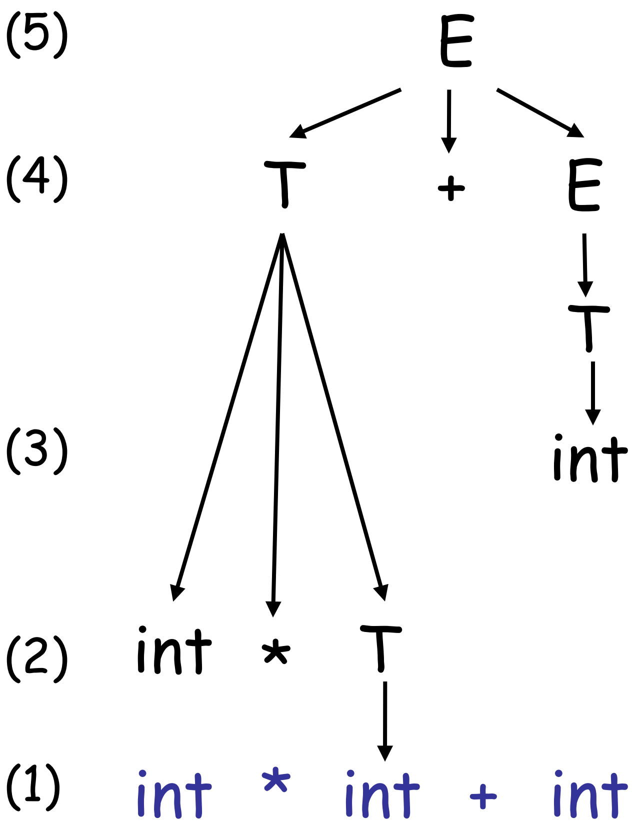
Bottom-up parsing

Bottom-up parsing

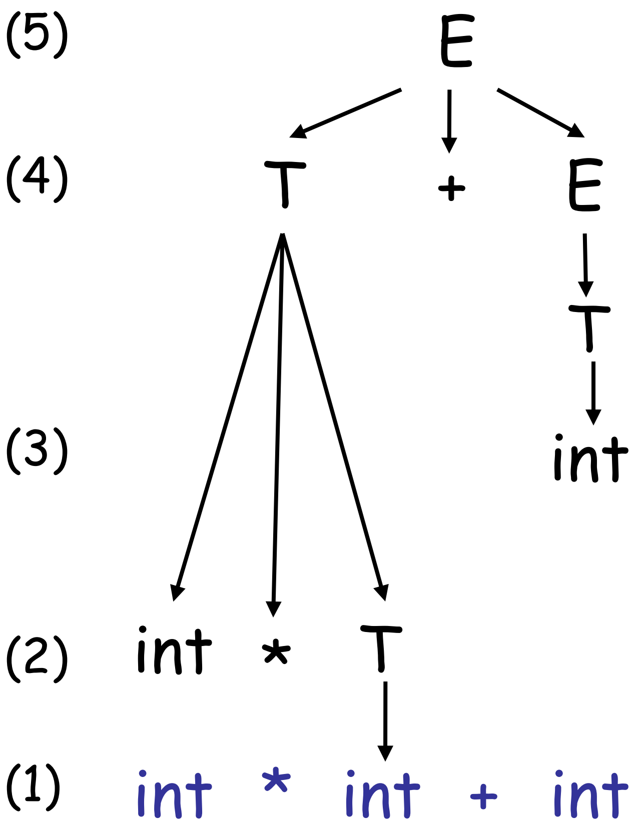
- Bottom-up parsing builds a parse tree from the leaves (terminals) to the start symbol



$$E \rightarrow T + E \mid T$$
$$T \rightarrow int * T \mid int$$

Bottom-up parsing

- Bottom-up parsing builds a parse tree from the leaves (terminals) to the start symbol

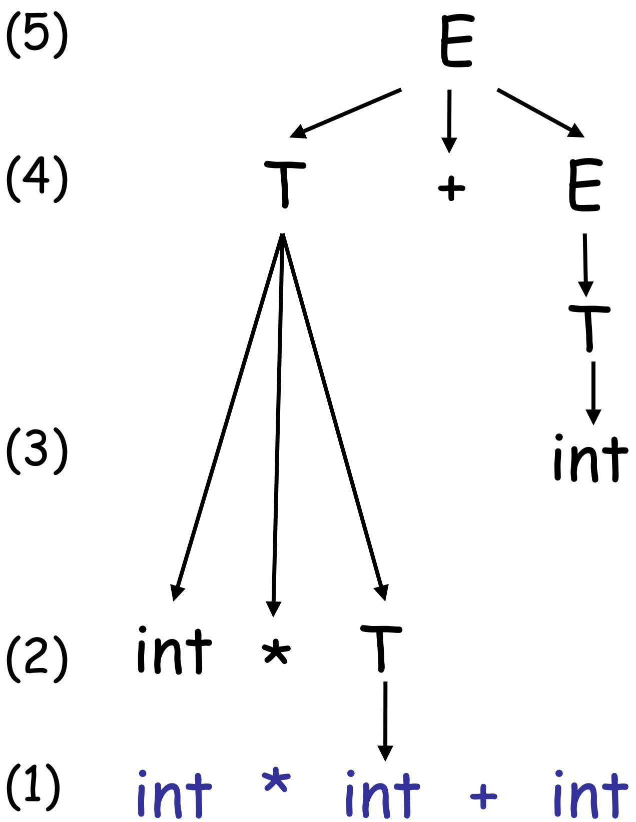


E => T + E
=> E + T
=> E + int
=> int * T + int
=> int * int + int

Rightmost Derivation

Bottom-up parsing

- Bottom-up parsing builds a parse tree from the leaves (terminals) to the start symbol



E => T + E
=> E + T
=> E + int
=> int * T + int
=> int * int + int

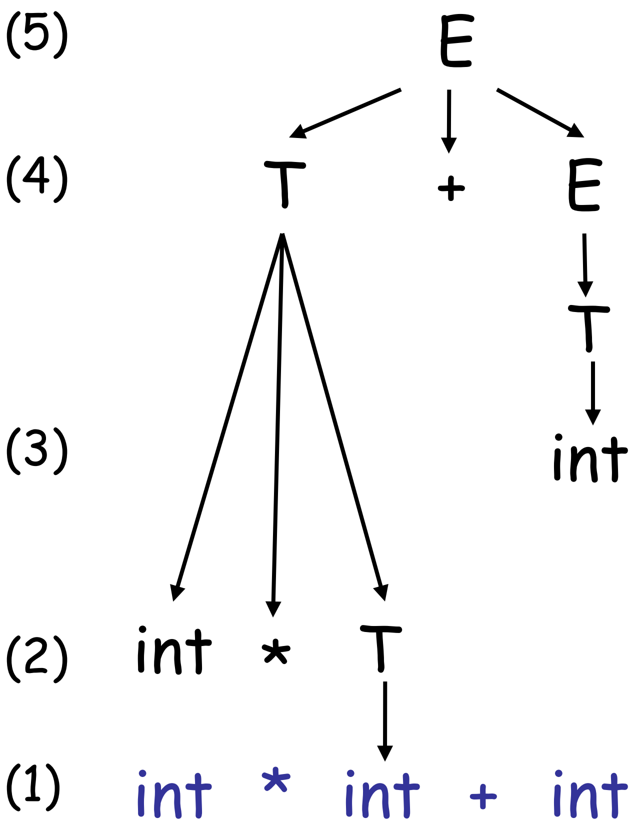
Rightmost Derivation

E => T + E
=> int * T + E
=> int * int + E
=> int * int + T
=> int * int + int

Leftmost Derivation

Bottom-up parsing II

- Bottom-up parsing is a series of reductions (inverses of productions), the reverse of which is the rightmost derivation



E → T + E | T
T → int * T | int

(1) int --> T
(2) int * T --> T
(3) int --> T
(4) T --> E
(5) T + E --> E

Reductions

E => T + E
=> E + T
=> E + int
=> int * T + int
=> int * int + int

Rightmost Derivation

LR(k)

- Most popular bottom-up parsing method is LR(k) parsing
- ***L***eft-to-right scanning of input
- ***R***ightmost derivation
- With an input lookahead of ***k***

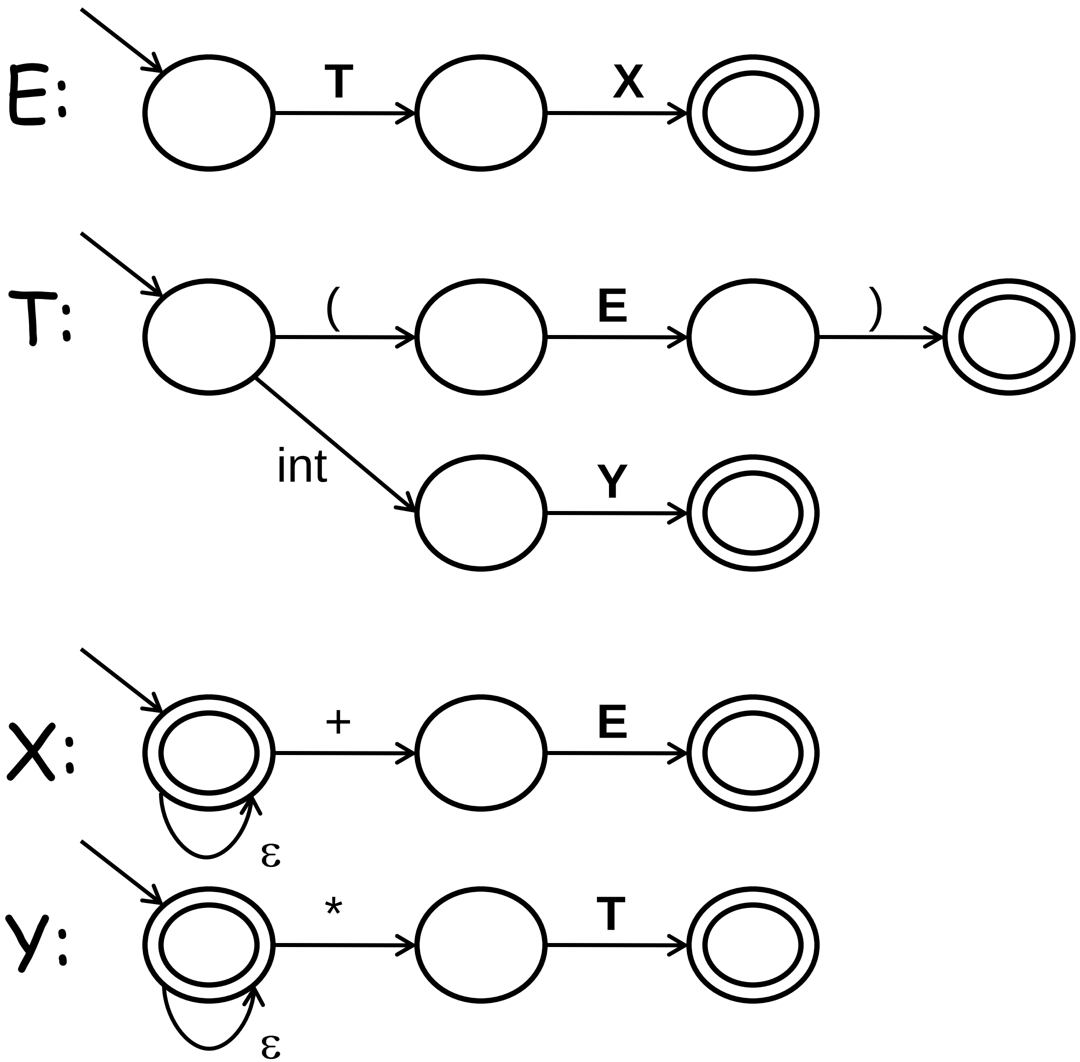
Why LR(k) parsing?

- Recognizes most programming language constructs
  - LR(k) recognizes the body of a production in right-sentential form with k symbols of lookahead
    - Determine when to apply reductions, $A \rightarrow \beta$, given string $\delta\beta a_1 \ldots a_k w$
  - LL(k) recognizes the use of a production after seeing the first k symbols of what the body derives
    - Determine when to apply productions, $A \rightarrow a_1 \ldots a_k \beta$, given string $w a_1 \ldots a_k \beta\delta$
- Possible to build efficient table-based algorithms
- LR(k) is a proper superset of LL(k)

Shift-reduce parsing

- LR parsers typically described as shift-reduce parsers
- Pushdown automata with 4 possible actions
  - Shift a: Move token a from input to stack
  - Reduce $A \to \beta$: Reduce sequence β on stack to A
  - Accept: Accept string
  - Error: Reject string
- Compare with actions used in LL parsing
  - Scan a: Pop token a from stack; Match a from input
  - Push $A \to \beta_1 \ldots \beta_n$: Pop A; Push states $\beta_n \ldots \beta_1$ to stack

Recall LL(1)



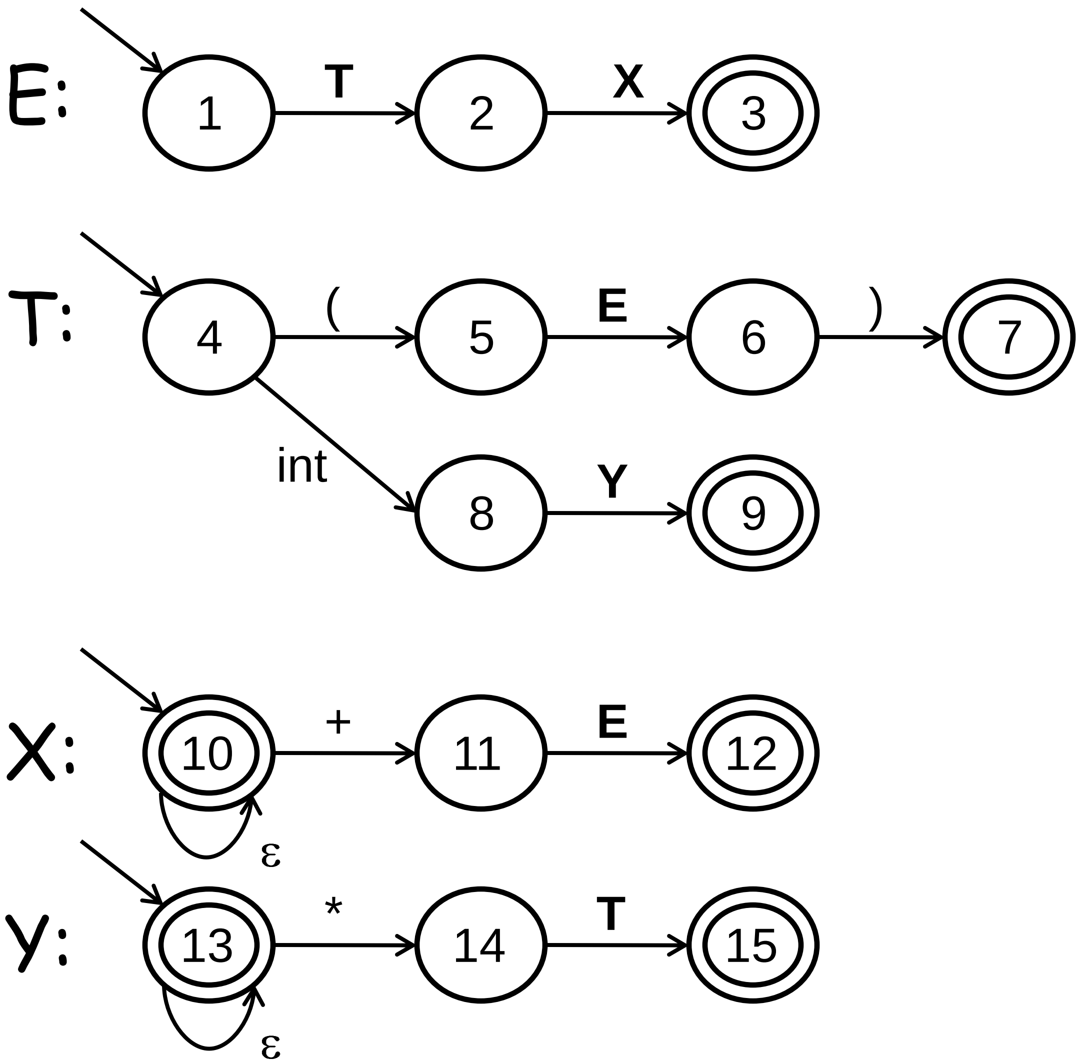
$E \to T\ X$

$T \to (\ E\ )\ |\ \text{int}\ Y$

$X \to +\ E\ |\ \varepsilon$

$Y \to *\ T\ |\ \varepsilon$

Recall LL(1)

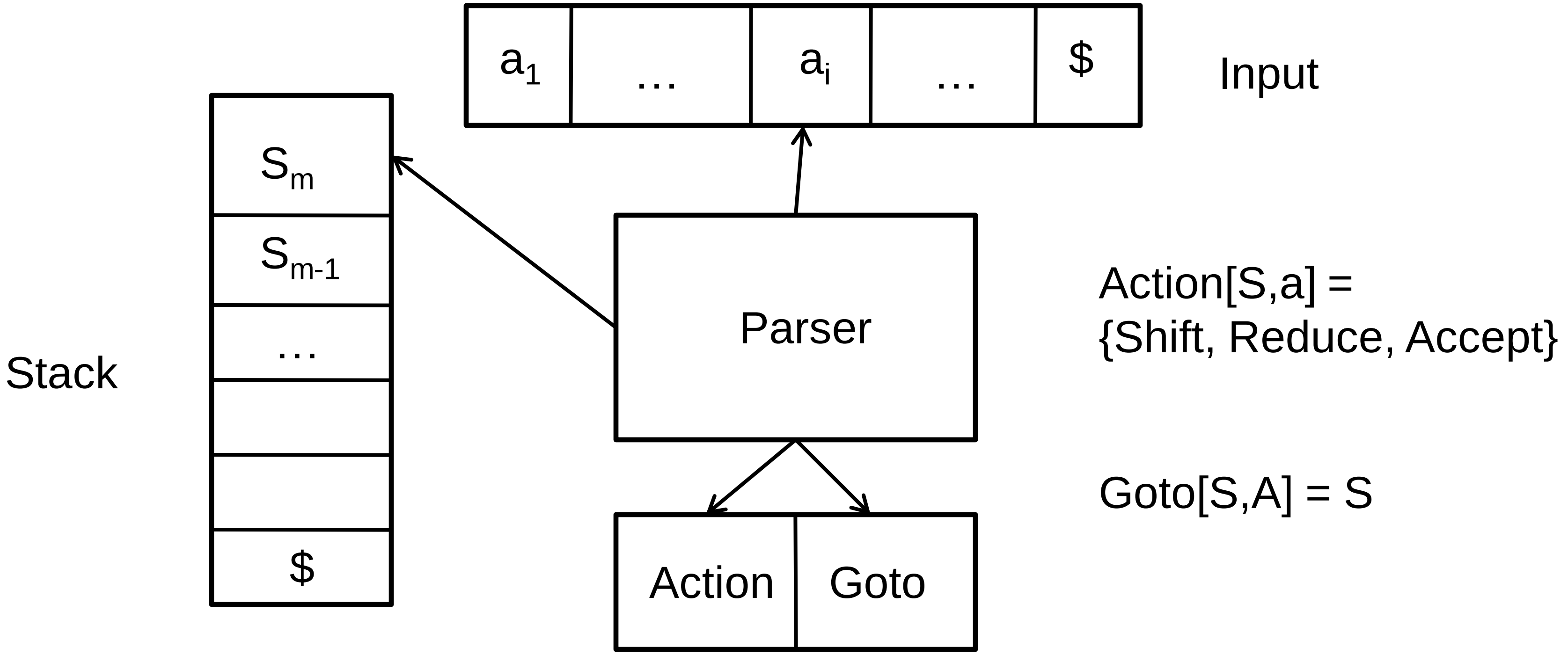


FOLLOW(X) = {), $ }
FOLLOW(Y) = { +,), $ }

Stack	Input
E	int * int $
X T	int * int $
X Y int	int * int $
X Y	* int $
X T *	* int $
X T	int $
X Y int	int $
X Y	$
X ε	$
…	…

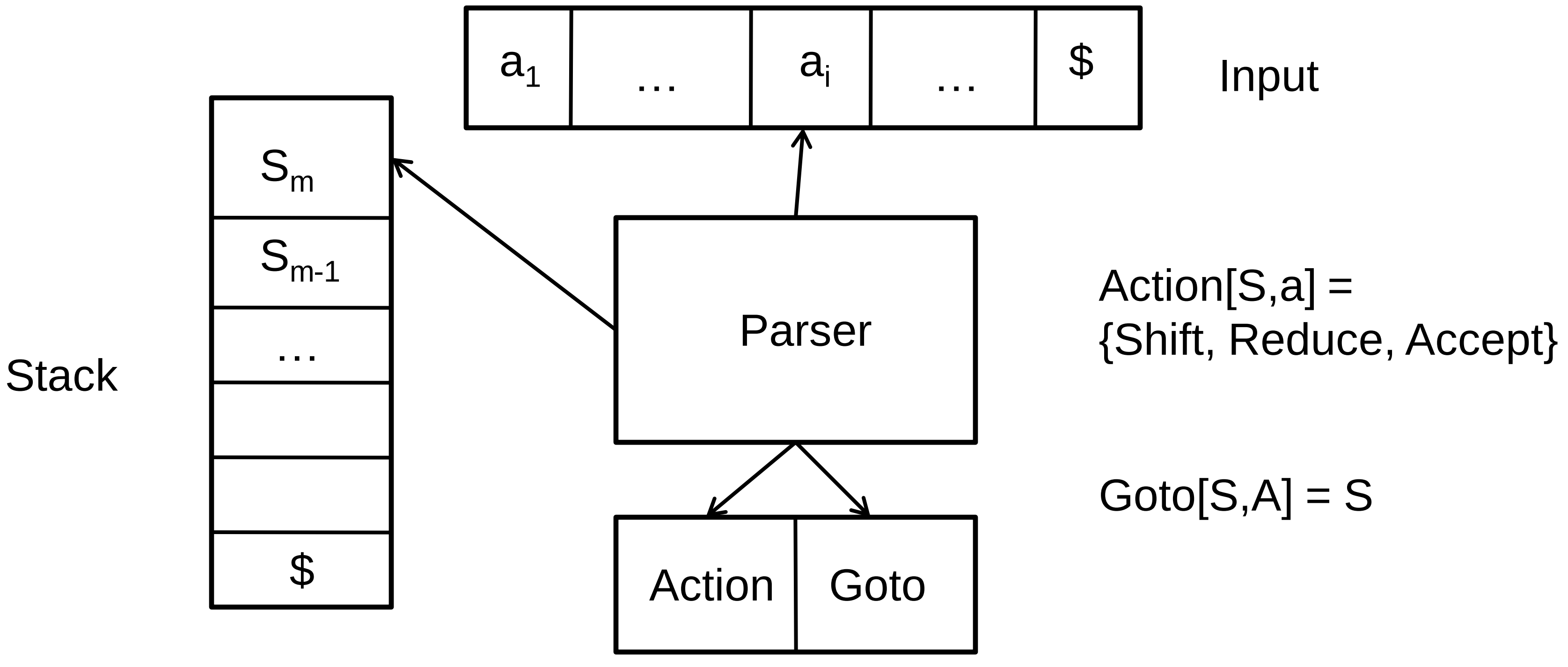
Shift-reduce parsing

- Shift S_m : Push S_m on stack; increment input position



Shift-reduce parsing

- Reduce $A \rightarrow \beta$: Pop $|\beta|$ symbols; push Goto[$S_{m-|\beta|}$,A] on stack



Shift-reduce parsing

State	Action						Goto			
	int	(	)	+	*	$	E	T	X	Y
1	S9	S6					13	2		
2			R5	S4		R5			3	
3			R1			R1				
4	S9	S6					5	2		
5			R4			R4				
6	S9	S6					7	2		
7			S8							
8				R2		R2				
9			R7	R7	S11	R7				10
10				R3		R3				
11	S9	S6						12		
12			R6	R6		R6				
13						acc				

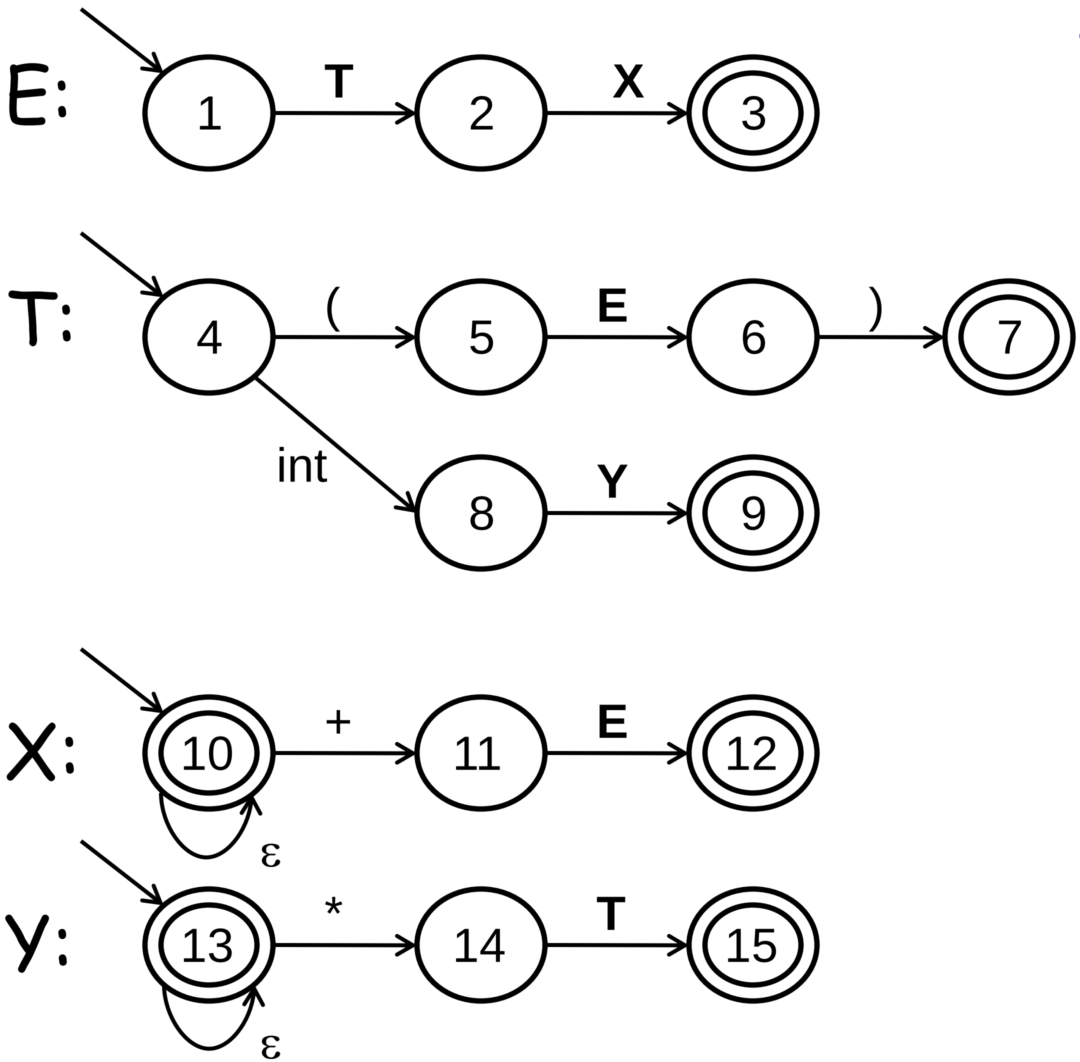
LR tables

- All LR parsers use the same basic algorithm, they only differ on how the transition tables are built
  - SLR(0), LR(1), LALR(1)
- The basic problem is determining when to shift and when to reduce
  - Use an LR(0) automaton to determine viable prefixes

How do we build transition tables?

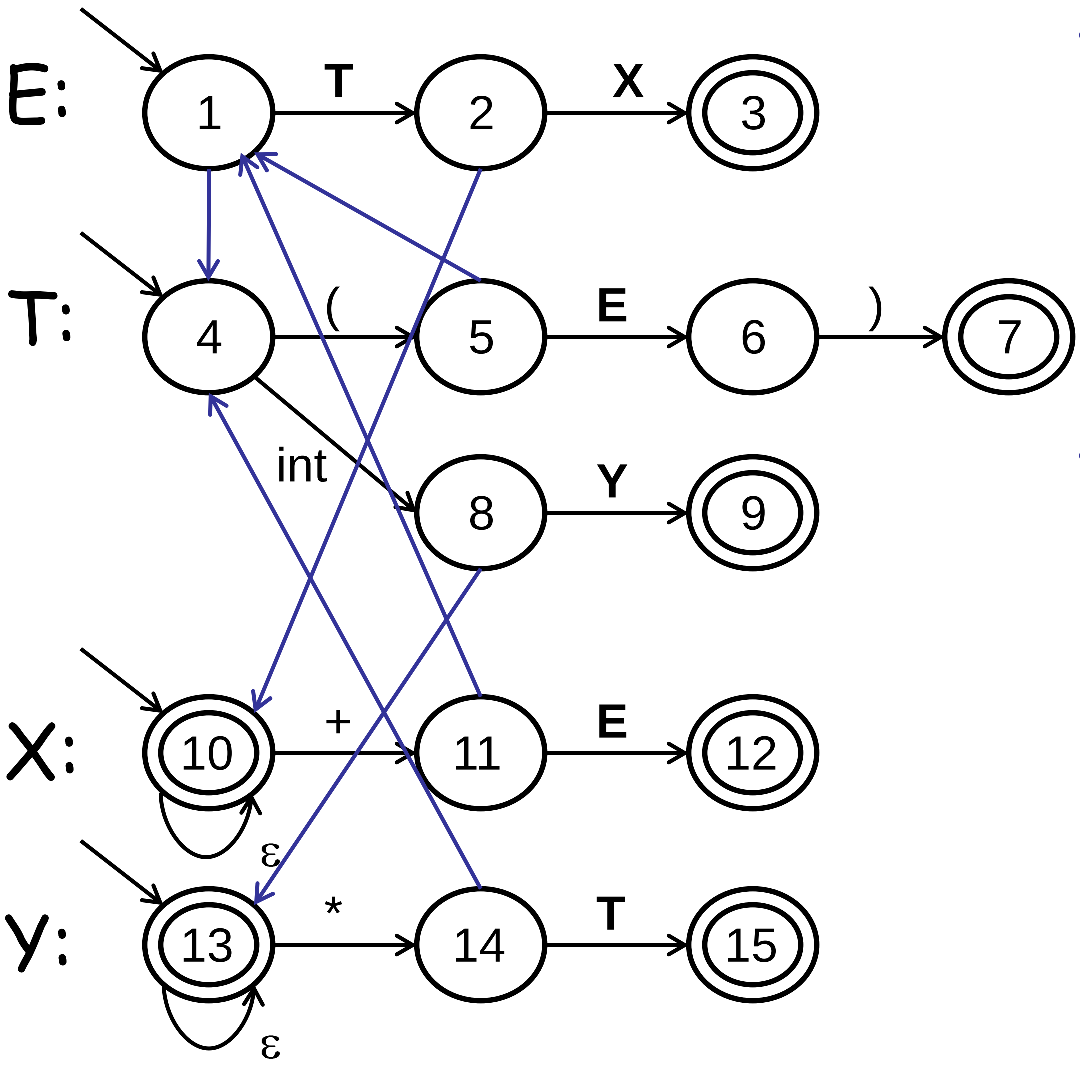
- LR(0) automata encapsulate all we need
  - Push-down automata with edges labeled with terminals and non-terminals
  - Reducing and accepting states
- Different capabilities due to
  - When to reduce
  - How automata are made deterministic

LR(0) automaton



- Start out with LL(1) picture
  - Separate automaton for each production

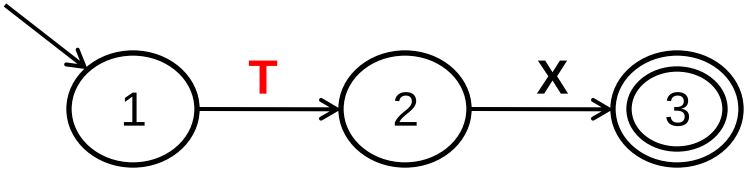
LR(0) automaton



- Add ε-transitions to indicate possible parsing states
  - Intuition: ε-transitions allow use to nondeterministically pick the right production to apply
- Apply NFA to DFA conversion

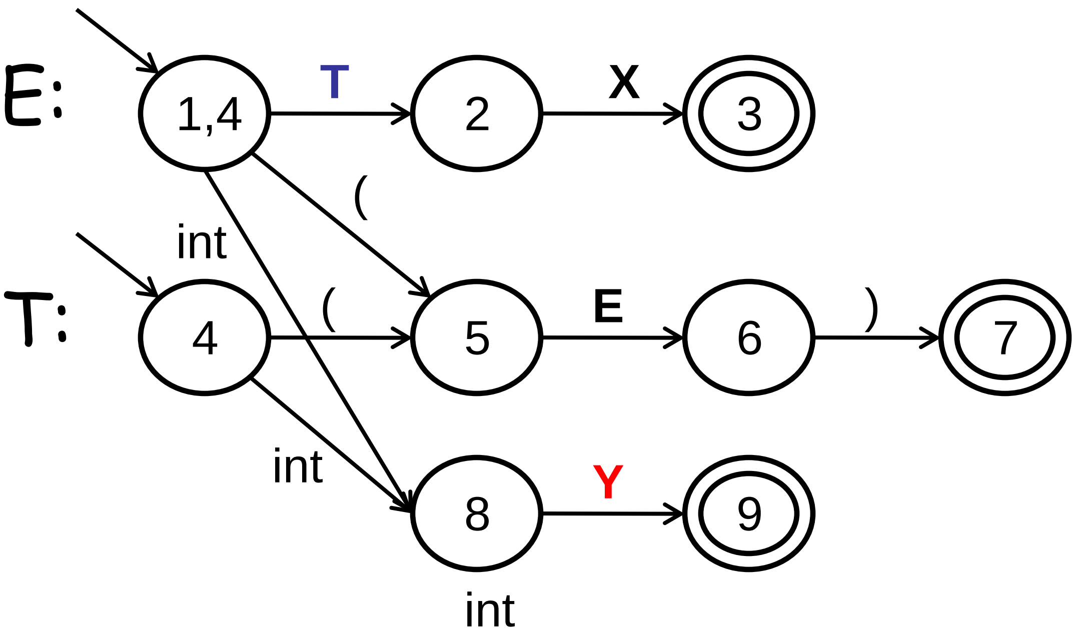
LR(0) automaton

E:



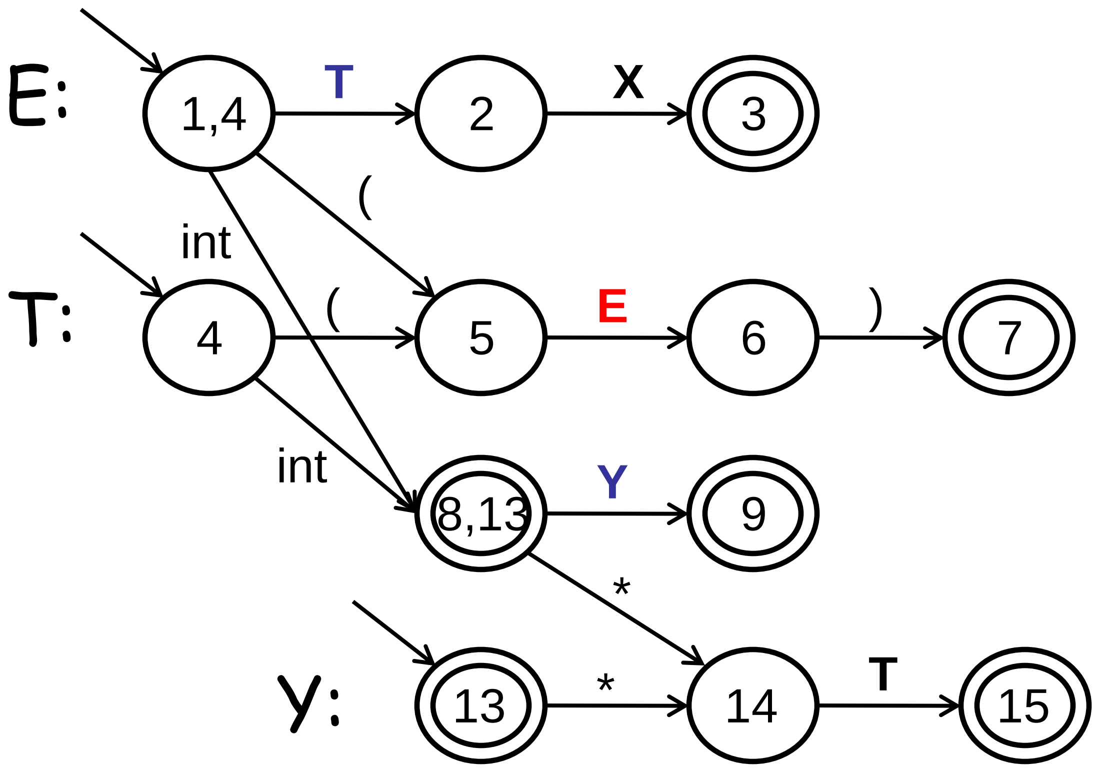
ε

LR(0) automaton

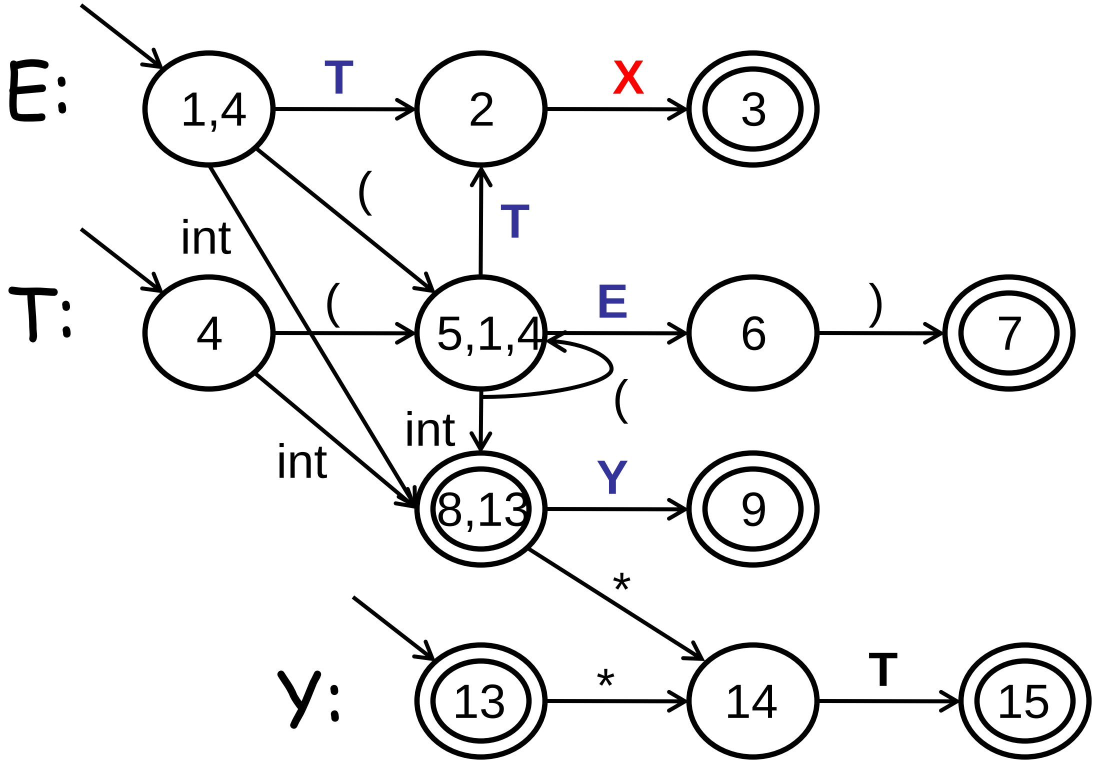


ε

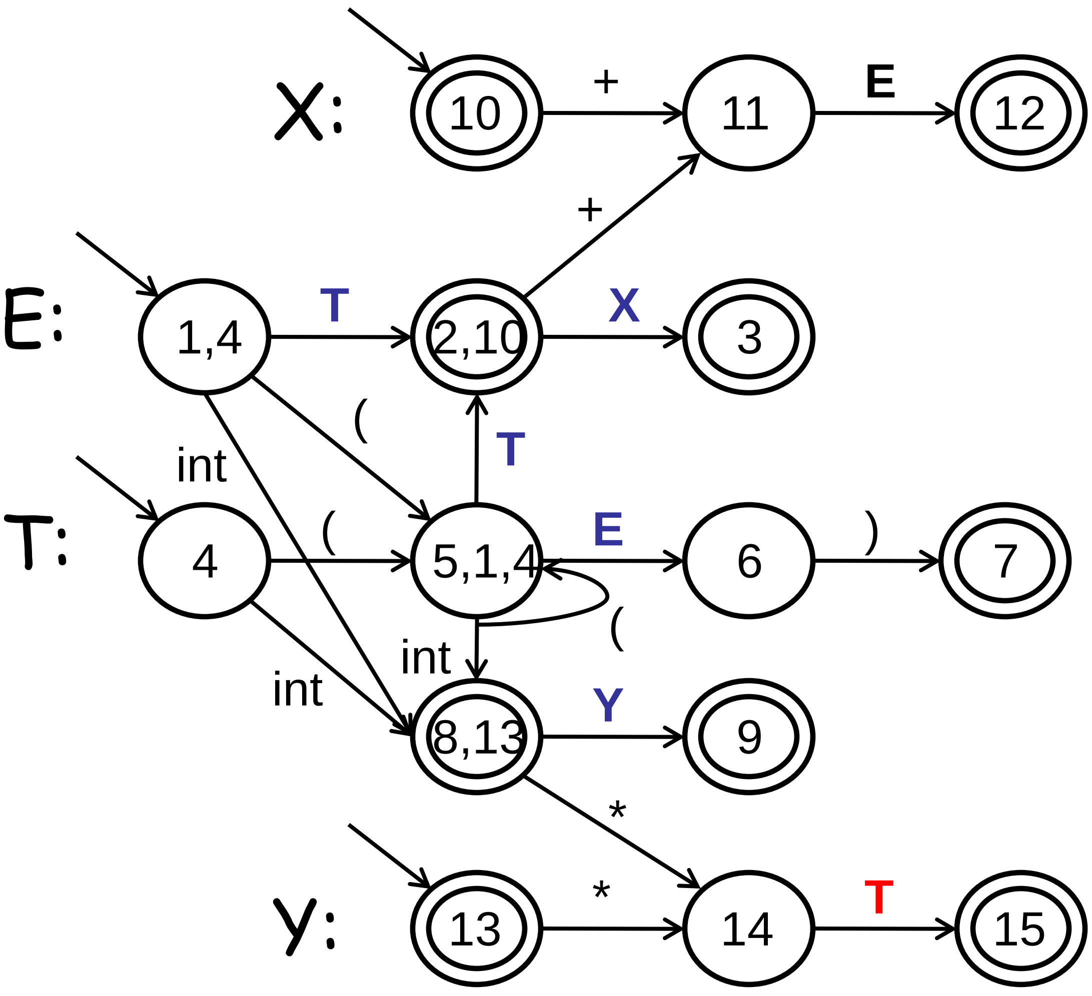
LR(0) automaton



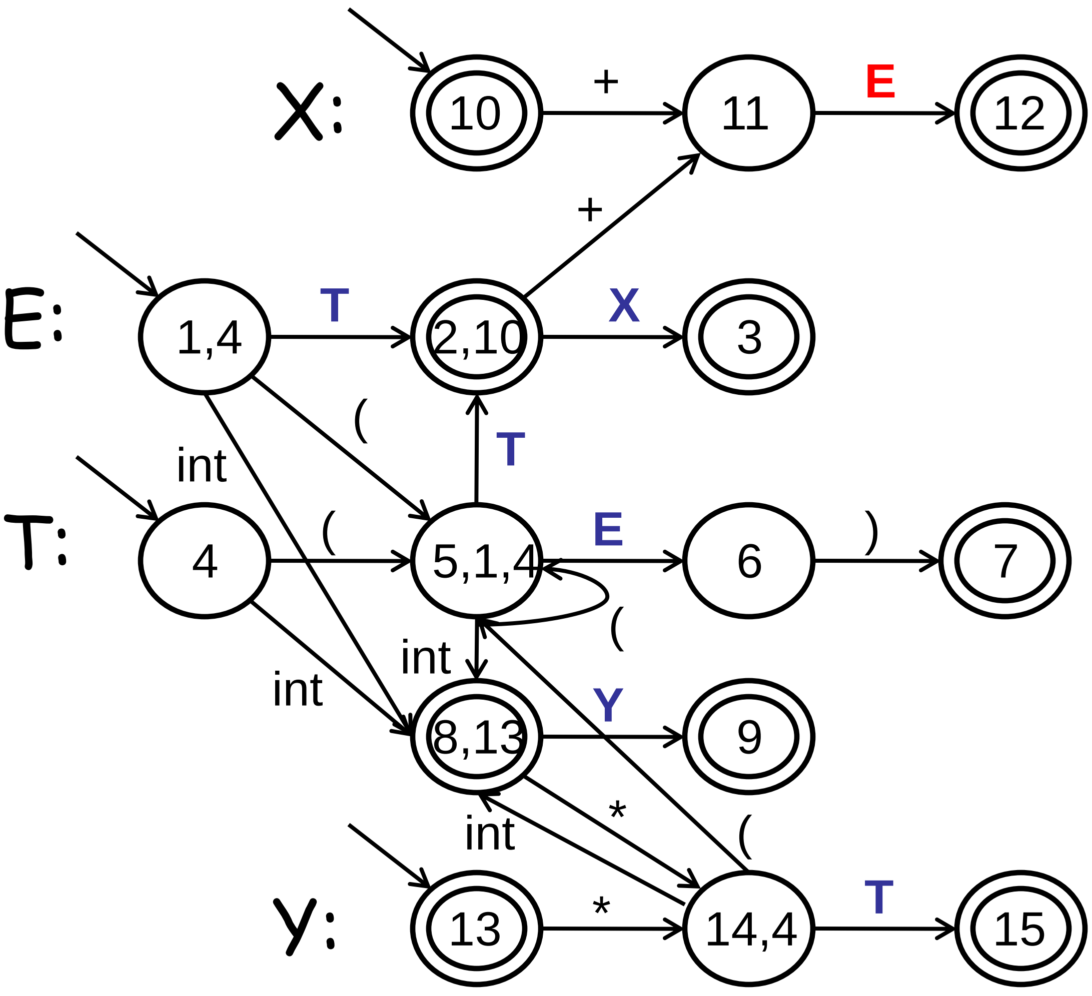
LR(0) automaton



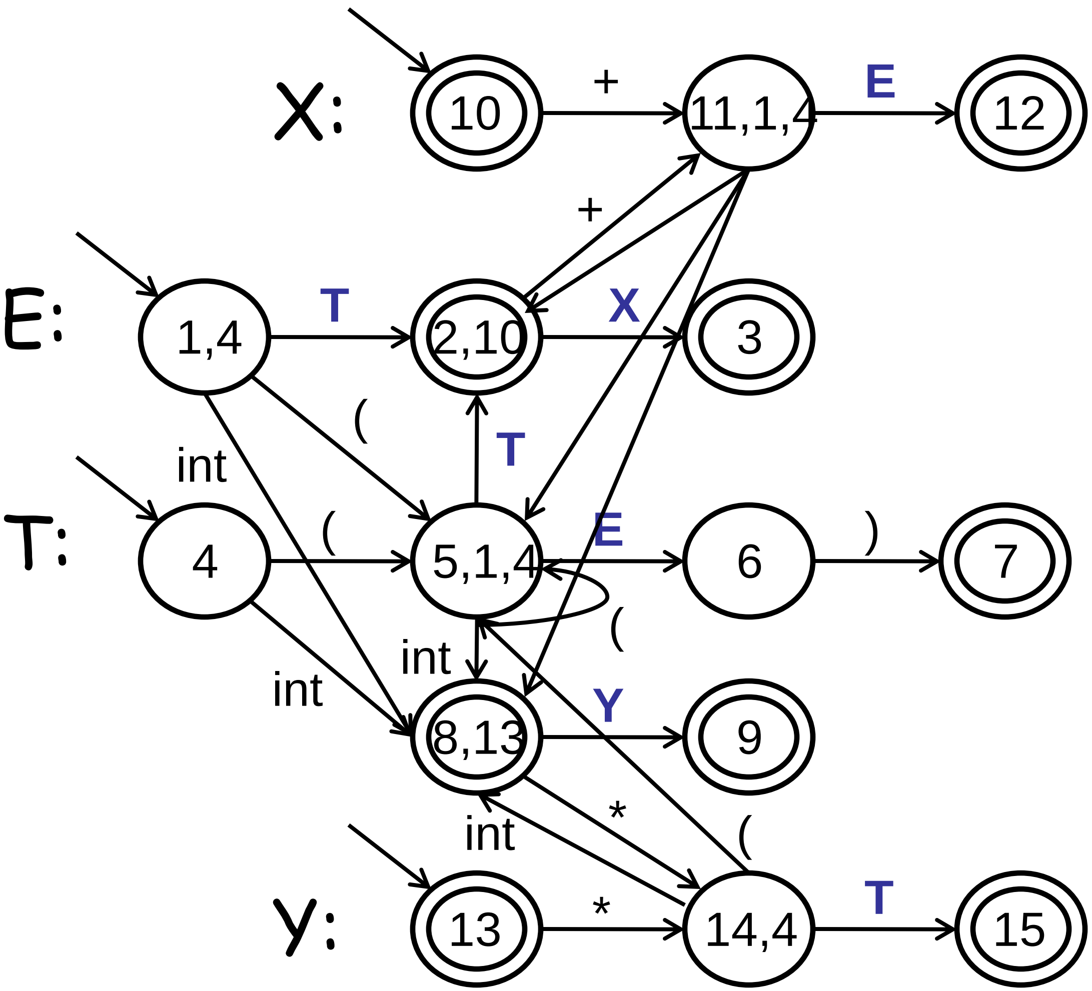
LR(0) automaton



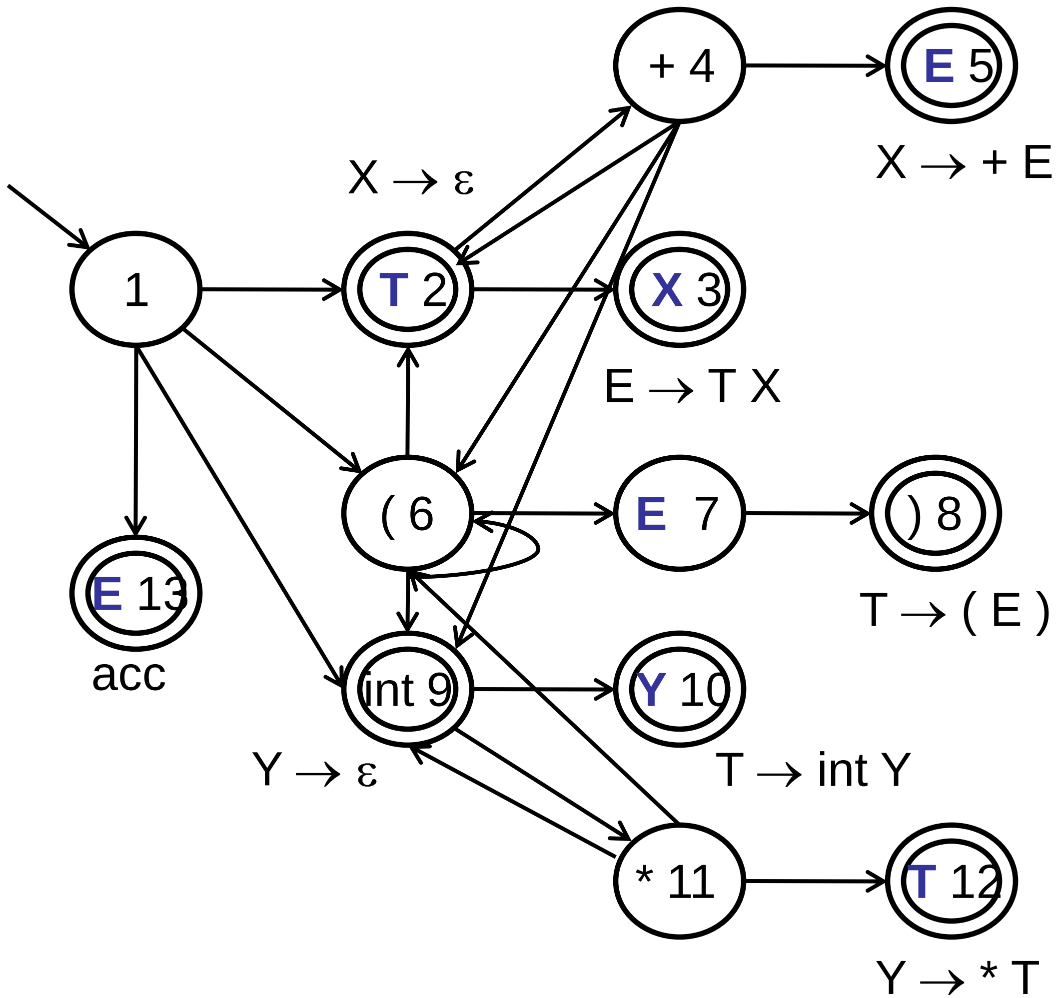
LR(0) automaton



LR(0) automaton



LR(0) automaton



$E \to T\ X$

$T \to (\ E\ )\ |\ int\ Y$

$X \to +\ E\ |\ \varepsilon$

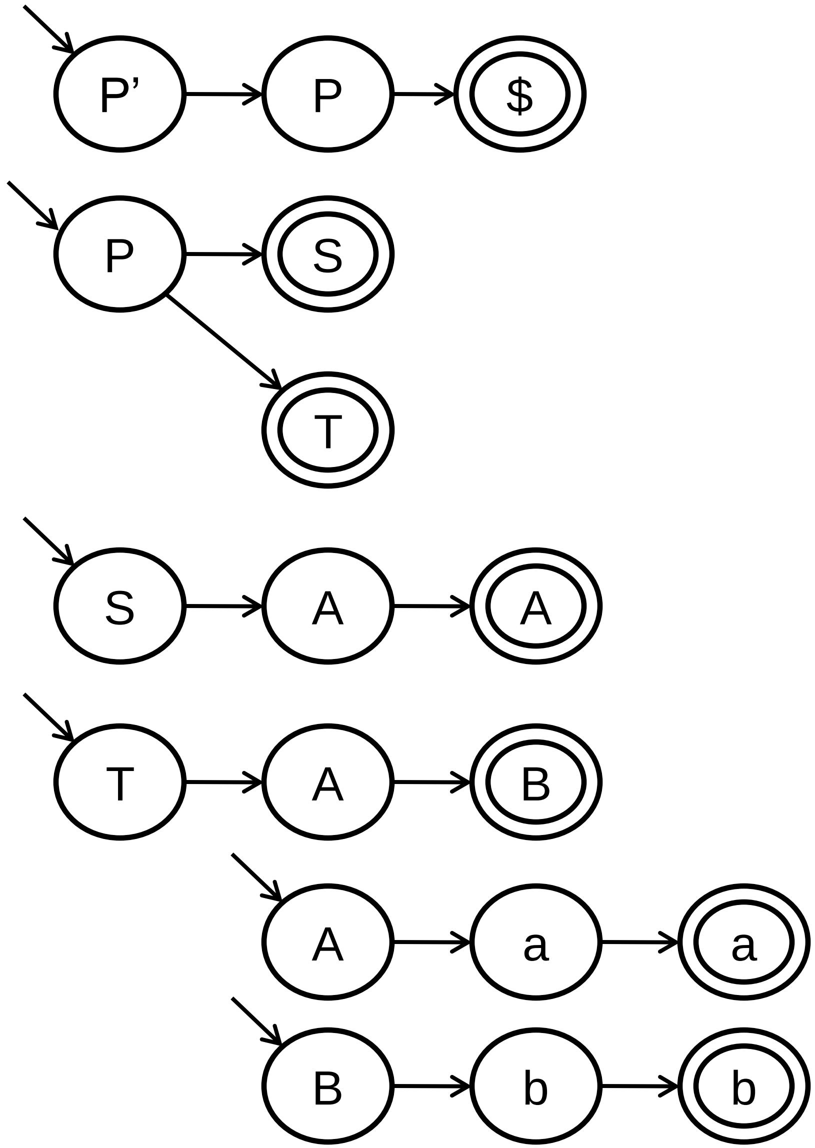
$Y \to *\ T\ |\ \varepsilon$

- Useful to augment grammar with rule S' → S to identify accepting state

Constructing an LR(0) automaton

1. Add a dummy start symbol [S' $\rightarrow$ S $]
   - Distinguishes accepting reductions
2. Make an automaton for each production
3. For transitions on non-terminals, add ε-edges to the corresponding automaton
4. Apply NFA to DFA conversion

Example



$P' \rightarrow P\ \$$

$P \rightarrow S \mid T$

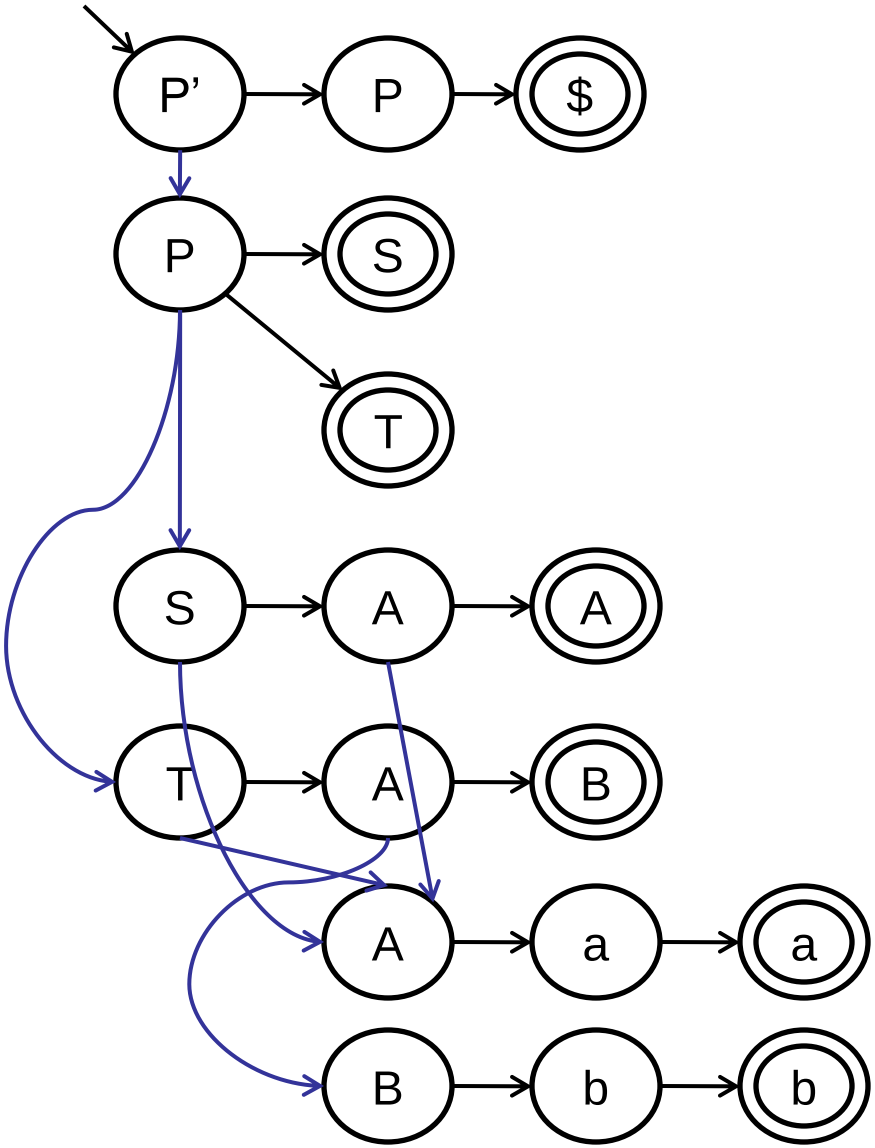
$S \rightarrow A\ A$

$T \rightarrow A\ B$

$A \rightarrow a\ a$

$B \rightarrow b\ b$

Example



$P' \rightarrow P\ \$$

$P \rightarrow S \mid T$

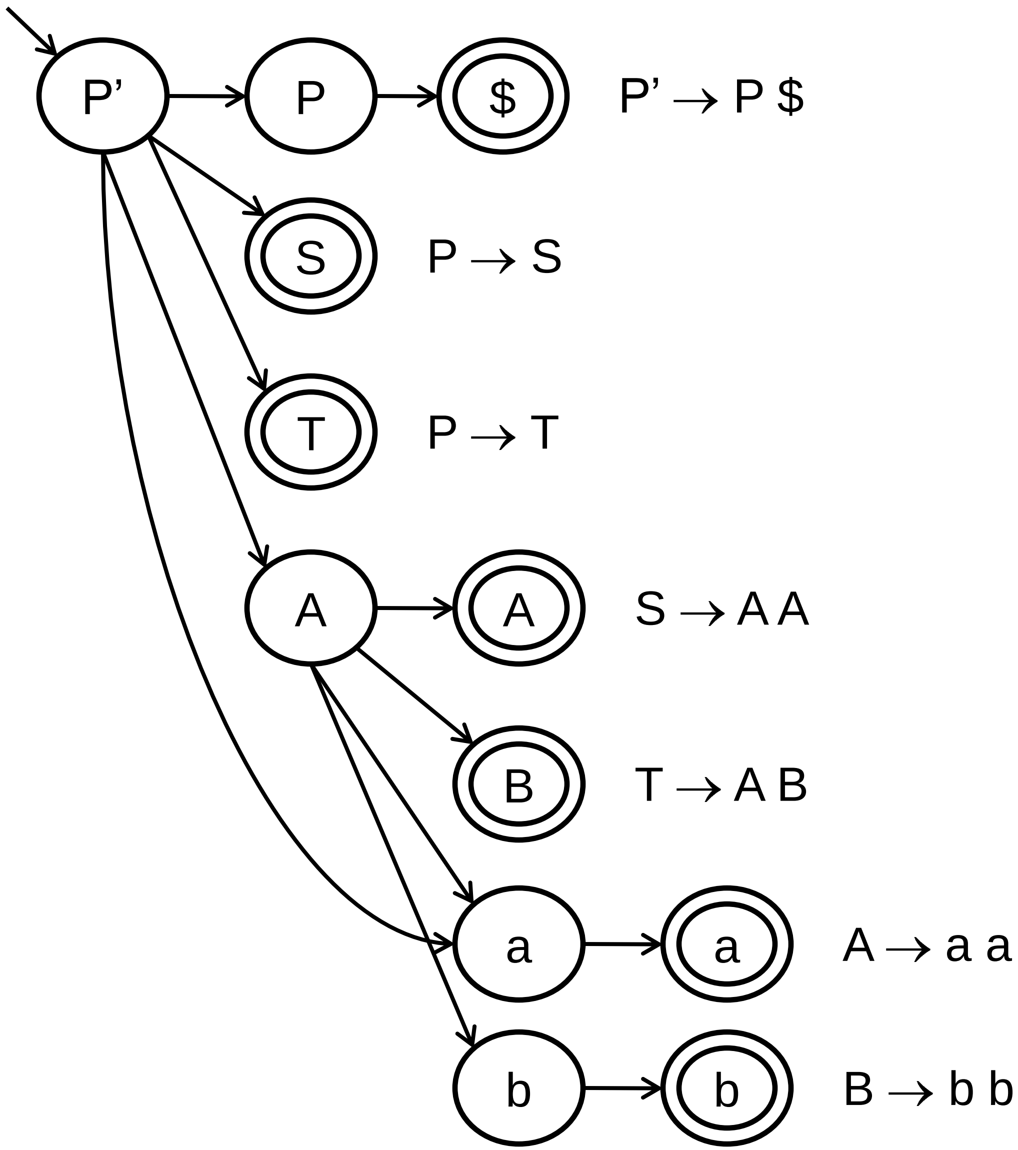
$S \rightarrow A\ A$

$T \rightarrow A\ B$

$A \rightarrow a\ a$

$B \rightarrow b\ b$

Example



$P' \rightarrow P\ \$$

$P \rightarrow S \mid T$

$S \rightarrow A\ A$

$T \rightarrow A\ B$

$A \rightarrow a\ a$

$B \rightarrow b\ b$

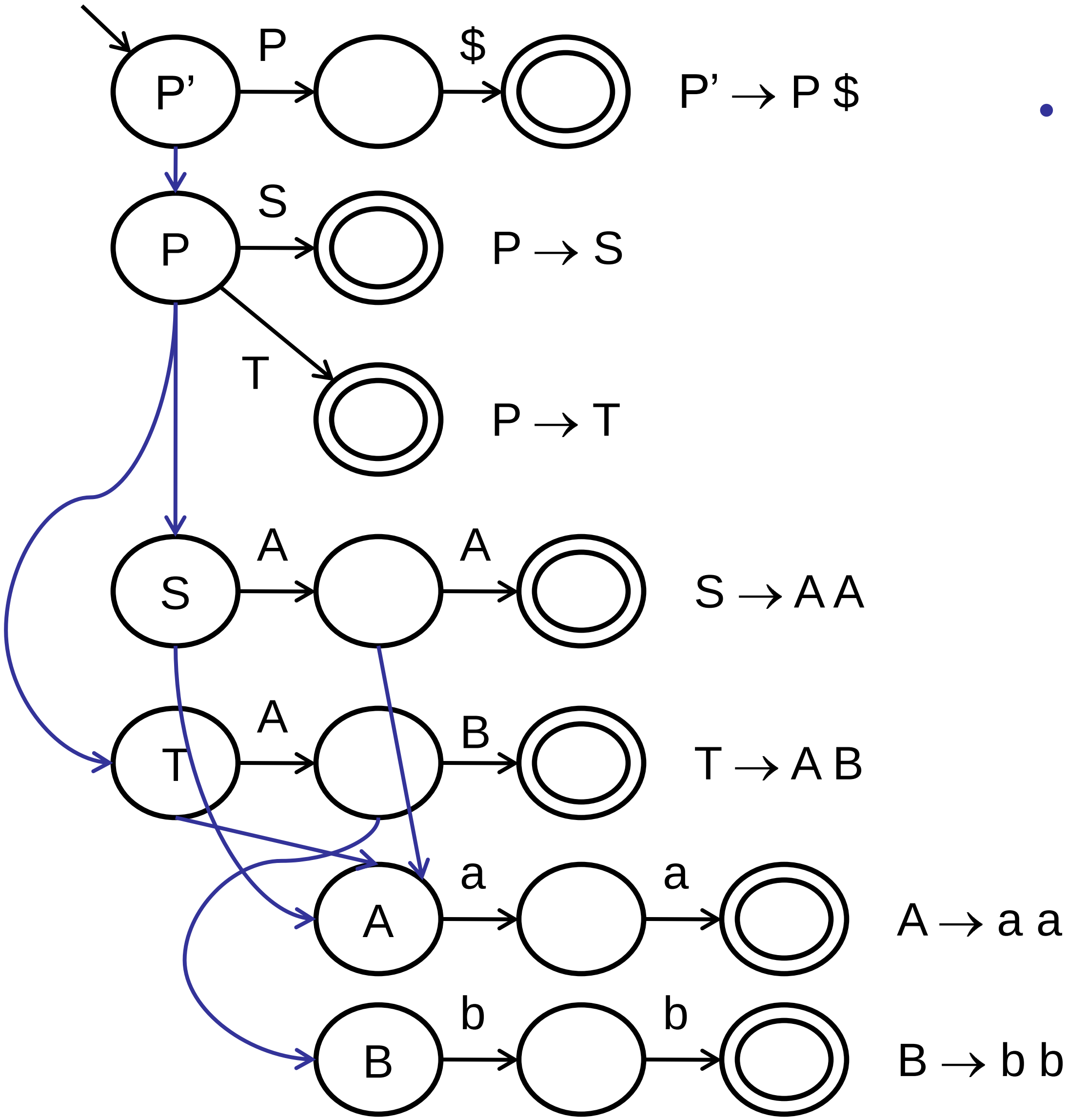
From sets to automaton

- Many of the important properties that we calculated from sets and constraints are encapsulated in the LR(0) automaton
  - Easier to calculate from set definitions
  - Perhaps easier to understand from automaton
- Examples
  - FIRST, FOLLOW, Items

FIRST

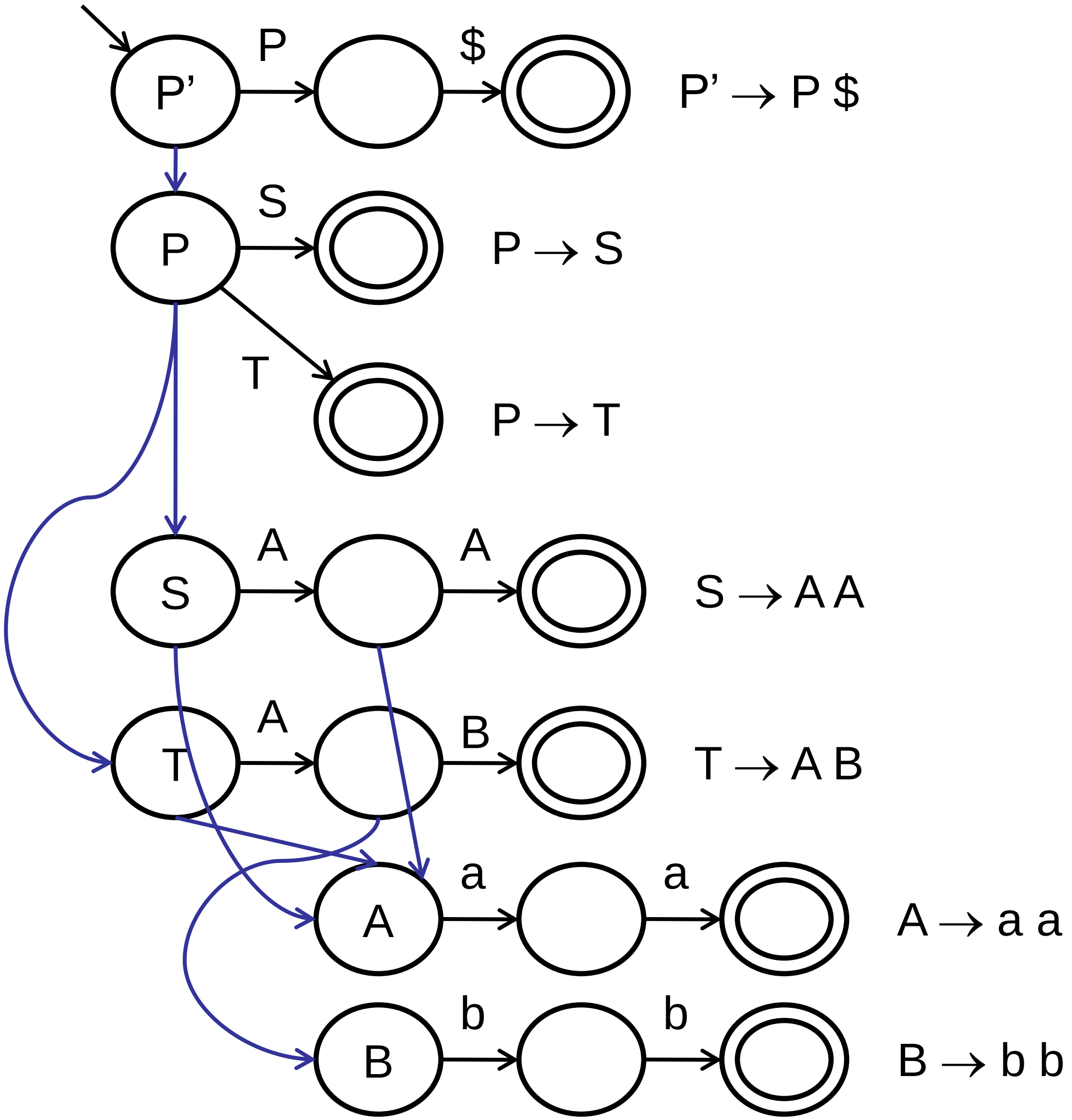
- “Possible first terminals for a non-terminal”
- Case 1
  - For [A $\rightarrow$ a] then a $\subseteq$ FIRST(A)
- Case 2
  - For [A $\rightarrow X_1 X_2 \ldots X_n$] then
    - FIRST(X_1) $\subseteq$ FIRST(A)
    - If NULLABLE(X_1) then FIRST(X_2) $\subseteq$ FIRST(A)
    - If NULLABLE(X_1, X_2, …, X_{n-1}) then FIRST(X_n) $\subseteq$ FIRST(A)
  - For [A $\rightarrow \varepsilon$] then no constraint

FIRST



- "Possible first terminals for a non-terminal"

FIRST

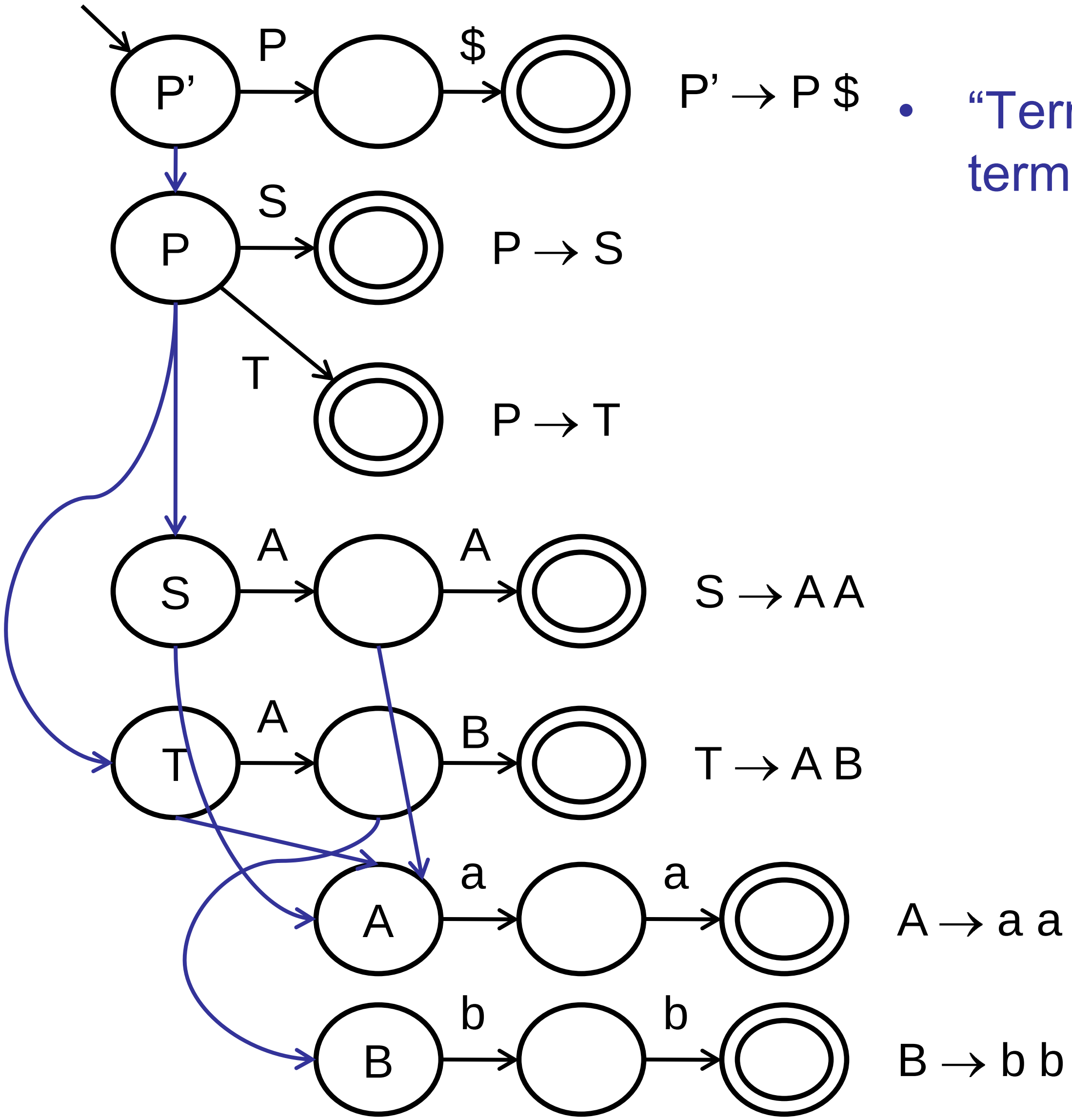


FIRST(P') = { a }
FIRST(P) = { a }
FIRST(S) = { a }
FIRST(T) = { a }
FIRST(A) = { a }
FIRST(B) = { b }

FOLLOW

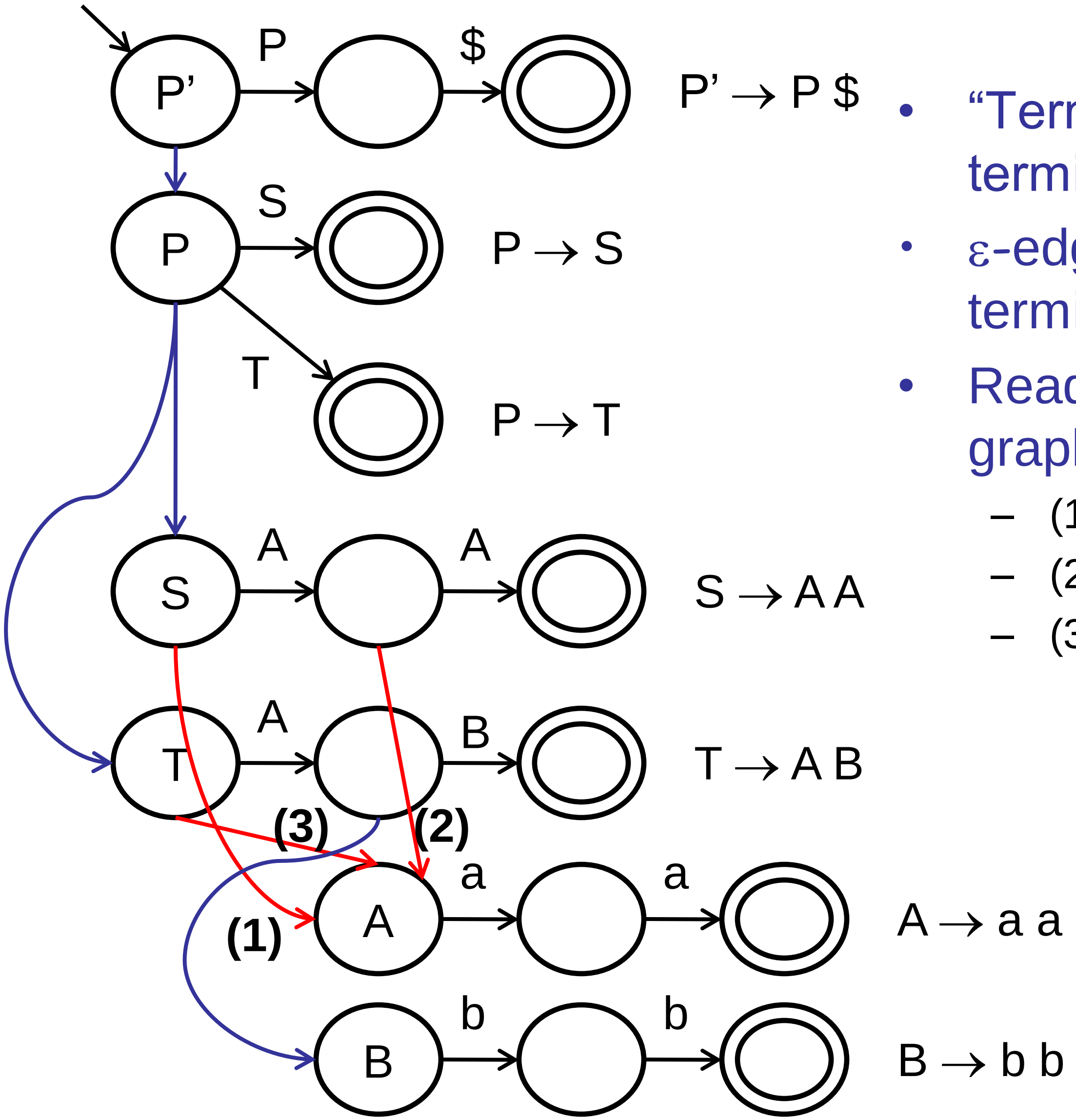
- "Terminals that can follow a non-terminal"
- For [$A \rightarrow \ldots X B_1 B_2 \ldots B_n$]
- Case 1
  - $FIRST(B_1) \subseteq FOLLOW(X)$
- Case 2
  - If $NULLABLE(B_1)$ then $FIRST(B_2) \subseteq FOLLOW(X)$
  - If $NULLABLE(B_1, B_2, \ldots, B_{n-1})$ then $FIRST(B_n) \subseteq FOLLOW(X)$
- Case 3
  - If $NULLABLE(B_1, B_2, \ldots, B_n)$ then $FOLLOW(A) \subseteq FOLLOW(X)$
  - $NULLABLE(\{\ \}) \triangleq true$

FOLLOW



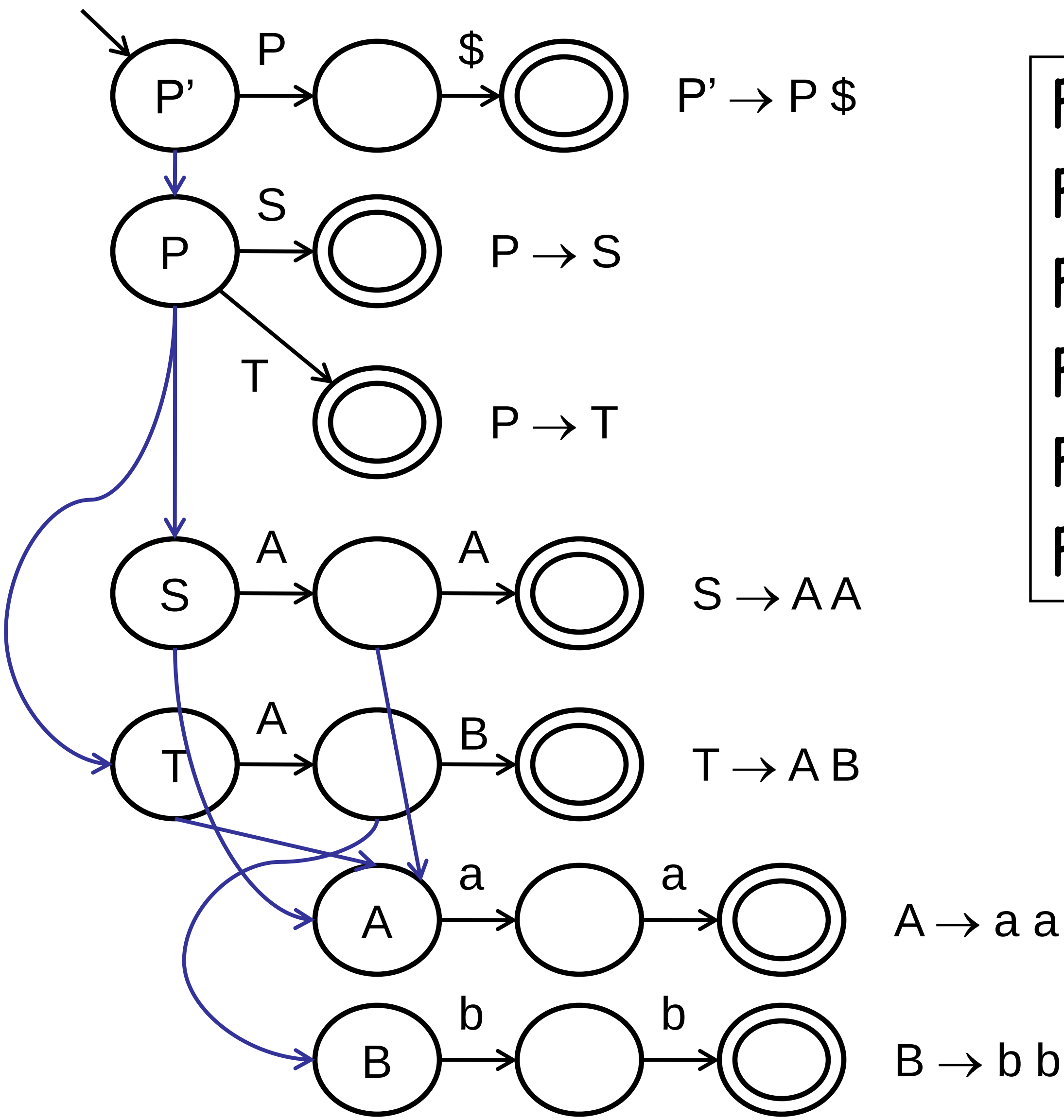
- “Terminals that can follow a non-terminal”

FOLLOW



- “Terminals that can follow a non-terminal”
- ε-edges show where non-terminals are used
- Read FOLLOW constraints from graph
  - (1) FIRST(A) ⊆ FOLLOW(A)
  - (2) FOLLOW(S) ⊆ FOLLOW(A)
  - (3) FIRST(B) ⊆ FOLLOW(A)

FOLLOW



FOLLOW(P') = { }

FOLLOW(P) = { $ }

FOLLOW(S) = { $ }

FOLLOW(T) = { $ }

FOLLOW(A) = { a, b, $ }

FOLLOW(B) = { a, $ }

LR(0) items

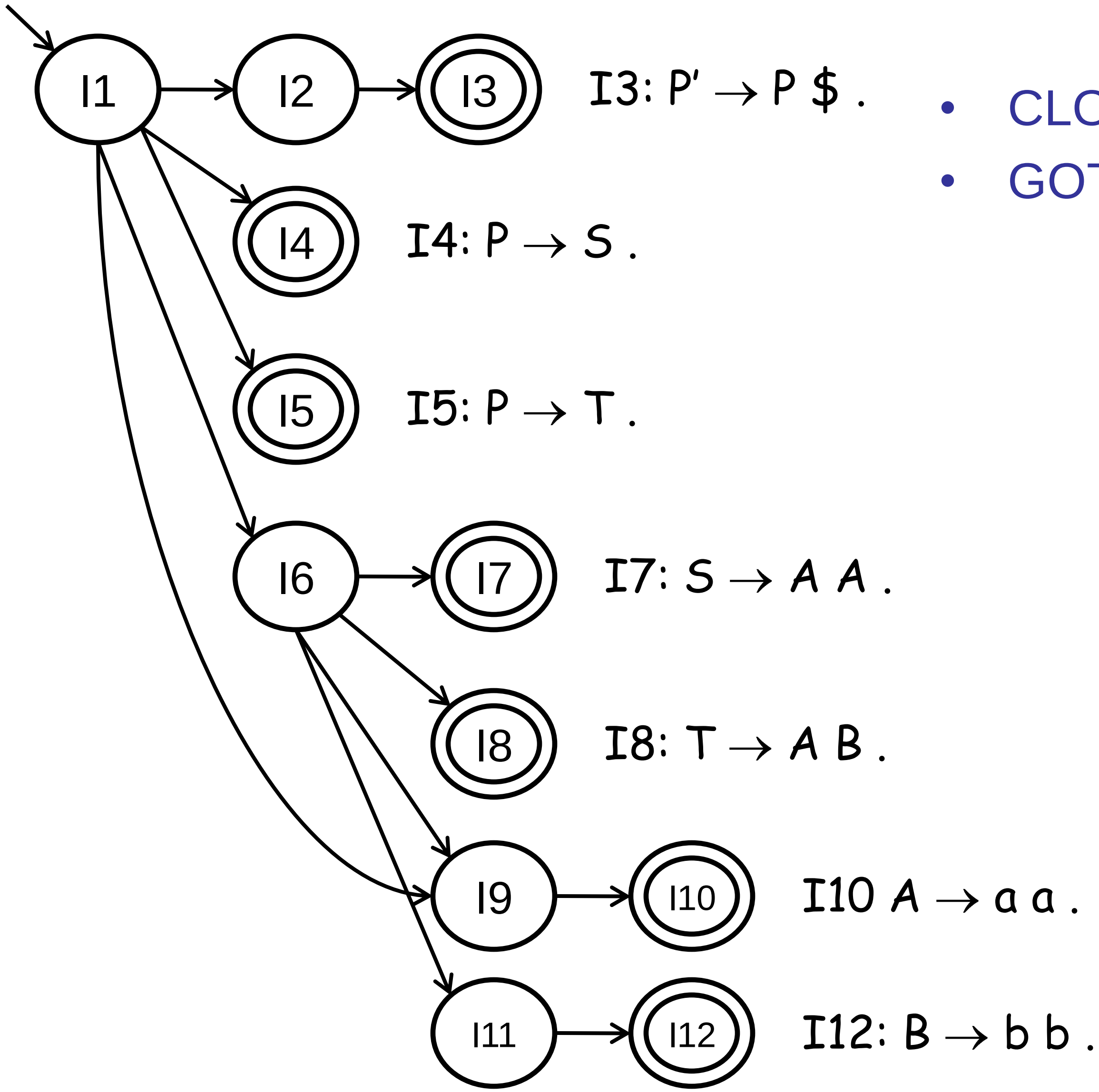
- A LR(0) item is
  - A production, $A \rightarrow X\ Y\ Z$, with a dot in the body, e.g., $A \rightarrow .\ X\ Y\ Z$
  - Represents state of parser
    - $A \rightarrow X\ .\ Y\ Z$ means that the parser has seen X so far and is looking for a string derivable from Y Z
- Alternate construction of LR(0) automaton

LR(0) items

- CLOSURE(I : item set)
  - I $\subseteq$ CLOSURE(I)
  - If
    - [A $\rightarrow$ α . B β] $\in$ CLOSURE(I) and
    - [B $\rightarrow$ γ]
  - Then
    - [B$\rightarrow$. γ] $\in$ CLOSURE(I)
  - "If we're looking for B β and B $\rightarrow$ γ then we should also be looking for γ"
- GOTO(I, X : symbol)
  - If [A $\rightarrow$ α . X β] $\in$ I then [A $\rightarrow$ α X . β] $\in$ GOTO(I,X)
  - CLOSURE(GOTO(I, X)) $\subseteq$ GOTO(I, X)
  - "If we're in state I and see symbol X, we are now in state I'"

LR(0) items

- CLOSURE defines states
- GOTO defines transitions



I1: $P' \to .\ P\ \$$
$P \to .\ S$
$P \to .\ T$
$P \to .\ A$
$P \to .\ a$

I2: $P' \to P\ .\ \$$

I6: $S \to A\ .\ A$
$T \to A\ .\ B$

I9: $A \to a\ .\ A$

I11: $B \to b\ .\ b$

Recap

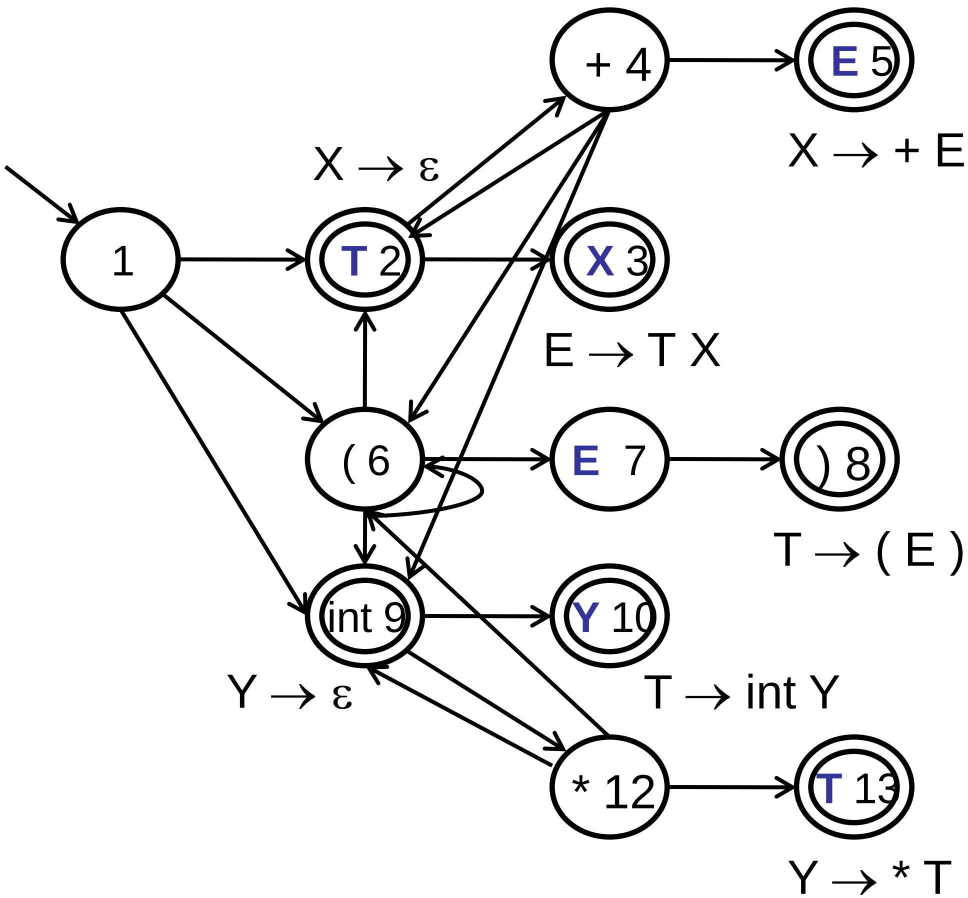
- Equivalence between LR(0) automaton properties and set constraints
  - Set constraints may be easier to calculate
  - FIRST(A): The set of terminals reachable from A through ε-moves
  - FOLLOW(A): For each incoming ε-edge to non-terminal, either FIRST(B) $\subseteq$ FOLLOW(A) or FOLLOW(B) $\subseteq$ FOLLOW(A) depending on incident state
  - LR(0) items ⇔ LR(0) automaton

How do we build transition tables?

- LR(0) automata encapsulate all we need
  - Push-down automata with edges labeled with terminals and non-terminals
  - Reducing and accepting states
- Now what about the transition tables?

SLR(1) parser

- Simple LR(1) parser



E → T X

T → (E) | int Y

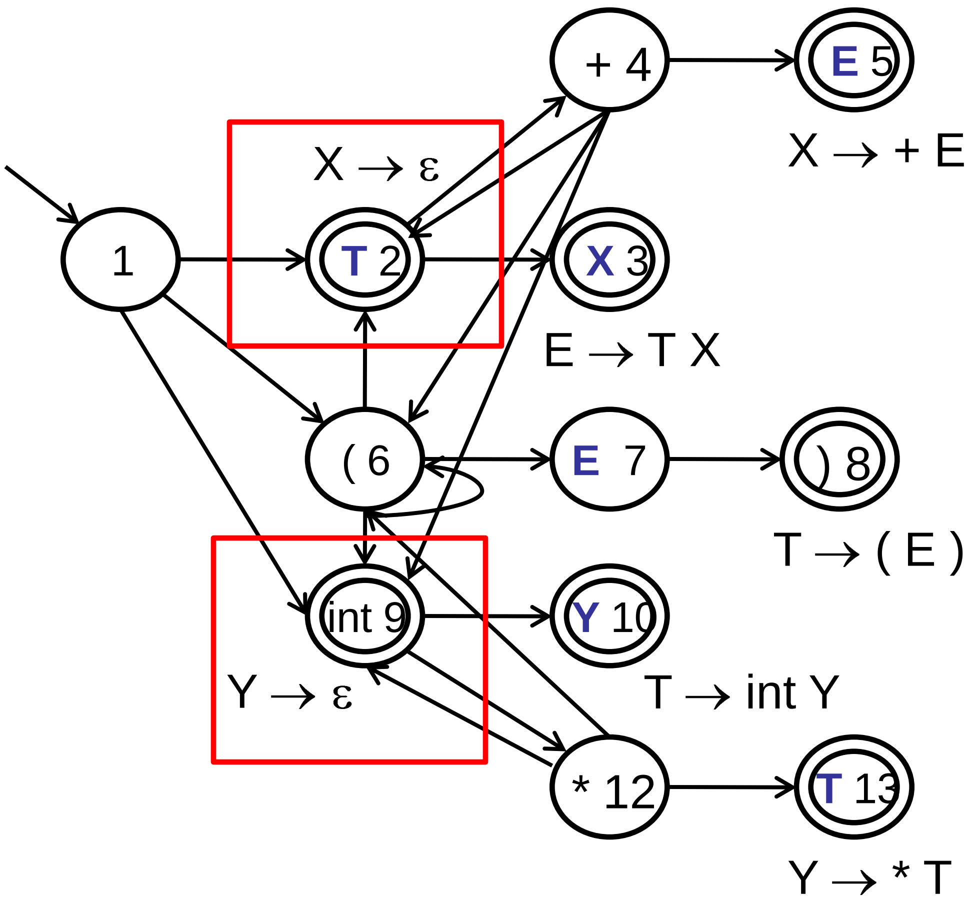
X → + E | ε

Y → * T | ε

SLR(1) parser

- When to apply ε-reductions?

$E \to T\ X$
$T \to (\ E\ )\ |\ \text{int}\ Y$
$X \to +\ E\ |\ \varepsilon$
$Y \to *\ T\ |\ \varepsilon$



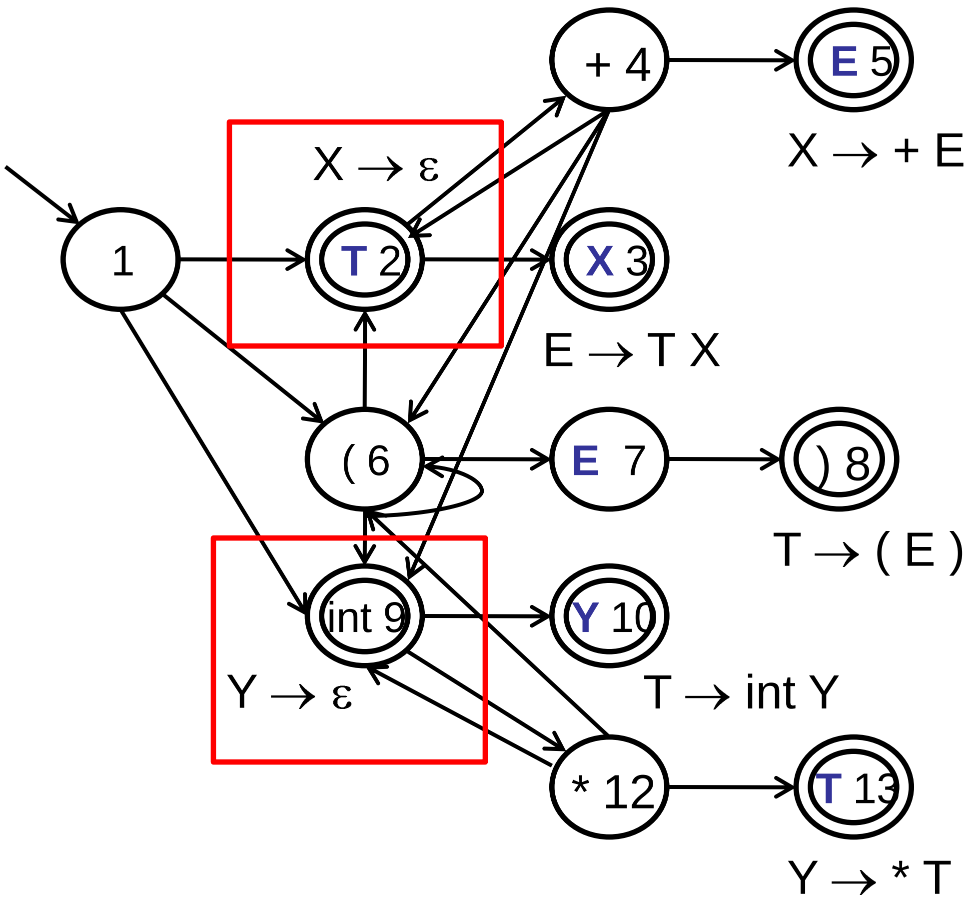
SLR(1) parser take 1

- Always reduce?
- Generate Action table from automaton
  - For each edge $S_i \overset{a}{==>} S_j$ in LR(0) automaton, Action[S_i,a] = shift S_j
  - For each "reduce" node S_i with reduction [$A \rightarrow \beta$], Action[S_i,Σ] = reduce $A \rightarrow \beta$
    - Exception: If the node corresponds to the reduction $S' \rightarrow S$, then Action[S_i, \$] = accept
  - All other actions are error
  - Conflict between actions ➔ grammar not SLR

SLR(1) parser

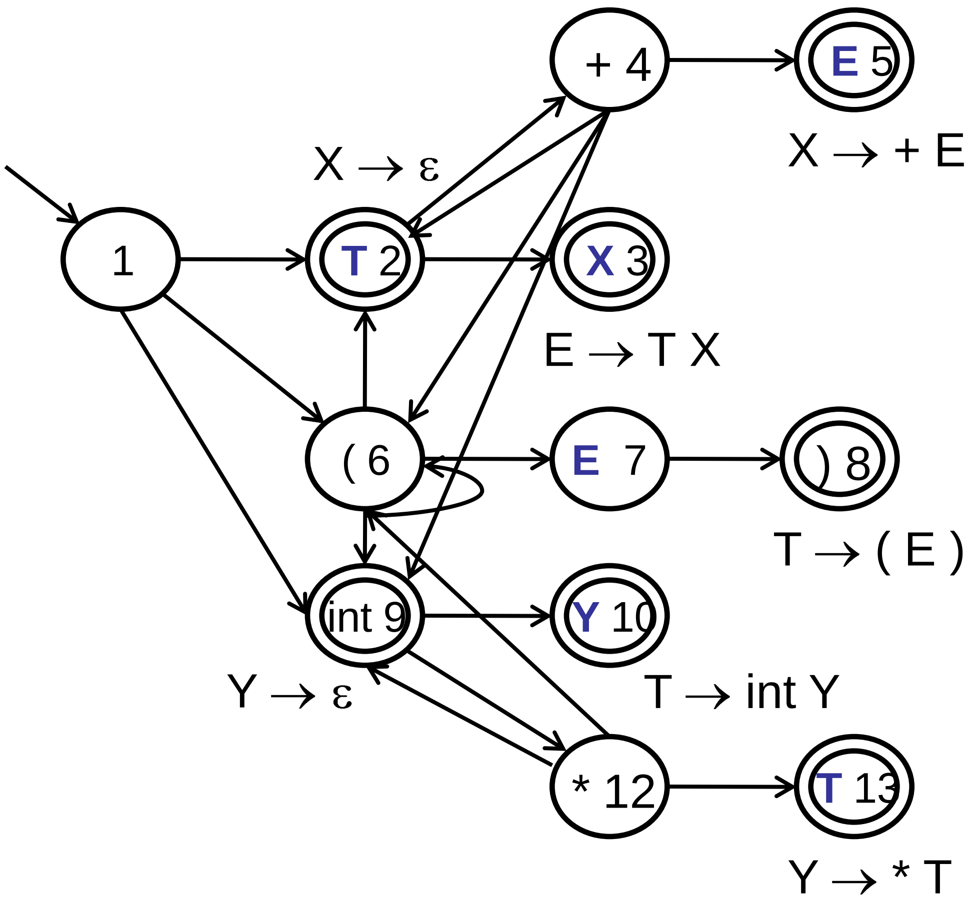
- When to apply ε-reductions?

E → T X
T → (E) | int Y
X → + E | ε
Y → * T | ε



SLR(1) parser

- When to apply ε-reductions?
  - When current token is in FOLLOW set



FOLLOW(X) = {), $ }
FOLLOW(Y) = { +,), $ }

E → T X
T → (E) | int Y
X → + E | ε
Y → * T | ε

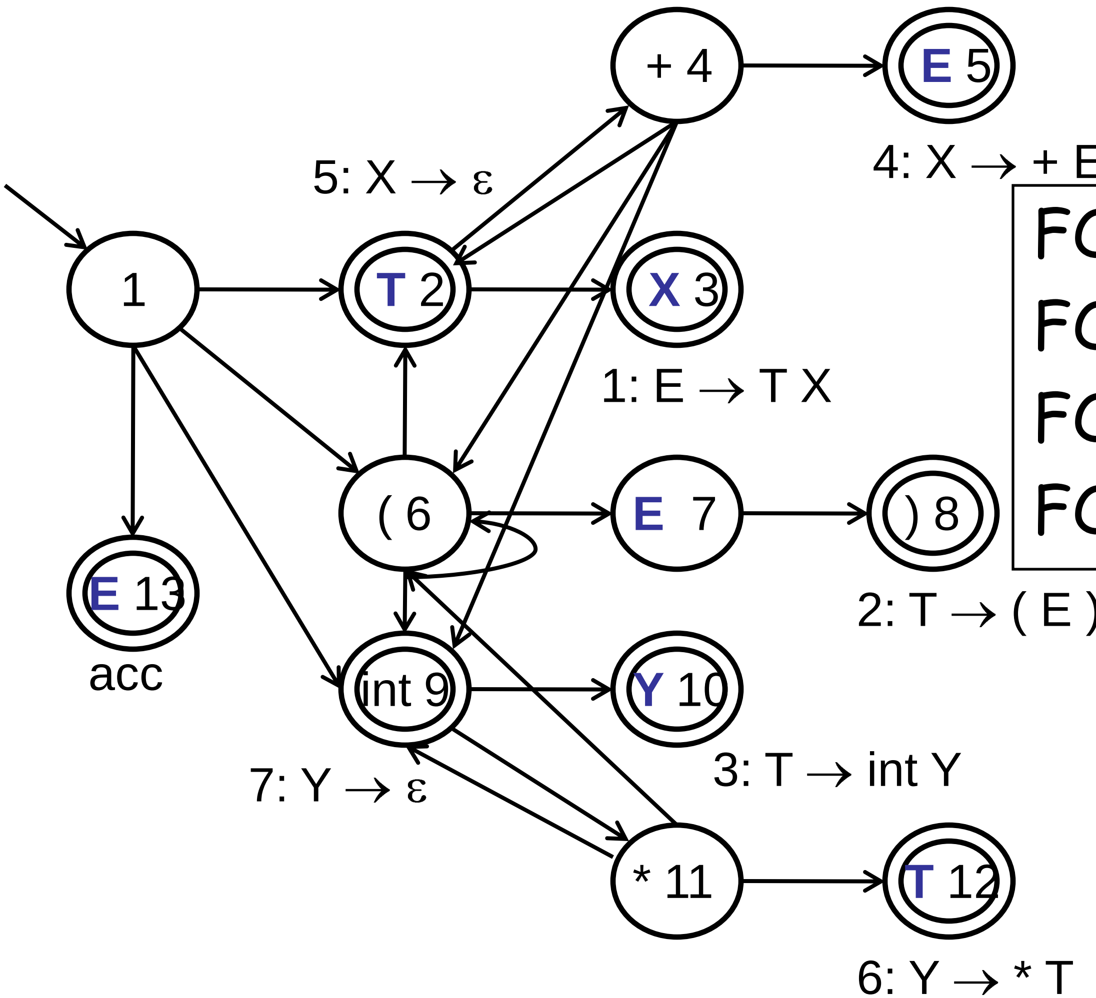
SLR(1) parser take 2

- Generate Action table from automaton
  - For each edge $S_i \overset{a}{\Longrightarrow} S_j$ in LR(0) automaton, Action[S_i,a] = shift S_j
  - For each "reduce" node S_i with reduction [$A \rightarrow \beta$], Action[S_i,a] = reduce $A \rightarrow \beta$ where $a \in$ FOLLOW(A)
    - Exception: If the node corresponds to the reduction $S' \rightarrow S$, Action[S_i, \$] = accept
  - All other actions are error
  - Conflict between actions ➔ grammar not SLR

SLR(1) parser take 2

- Generate Goto table from automaton
  - For each edge $S_i \xrightarrow{A} S_j$ in LR(0) automaton, Goto$[S_i, A] = S_j$

SLR(1) parsing example



FOLLOW(X) = {), $ }

FOLLOW(Y) = { +,), $ }

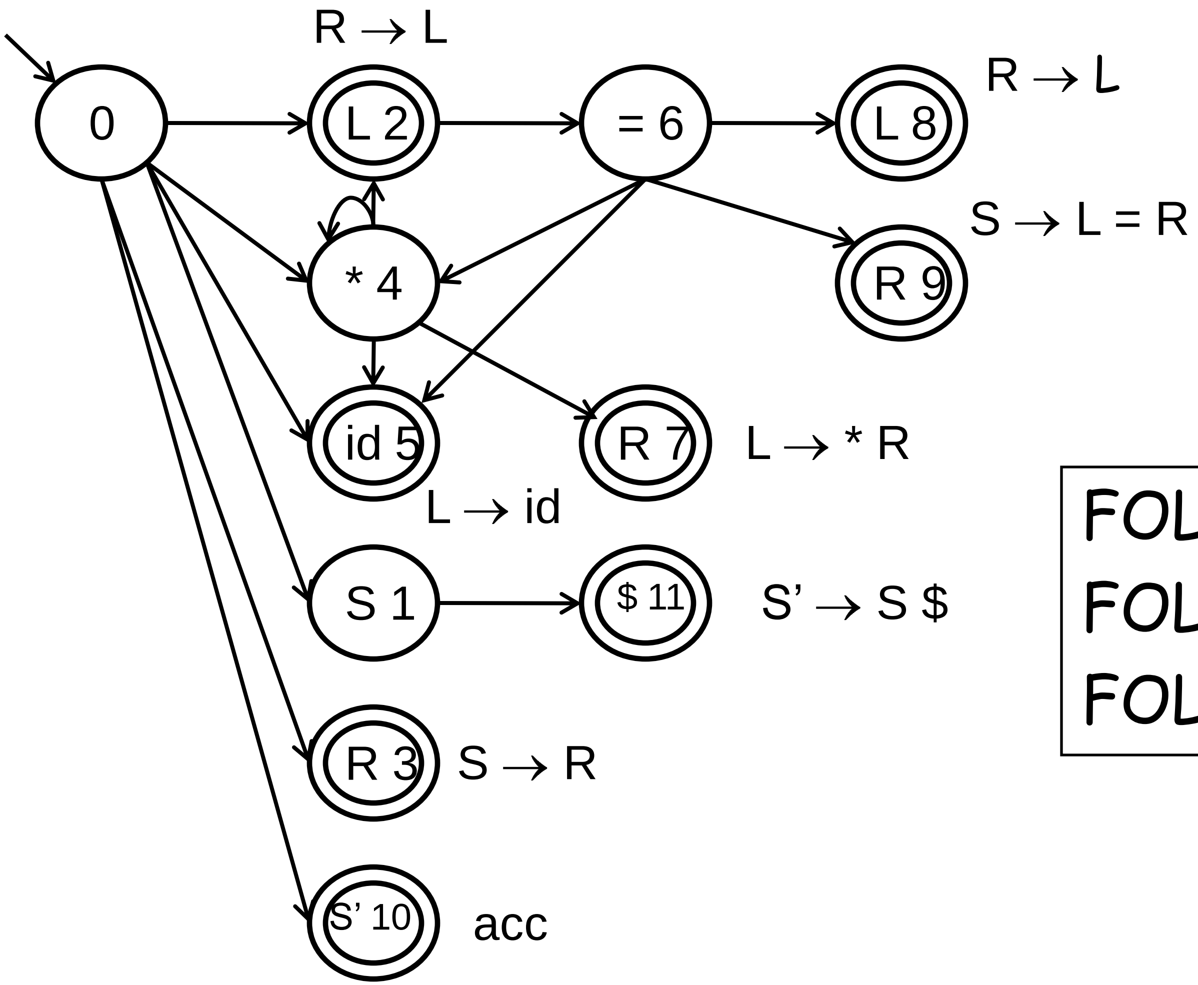
FOLLOW(E) = {), $ }

FOLLOW(T) = { +, $ }

SLR(1) parsing example

State	Action						Goto			
	int	(	)	+	*	$	E	T	X	Y
1	S9	S6					13	2		
2			R5	S4		R5			3	
3			R1			R1				
4	S9	S6					5	2		
5			R4			R4				
6	S9	S6					7	2		
7			S8							
8				R2		R2				
9			R7	R7	S11	R7				10
10				R3		R3				
11	S9	S6						12		
12			R6	R6		R6				
13						acc				

Problem with SLR(1)



$S' \rightarrow S\ \$$

$S \rightarrow L = R \mid R$

$L \rightarrow *R \mid id$

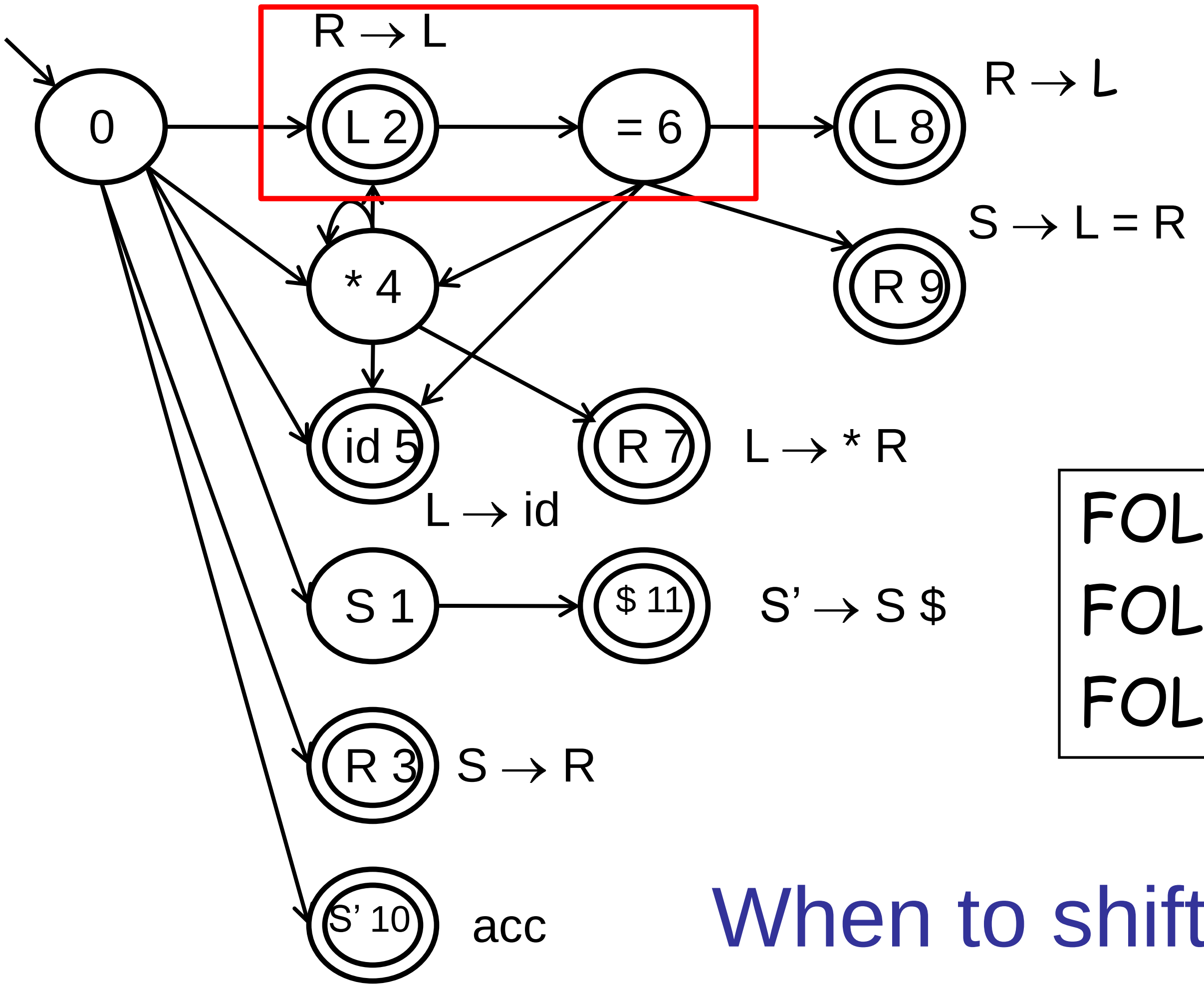
$R \rightarrow L$

FOLLOW(S) = { $ }

FOLLOW(R) = { =, $ }

FOLLOW(L) = { =, $ }

Problem with SLR(1)



$S' \to S\ \$$

$S \to L = R \mid R$

$L \to {*}R \mid id$

$R \to L$

FOLLOW(S) = { $ }

FOLLOW(R) = { =, $ }

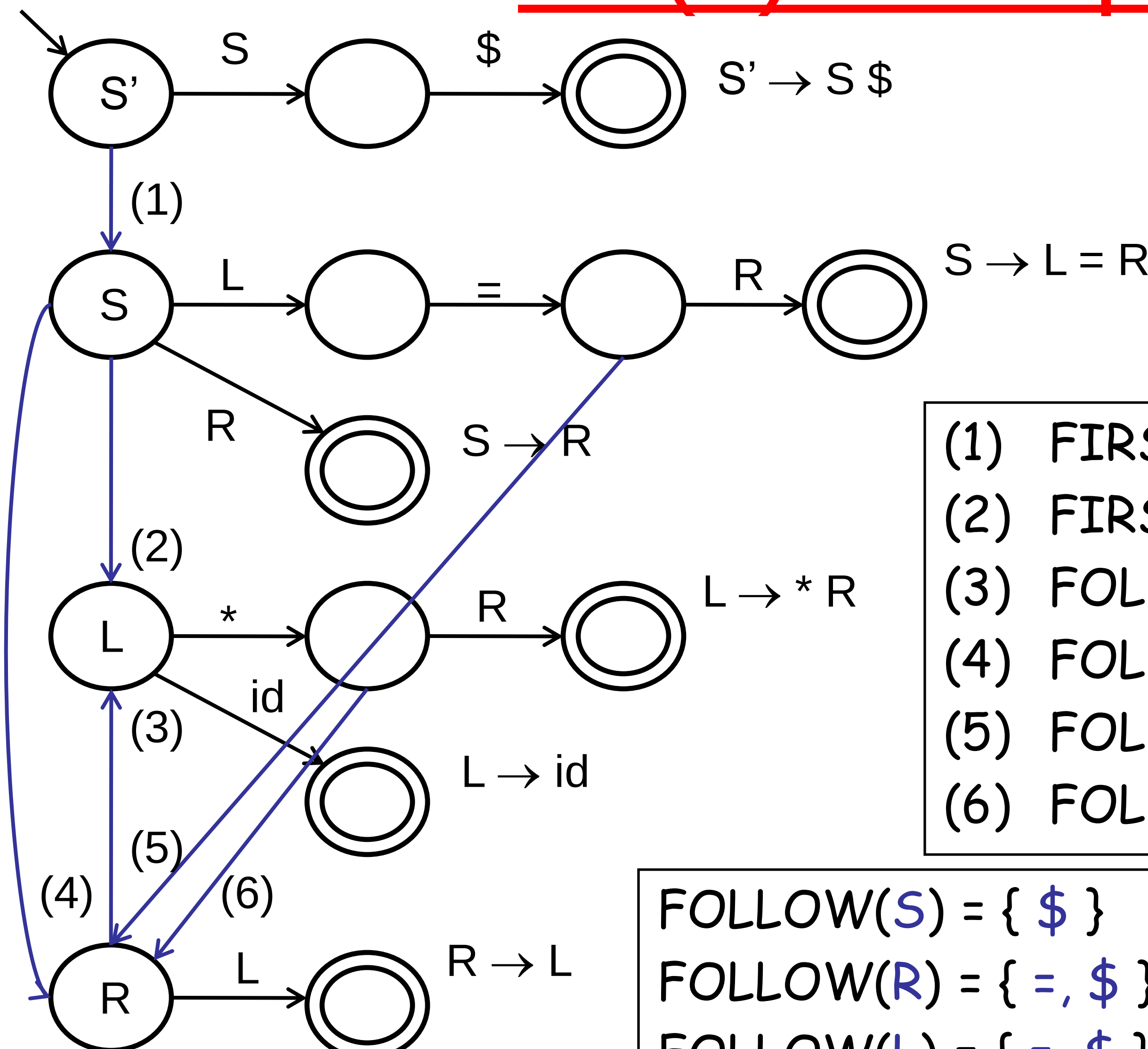
FOLLOW(L) = { =, $ }

When to shift and when to reduce?

LR(1)

- Use k = 1 lookahead symbols to determine when to shift rather than reduce
  - Reduce only when we have a matching lookahead
  - The set of lookahead symbols for A is some subset of FOLLOW(A)
- Use LR(0) automaton to give intuition about LR(1)

LR(1) example



S' → S $
S → L = R | R
L → *R | id
R → L

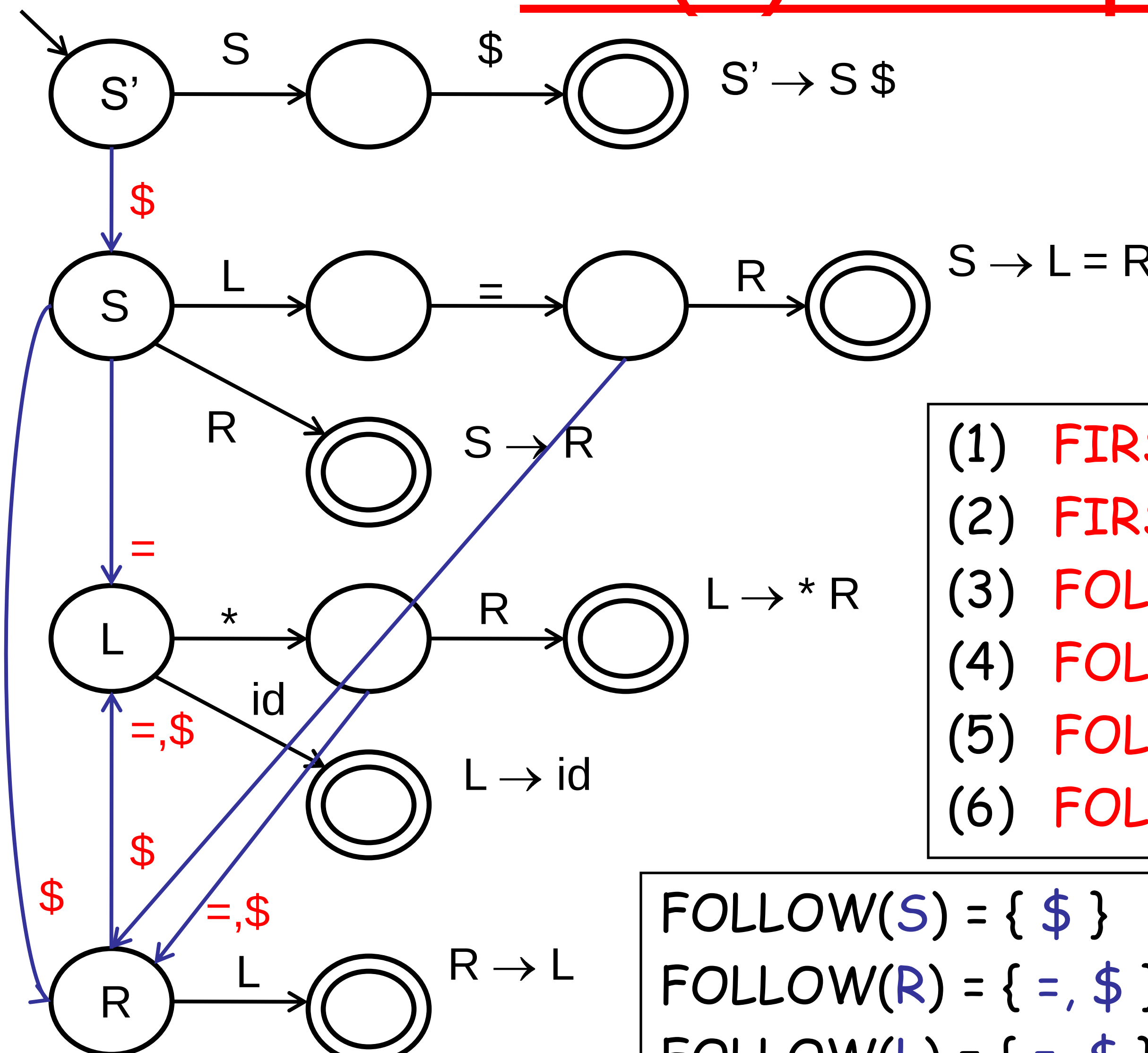
(1) FIRST($) ⊆ FOLLOW(S)
(2) FIRST(=) ⊆ FOLLOW(L)
(3) FOLLOW(R) ⊆ FOLLOW(L)
(4) FOLLOW(S) ⊆ FOLLOW(R)
(5) FOLLOW(S) ⊆ FOLLOW(R)
(6) FOLLOW(L) ⊆ FOLLOW(R)

Constraints

FOLLOW(S) = { $ }
FOLLOW(R) = { =, $ }
FOLLOW(L) = { =, $ }

Solutions

LR(1) example



S' → S $
S → L = R | R
L → *R | id
R → L

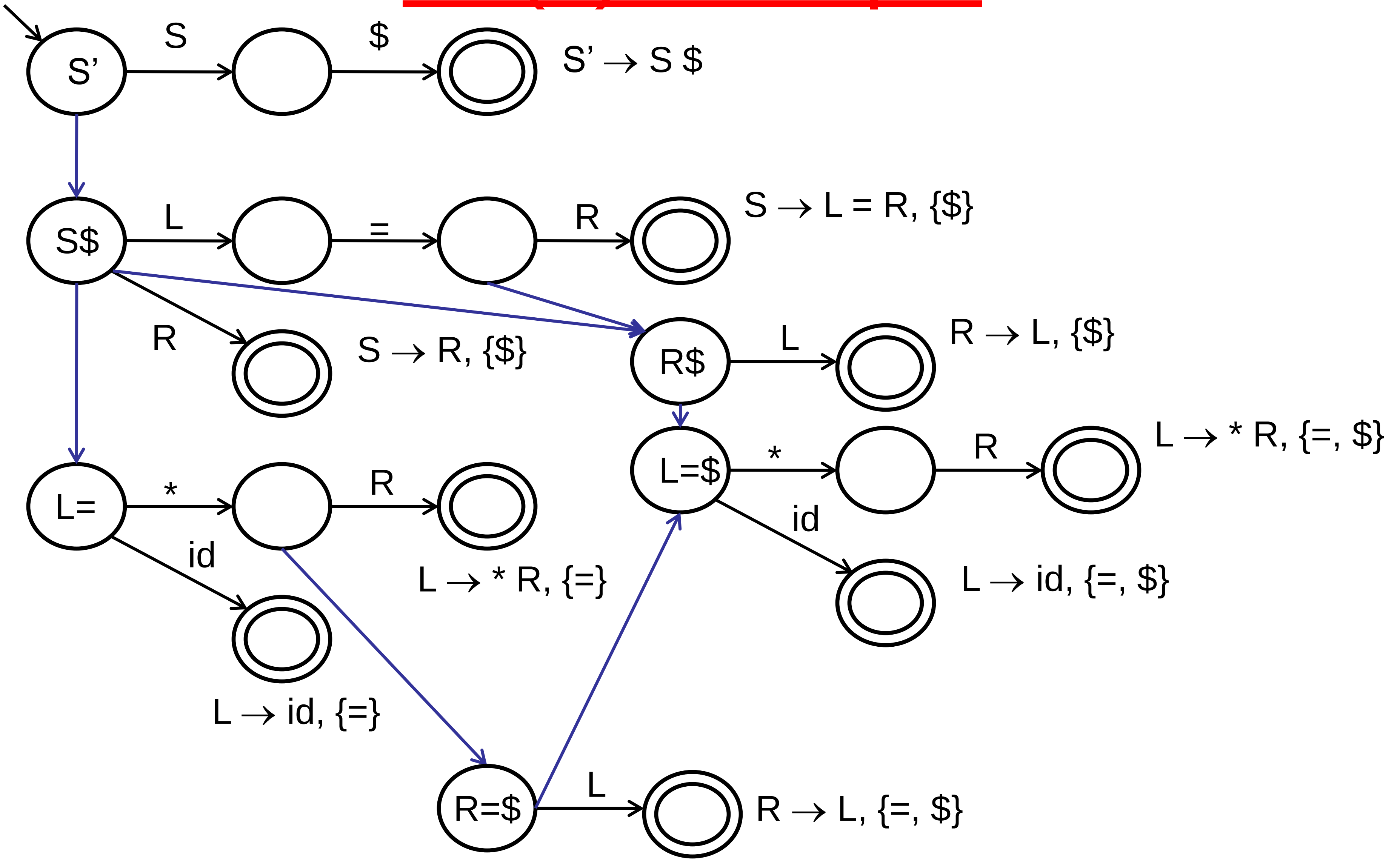
(1) FIRST($) ⊆ FOLLOW(S)
(2) FIRST(=) ⊆ FOLLOW(L)
(3) FOLLOW(R) ⊆ FOLLOW(L)
(4) FOLLOW(S) ⊆ FOLLOW(R)
(5) FOLLOW(S) ⊆ FOLLOW(R)
(6) FOLLOW(L) ⊆ FOLLOW(R)

Constraints

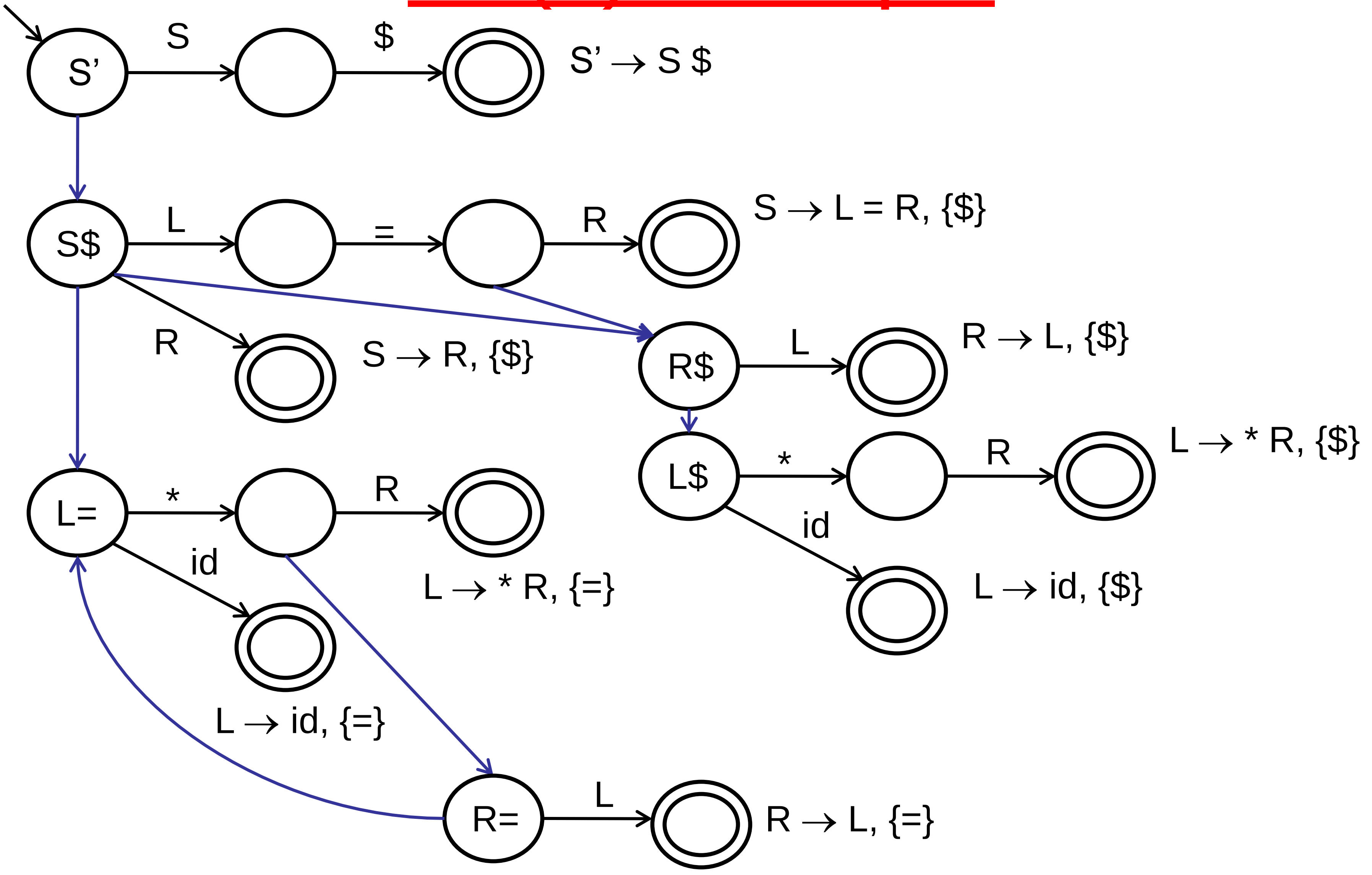
FOLLOW(S) = { $ }
FOLLOW(R) = { =, $ }
FOLLOW(L) = { =, $ }

Solutions

LR(1) example



LR(1) example



Recap

1. Use "context" of ε-moves to introduce states corresponding to the terminal(s) we expect to see after non-terminal
   – State dependent FOLLOW
   – Subset of FOLLOW
2. Propagate lookahead to reduction rules
3. Perform NFA to DFA conversion

LR(1) items

- Equivalence between LR(1) automaton and LR(1) item sets
- A LR(1) item is
  - An LR(0) item augmented with a lookahead symbol (terminal), e.g., [A $\rightarrow$. X Y Z, a]
  - The item [A $\rightarrow$ X Y Z ., a] calls for a reduction only if the next input symbol is a

LR(1) items

- CLOSURE(I : item set)
  - $I \subseteq$ CLOSURE(I)
  - If
    - [$A \to \alpha\ .\ B\ \beta$, a] $\in$ CLOSURE(I),
    - [$B \to \gamma$], and
    - b $\in$ FIRST(β a)
  - Then
    - [$B \to .\ \gamma$, b] $\in$ CLOSURE(I)
  - "If we're looking for B β and $B \to \gamma$ then we should also be looking for γ"
- GOTO(I, X : symbol)
  - If [$A \to \alpha\ .\ X\ \beta$, a] $\in$ I then [$A \to \alpha\ X\ .\ \beta$, a] $\in$ GOTO(I,X)
  - CLOSURE(GOTO(I, X)) $\subseteq$ GOTO(I, X)
  - "If we're in state I and see symbol X, we are now in state I'"

Like FOLLOW(B) but takes into account of current production; "a" term handles the case when $\beta = \varepsilon$; FIRST(a) $\subseteq$ FOLLOW(A)

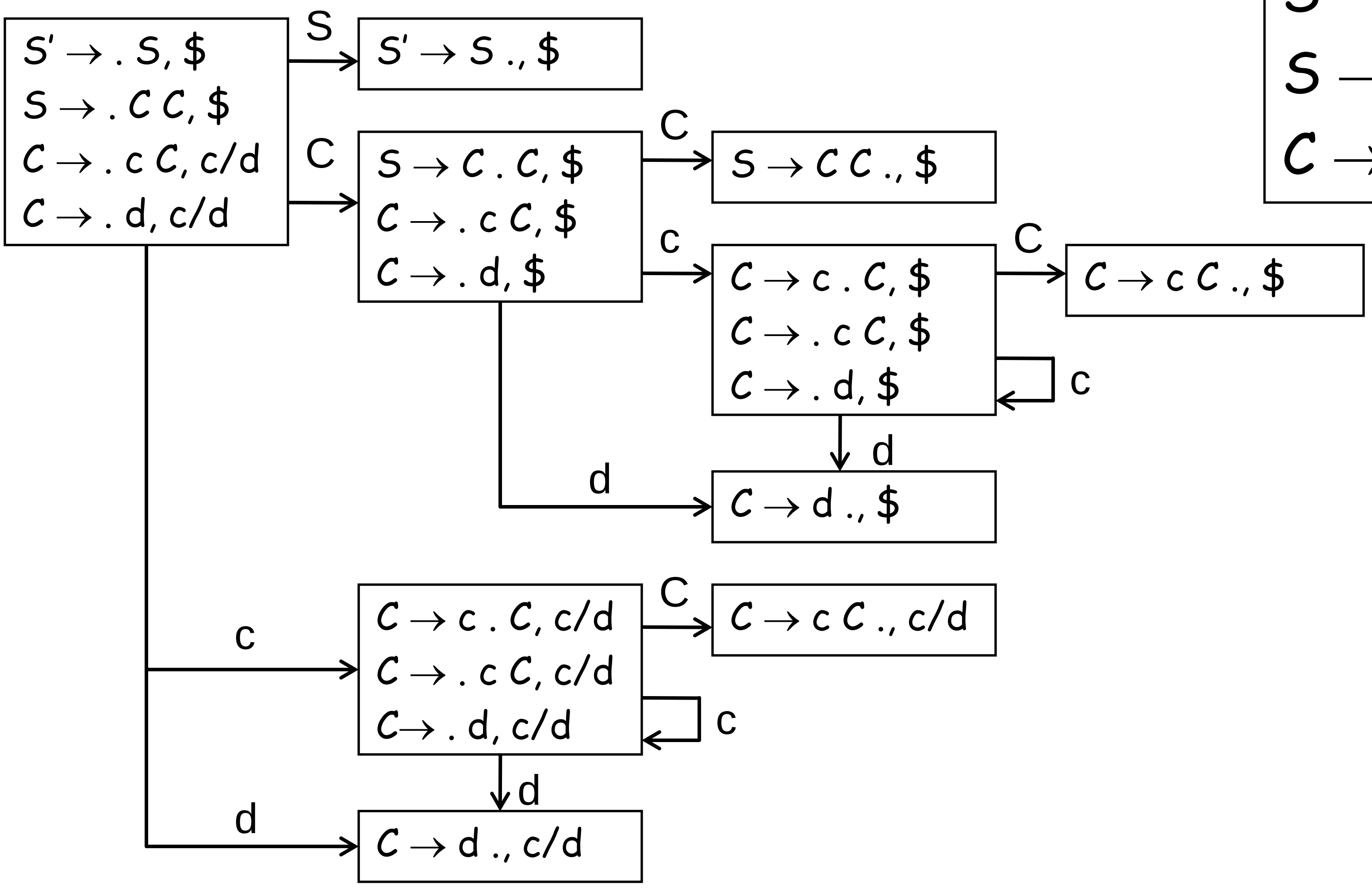
LR(1) parser

- Generate Action table from automaton
  - For each edge $S_i \overset{a}{==>} S_j$ in LR(1) automaton, Action[S_i,a] = shift S_j
  - For each "reduce" node S_i with reduction [$A \rightarrow \beta$, a], Action[S_i,a] = reduce $A \rightarrow \beta$
    - Exception: If the node corresponds to the reduction [$S' \rightarrow S$, \$], then Action[S_i, \$] = accept
  - All other actions are error
  - If there is a conflict between actions, grammar is not in LR(1)
- Goto table generated as in SLR(1)

LALR(1)

- LR(1) construction can generate many more states than SLR(1)
  - Lookahead may only be needed for a few constructs in grammar
  - Merge states that only differ on lookahead symbol (i.e., identical *cores*)
  - Cannot introduce a shift-reduce conflict because shift actions only depend on core
  - May introduce reduce-reduce conflicts

LALR(1) example

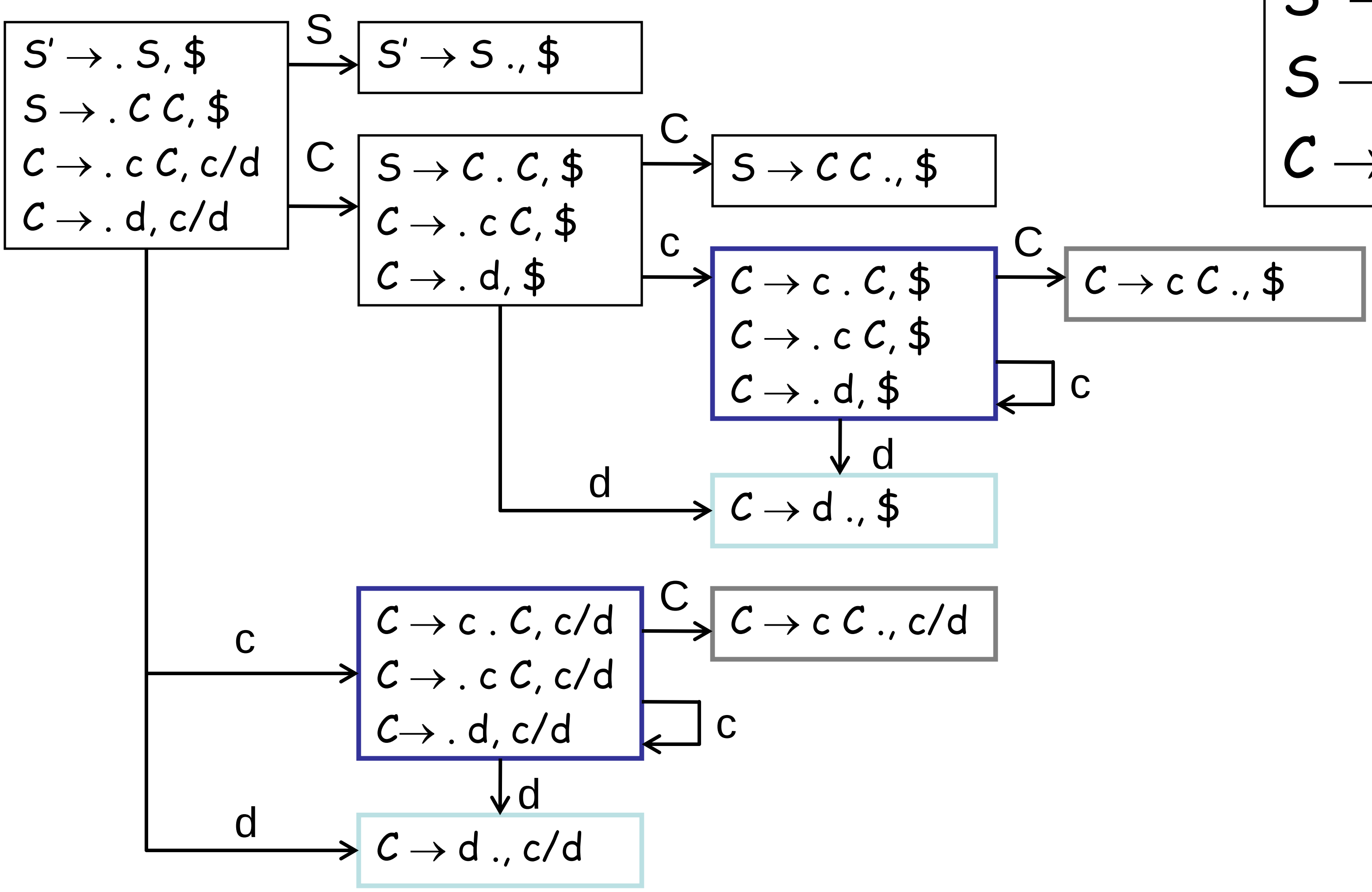


$S' \to S\ \$$

$S \to C\ C$

$C \to c\ C \mid d$

LALR(1) example



$S' \rightarrow S\ \$$

$S \rightarrow C\ C$

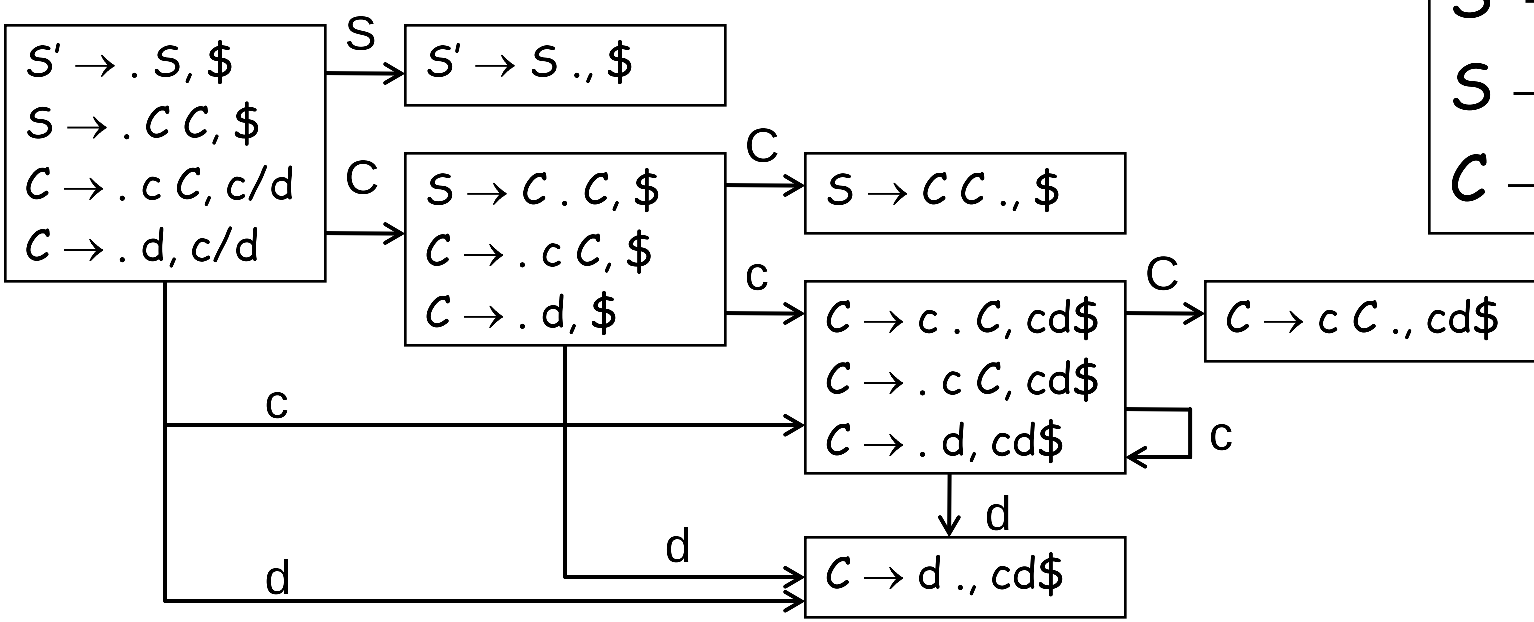
$C \rightarrow c\ C \mid d$

LALR(1) example

$S' \to S\ \$$

$S \to C\ C$

$C \to c\ C \mid d$



Dealing with ambiguity

- Commonly, parser generators allow methods for dealing with ambiguous grammars
  - Precedence and associativity rules for operators
  - Implemented by modifying generated parser table

JFlex and CUP

- JFlex, a lexer generator for Java
- CUP, a parser generator for Java
- Both take specifications and generate Java code

JFlex

```
import java_cup.runtime.Symbol;
%%
%class Lexer
%cup
%{
  private Symbol symbol(int sym) { return new Symbol(sym, yyline+1, yycolumn+1); }
  private Symbol symbol(int sym, Object val) { return new Symbol(sym, val); }
%}
IntLiteral = 0 | [1-9][0-9]*
new_line = \r|\n|\r\n;
white_space = {new_line} | [ \t\f]
%%
{IntLiteral}      { return symbol(sym.INT, new Integer(Integer.parseInt(yytext()))); }
"("               { return symbol(sym.LPAREN); }
")"               { return symbol(sym.RPAREN); }
"+"               { return symbol(sym.PLUS); }
...
{white_space}     { /* ignore */ }
.|\n              { error("Illegal character <"+ yytext()+">"); }
```

CUP

```
/* Terminals (tokens returned by lexer). */
terminal PLUS, MINUS, SLASH, STAR, QUESTION, COLON, LPAREN, RPAREN;
terminal Integer INT;

non terminal Integer Exp;

precedence left QUESTION;
precedence left PLUS, MINUS;
precedence left STAR, SLASH;

Exp ::= INT:i               {: RESULT = i; :}
      | Exp:e1 PLUS Exp:e2  {: RESULT = e1 + e2; :}
      | Exp:e1 MINUS Exp:e2 {: RESULT = e1 - e2; :}
      | Exp:e1 SLASH Exp:e2 {: RESULT = e1 / e2; :}
      | Exp:e1 STAR Exp:e2  {: RESULT = e1 * e2; :}
      | Exp:e1 QUESTION Exp:e2 COLON Exp:e3 {: RESULT = e1 == 0 ? e3 : e2; :}
      | LPAREN Exp:e1 RPAREN {: RESULT = e1; :}
      ;
```