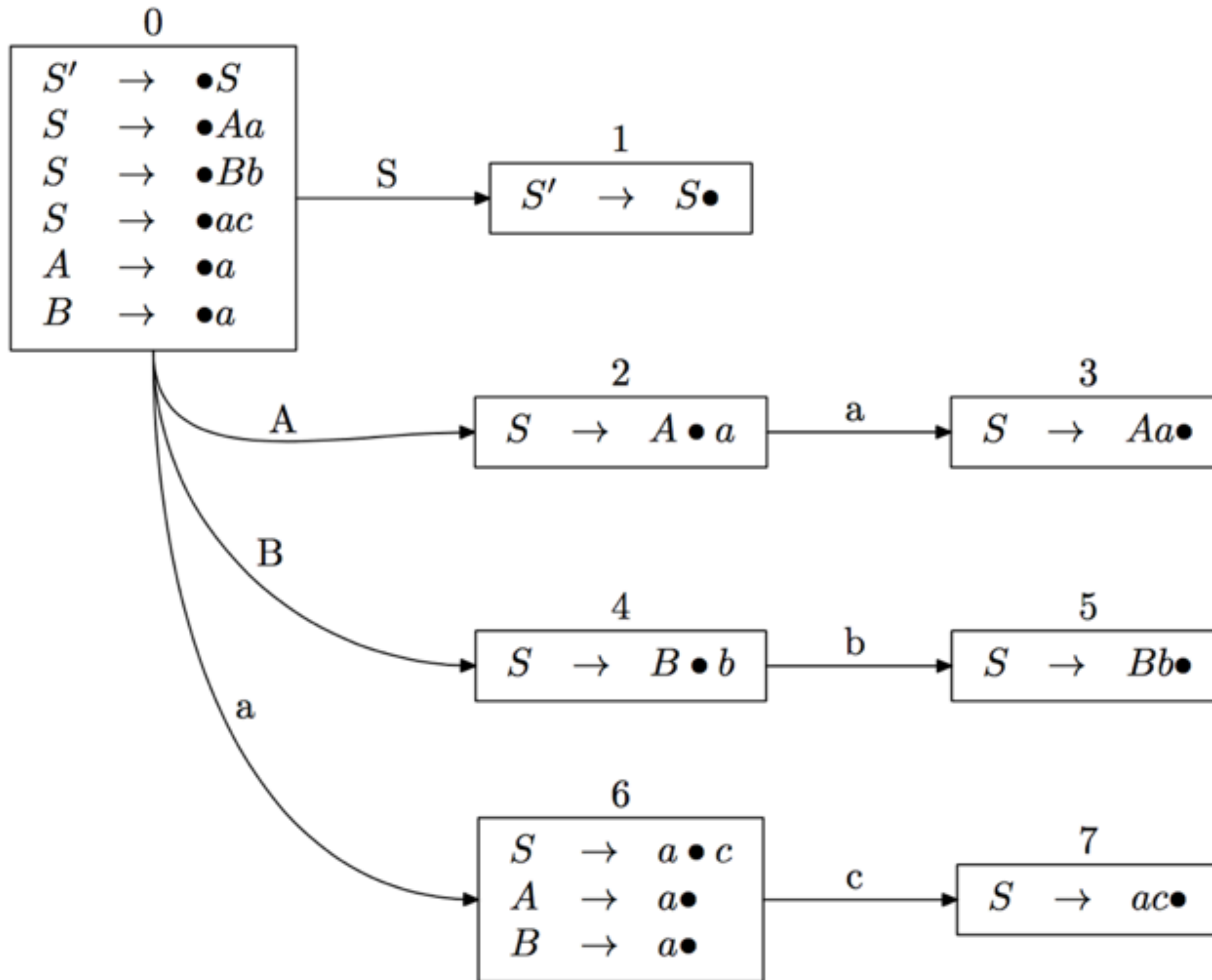
1. Pop one state on the stack for every symbol in α (note: symbols associated with these states will *always* match symbols in α).
2. Let the top state on the stack now be q_i. There will be a transition in the LR(0) machine on A to a state q_j. Push q_j on the stack

error If the state has no reduce item, the next input is a, and there is no transition on a, report a parse error and halt.

accept When the item $S' \to S \bullet$ is reduced accept if the next input symbol is \$, otherwise report an error and halt. (This rule is a bit weird. The remaining LR-style parsing algorithms don't need to check if the input is empty. We define it this way so LR(0) parsing can do our simple example grammar.)

LR(0) Fails on...

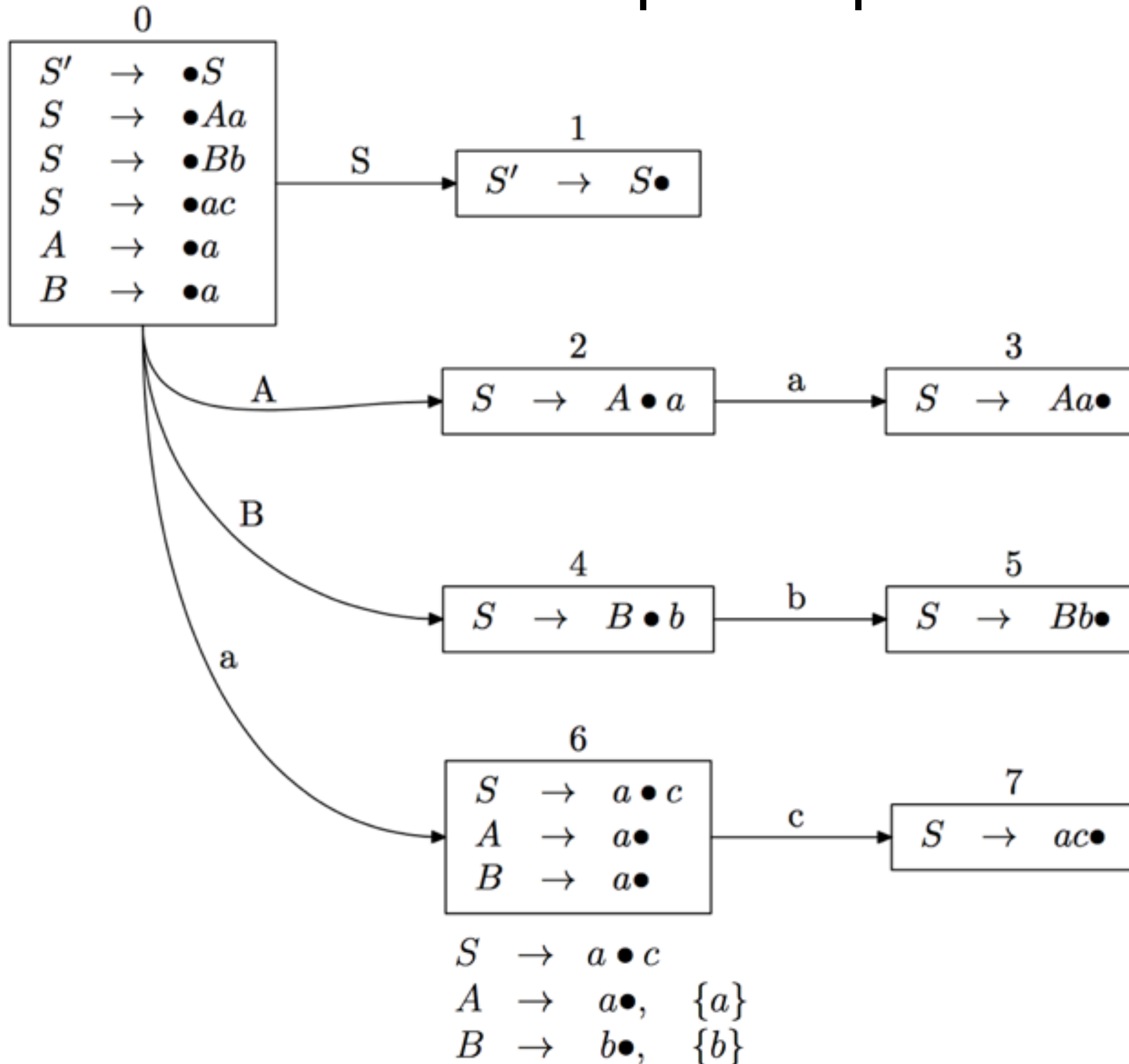
- a state has both shift and reduce or two reduces



0	S'	$\rightarrow$	S
1	S	$\rightarrow$	Aa
2	S	$\rightarrow$	Bb
3	S	$\rightarrow$	ac
4	A	$\rightarrow$	a
5	B	$\rightarrow$	a

SLR(1) -- use FOLLOW sets

- FOLLOW sets are precomputed



0	S'	$\rightarrow$	S
1	S	$\rightarrow$	Aa
2	S	$\rightarrow$	Bb
3	S	$\rightarrow$	ac
4	A	$\rightarrow$	a
5	B	$\rightarrow$	a

	ACTION				*GOTO*		
	a	b	c	$	S	A	B
0	s6				1	2	4
1				acc			
2	s3						
3				r1			
4		s5					
5				r2			
6	r4	r5	s7				
7				r3			

SLR(1) Fails on...

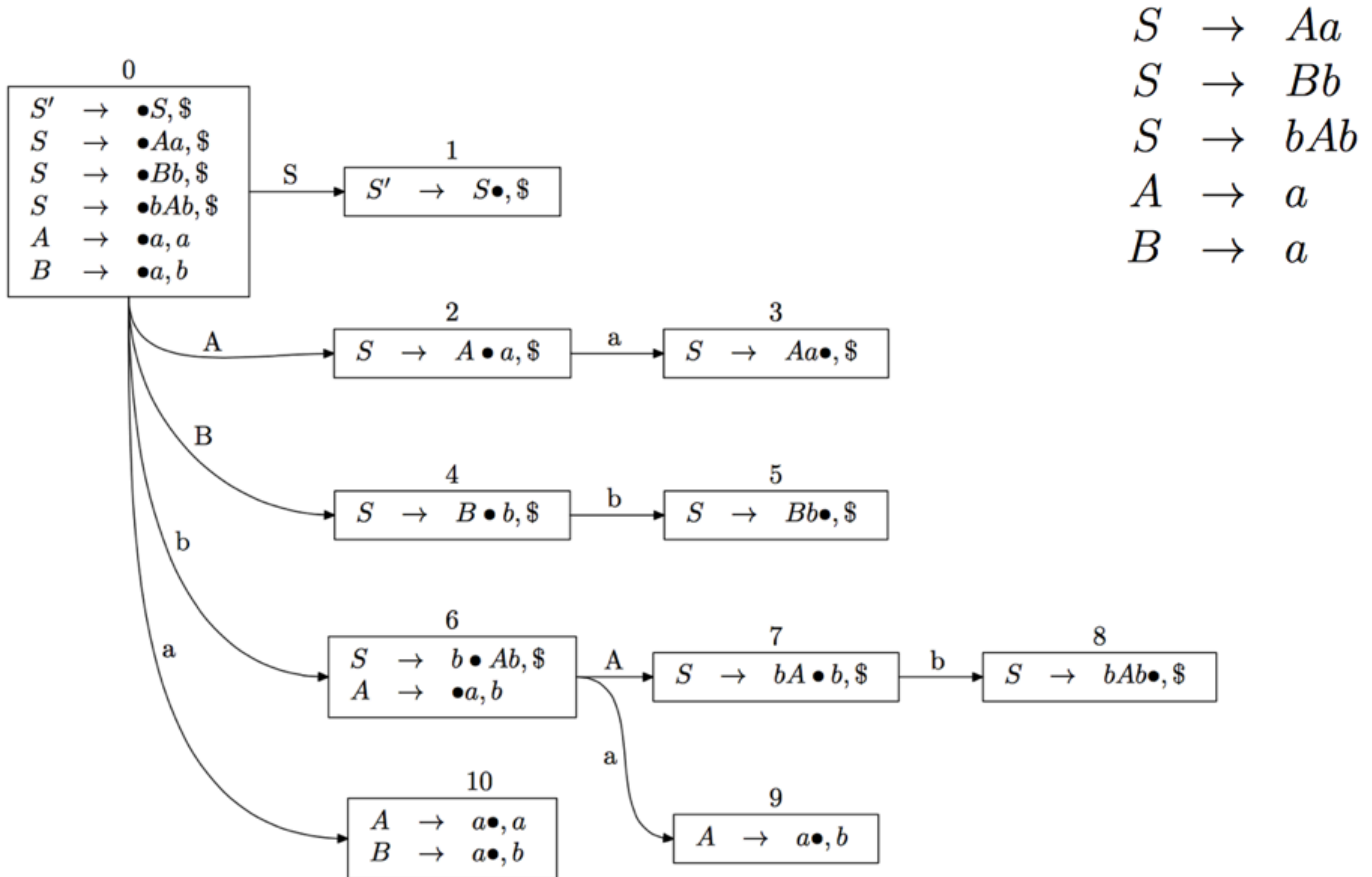
- unnecessary conflicts

$$\begin{array}{rcl} S' & \rightarrow & \bullet S \\ S & \rightarrow & \bullet Aa \\ S & \rightarrow & \bullet Bb \\ S & \rightarrow & \bullet bAb \\ A & \rightarrow & \bullet a \\ B & \rightarrow & \bullet a \end{array}$$

$$\begin{array}{rcl} A & \rightarrow & a\bullet \\ B & \rightarrow & a\bullet \end{array}$$

$$\begin{array}{rcl} S & \rightarrow & Aa \\ S & \rightarrow & Bb \\ S & \rightarrow & bAb \\ A & \rightarrow & a \\ B & \rightarrow & a \end{array}$$

LR(1)



LR(1) shift-reduce conflict

```
0
    S -> . E          $
    E -> . E + E      $/+
    E -> . int        $/+
```

```
1
    E -> int .        $/+
```

```
2
    S -> E .          $
    E -> E . + E      $/+
```

```
3
    E -> E + . E      $/+
    E -> . E + E      $/+
    E -> . int        $/+
```

```
4
    E -> E + E .      $/+
    E -> E . + E      $/+
```

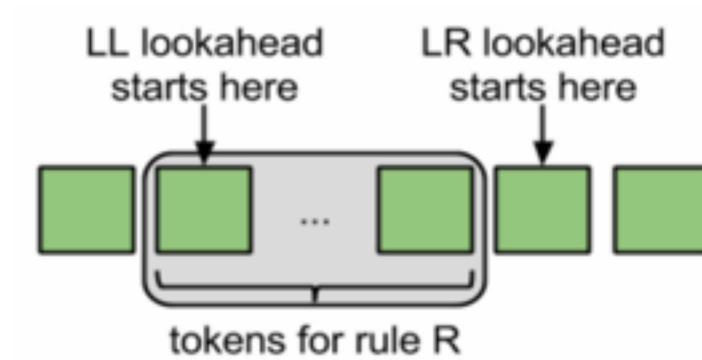
Projected Grade Stats

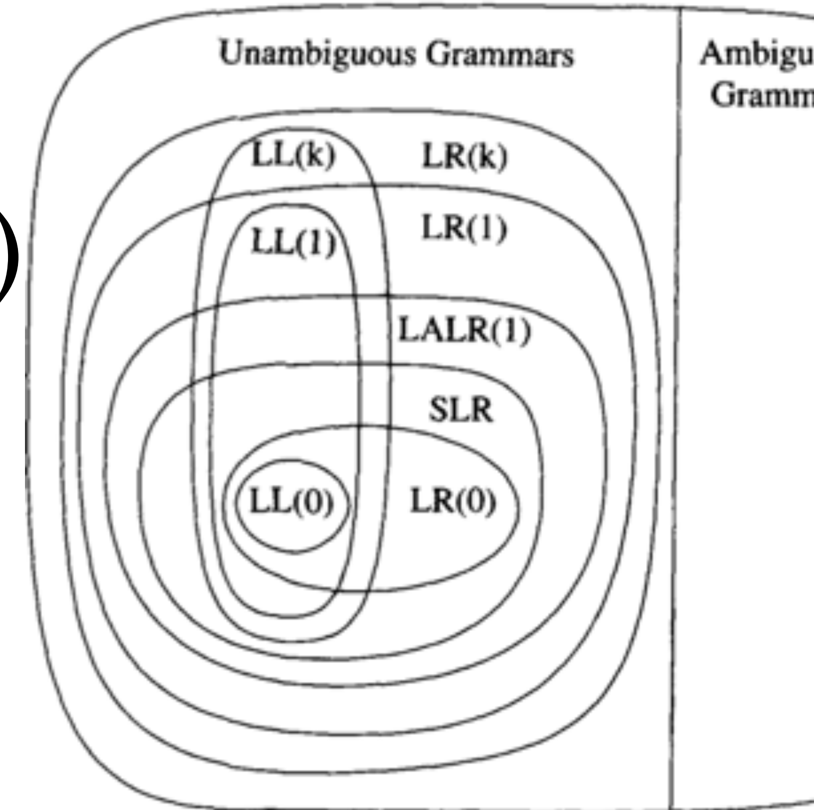
- based on total%; projected grade in "Notes" column
- A (11): 83+
- A- (9): 74~80
- B+ (6): 72~74
- B (12): 63~70
- B- (3): 59~62
- C+ (11): 54-56
- C (12): 44~51
- D, D+, C- (3): 35~39
- F (4): missing three or more HWs

my courses *before* withdraw deadline		**my courses *after* withdraw deadline**	
A/A-	25%	**A/A-**	30%
B+/B/B-	30%	**B+/B/B-**	40%
C+/C	20%	**C+/C**	20%
C-/D+	10%	**C-/D+**	5%
F	15%	**F**	5%

Top-Down (LL) Parsing

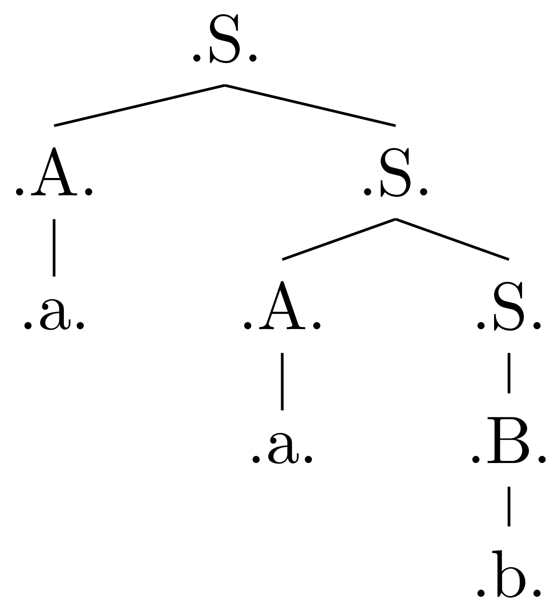
- predictive (deductive) instead of inductive
- predict next rule by lookahead (FIRST sets)
- two implementations
  - "recursive descent"
  - construct LL action table, and use stack





- many grammars are not LL(1) but can be converted to LL(1)

$$
\begin{aligned}
S &\rightarrow AS \mid B \\
A &\rightarrow a \\
B &\rightarrow b
\end{aligned}
$$



parse	(top) stack	action
aab$	S$	expand $S \rightarrow AS$
aab$	AS$	expand $A \rightarrow a$
aab$	aS$	match
ab$	S$	expand $S \rightarrow AS$
ab$	AS$	expand $A \rightarrow a$
ab$	aS$	match
b$	S$	expand $S \rightarrow B$
b$	B$	expand $B \rightarrow b$
b$	b$	match
$	$	accept

Are these grammars in LL(1)?

- infix expression
- Polish notation
- reverse-Polish notation
- P_0 from HW1
- P_0.8 from midterm review and P_0.9 from midterm

```
module : simple_stmt
simple_stmt : "print" expr NEWLINE
expr : decint
     | "-" expr
     | "(" expr ")"
```

```
module : stmt+
stmt : (assign_stmt | print_stmt) NEWLINE
assign_stmt : name "=" decint
print_stmt : "print" name
```

```
stmt : (assign_stmt | iadd_stmt | print_stmt) NEWLINE
iadd_stmt : name "+=" decint
```

Surgery 1: Left Factoring

- common left factors: right-hand sides of two rules from the same nonterminal start with the same symbol
  $A \rightarrow \alpha \quad A \rightarrow \beta \quad \text{First}(\alpha) \cap \text{First}(\beta) \neq \varnothing$

- solution: extract common left factors

$$
\begin{array}{lcl}
E & \rightarrow & T + E \\
E & \rightarrow & T
\end{array}
$$

This converts to $E \rightarrow T(+E \mid \epsilon)$, which converts back to

$$
\begin{array}{lcl}
E & \rightarrow & T A \\
A & \rightarrow & + E \\
A & \rightarrow & \epsilon
\end{array}
$$

Surgery 2: Eliminate Left Recursion

- left recursion: $A \Rightarrow^{+} A\ \alpha$
- solution: make it right recursive

$$\begin{aligned} E &\rightarrow E + T \\ E &\rightarrow T \end{aligned}$$

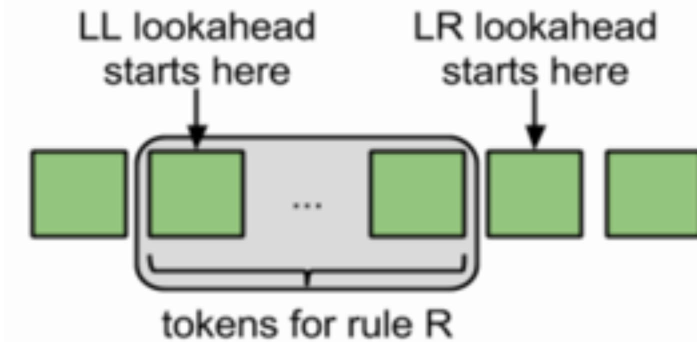
which can be rewritten as $E \rightarrow E + T \mid T$. In turn, this is $E \rightarrow T(+T)^{*}$, which re-expanded right recursively into

$$\begin{aligned} E &\rightarrow T A \\ A &\rightarrow + T A \\ A &\rightarrow \epsilon \end{aligned}$$

Precise Definition of LL(1)

$$\text{First}(A) = \{a \mid a \in \Sigma \wedge A \Rightarrow^* a\beta\}$$

$$\text{Follow}(A) = \{a \mid a \in \Sigma \wedge S' \Rightarrow^* \alpha A a \beta\}$$



- for all A, if $A \rightarrow \alpha \mid \beta$, then $\text{First}(\alpha) \cap \text{First}(\beta) = \varnothing$
- if $\alpha \Rightarrow^* \varepsilon$, then $\text{First}(\beta) \cap \text{Follow}(A) = \varnothing$

$$\text{Lookahead}(A \rightarrow B_1 B_2 \ldots B_n) = \bigcup \{\text{First}(B_i) \mid \forall 1 \leq k < i . B_k \Rightarrow^* \epsilon\}$$

$$\cup \begin{cases} \text{Follow}(A) & \text{if } B_1 B_2 \ldots B_k \Rightarrow \epsilon \\ \emptyset & \text{otherwise} \end{cases}$$

equivalent definition: Lookahead sets disjoint

Example

```
E -> E + T | T
T -> T * F | F
F -> (E) | int
```

```
E  -> T E'
E' -> + T E' | ε
T  -> F T'
T' -> * F T' | ε
F  -> (E) | int
```

```
FIRST(E)  = {(, int}
FIRST(E') = {+}
FIRST(T)  = {(, int}
FIRST(T') = {*}
FIRST(F)  = {(, int}
```

```
FOLLOW(S)  = {$}
FOLLOW(E)  = {$, )}
FOLLOW(E') = {$, )}
FOLLOW(T)  = {+, $, )}
FOLLOW(T') = {+, $, )}
FOLLOW(F)  = {*, +, $, )}
```

	+	*	(	)	int	$
E			E -> T E'		E -> T E'	
E'	E' -> + T E'			E' -> ε		E' -> ε
T			T -> F T'		T -> F T'	
T'	T' -> ε	T' -> * F T'		T' -> ε		T' -> ε
F			F -> (E)		F -> int	

#	Input	Stack (top is left)
1	3 + 5 * 7 $	E
2	3 + 5 * 7 $	T E'
3	3 + 5 * 7 $	F T' E'
4	3 + 5 * 7 $	int T' E'
5	+ 5 * 7 $	T' E'
6	+ 5 * 7 $	E' (use T' -> ε)
7	+ 5 * 7 $	+ T E'
8	5 * 7 $	T E'
9	5 * 7 $	F T' E'
10	5 * 7 $	int T' E'
11	* 7 $	T' E'
12	* 7 $	* F T' E'
13	7 $	F T' E'
14	7 $	int T' E'
15	$	T' E'
16	$	E' (use T' -> ε)
17	$	DONE! (use E' -> ε)

Recursive Descent Implementation

- one function per non-terminal

	a	$+$	$*$	$($	$)$	$\$$
E	$E \to TA$			$E \to TA$		
T	$T \to FB$			$T \to FB$		
F	$F \to a$			$F \to (E)$		
A		$A \to +TA$			$A \to \epsilon$	$A \to \epsilon$
B		$B \to \epsilon$	$B \to *FB$		$B \to \epsilon$	$B \to \epsilon$

```
int parse_B() {
  next = peektok();    /* look at next token */
  if (next == '*') {
    gettok();    /* remove '*' from input */
    parse_F();   /* should check error returns for these, */
    parse_B();   /* but I want to keep the code short */
    return 1;    /* successfully parsed B */
  }
  else if ((next == '+') || (next == ')') || (next == '$')) {
    return 1;        /* successfully parsed B -> ε */
  }
  else {
    error("got %s, but expected *, +, ) or $ while parsing B\n", next);
    return 0;
  }

}

int parse_T() {
  parse_F();  /* again, should check the return code */
  while ((next = peektok()) == '*') {
    gettok();    /* remove '*' from input */
    parse_F(); }

}
```