Chapter 6

Bottom-Up Parsing

Bottom-up Parsing

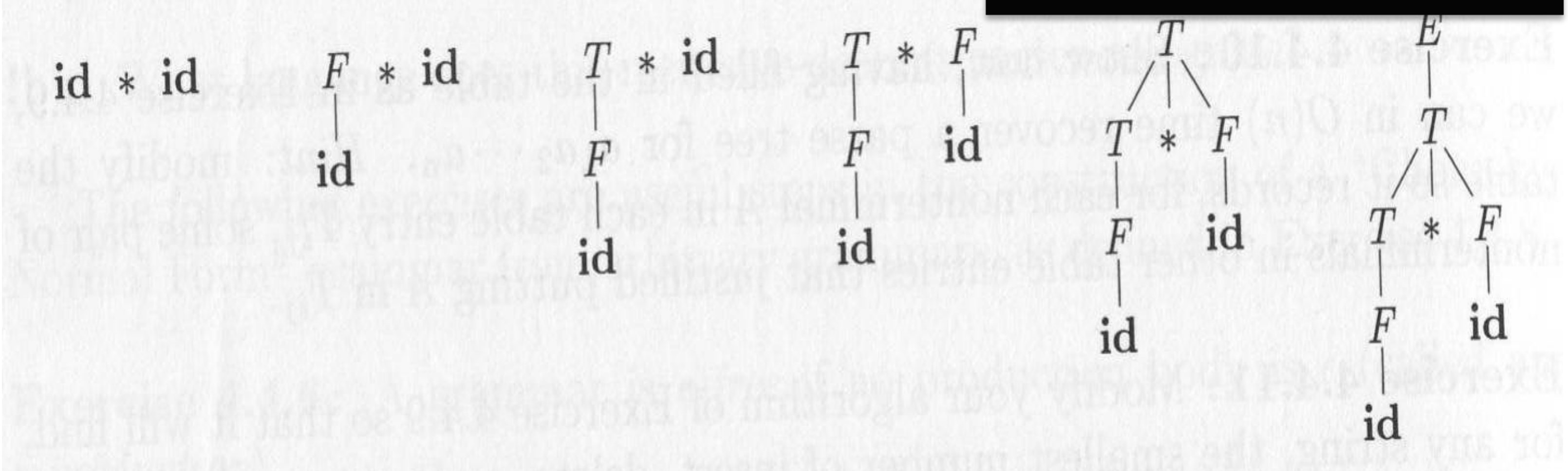
- A bottom-up parsing corresponds to the construction of a parse tree for an input tokens beginning at the leaves (the bottom) and working up towards the root (the top).

- An example follows.

Bottom-up Parsing (Cont.)

- Given the grammar:
  - E → T
  - T → T * F
  - T → F
  - F → id

at this point, a smart parser knows it should not continue to choose E → T to change T to E, but instead, it shift to process id SMARTLY
How can we do that?



Reduction

- The bottom-up parsing as the process of “reducing” a token string to the start symbol of the grammar.

- At each *reduction*, the token string matching the RHS of a production is replaced by the LHS non-terminal of that production.

Reduction (Cont.)

- The key decisions during bottom-up parsing are about when to reduce and about what production to apply.
- Again, just like a top-down parsing, we still have cases that is ambiguous in choosing more than one production rule to reduce

Shift-reduce Parsing

- Shift-reduce parsing is a form of bottom-up parsing in which a stack holds grammar symbols and an input buffer holds the rest of the tokens to be parsed.

- We use $ to mark the bottom of the stack and also the end of the input.

- During a left-to-right scan of the input tokens, the parser shifts zero or more input tokens into the stack, until it is ready to reduce a string β of grammar symbols on top of the stack.

A Shift-reduce Example

STACK	INPUT	ACTION
$\$$	$\mathbf{id}_1 * \mathbf{id}_2 \$$	shift
$\$\,\mathbf{id}_1$	$* \mathbf{id}_2 \$$	reduce by $F \to \mathbf{id}$
$\$\,F$	$* \mathbf{id}_2 \$$	reduce by $T \to F$
$\$\,T$	$* \mathbf{id}_2 \$$	shift
$\$\,T\,*$	$\mathbf{id}_2 \$$	shift
$\$\,T * \mathbf{id}_2$	$\$$	reduce by $F \to \mathbf{id}$
$\$\,T * F$	$\$$	reduce by $T \to T * F$
$\$\,T$	$\$$	reduce by $E \to T$
$\$\,E$	$\$$	accept

$E \to T$

$T \to T * F$

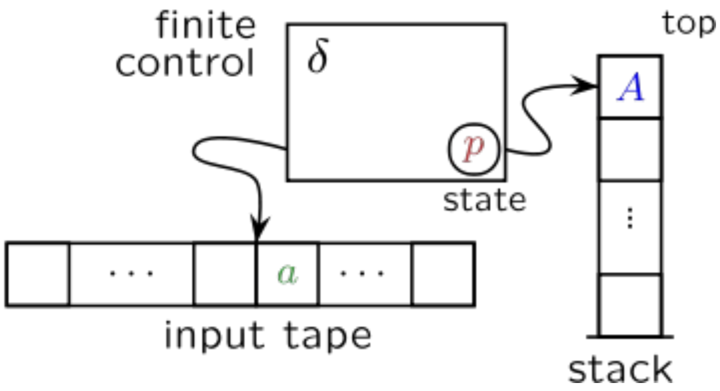
$T \to F$

$F \to id$

OK, here we actually can continue reducing E->T, but it does not, instead it choose a shift smartly.
Why?
Yes, we need to make it predictive and deterministic

NOTES

- As described in CH5, basically this is still a kind of pushdown automata. Remember that CFG and pushdown automata are computationally equivalent



Shift-reduce Parsing (Cont.)

- Shift: shift the next input token onto the top of the stack.
- Reduce: the right end of the string to be reduced must be at the top of the stack. Locate the left end of the string within the stack and decide what non-terminal to replace that string.
- Accept: announce successful completion of parsing.
- Error: discover a syntax error and call an error recovery routine.

LR Parsers

- Left-scan Rightmost derivation in reverse (LR) parsers are characterized by

  the number of ***look-ahead*** *symbols* that are examined to determine *parsing actions*.

- We can make the look-ahead parameter explicit and discuss LR(k) parsers,

  where k is the look-ahead size.

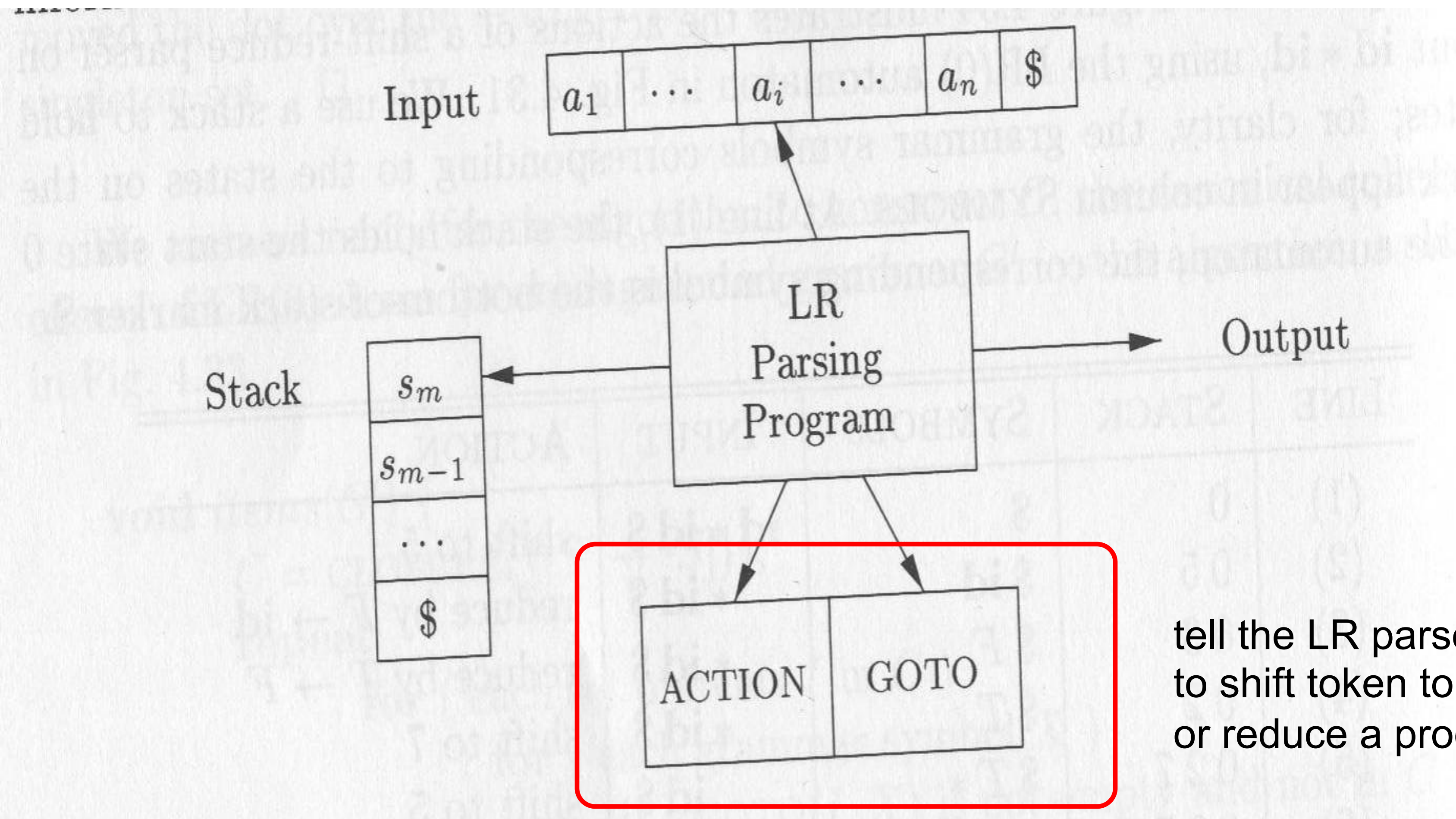
LR(k) Parsers

- LR(k) parsers are of interest in that they are the most *powerful* class of deterministic bottom-up parsers using at most K look-ahead tokens.
- Deterministic parsers must **uniquely determine** the correct parsing action at each step;

  they cannot back up or retry parsing actions (just like we do not want a back tracking LL parser).

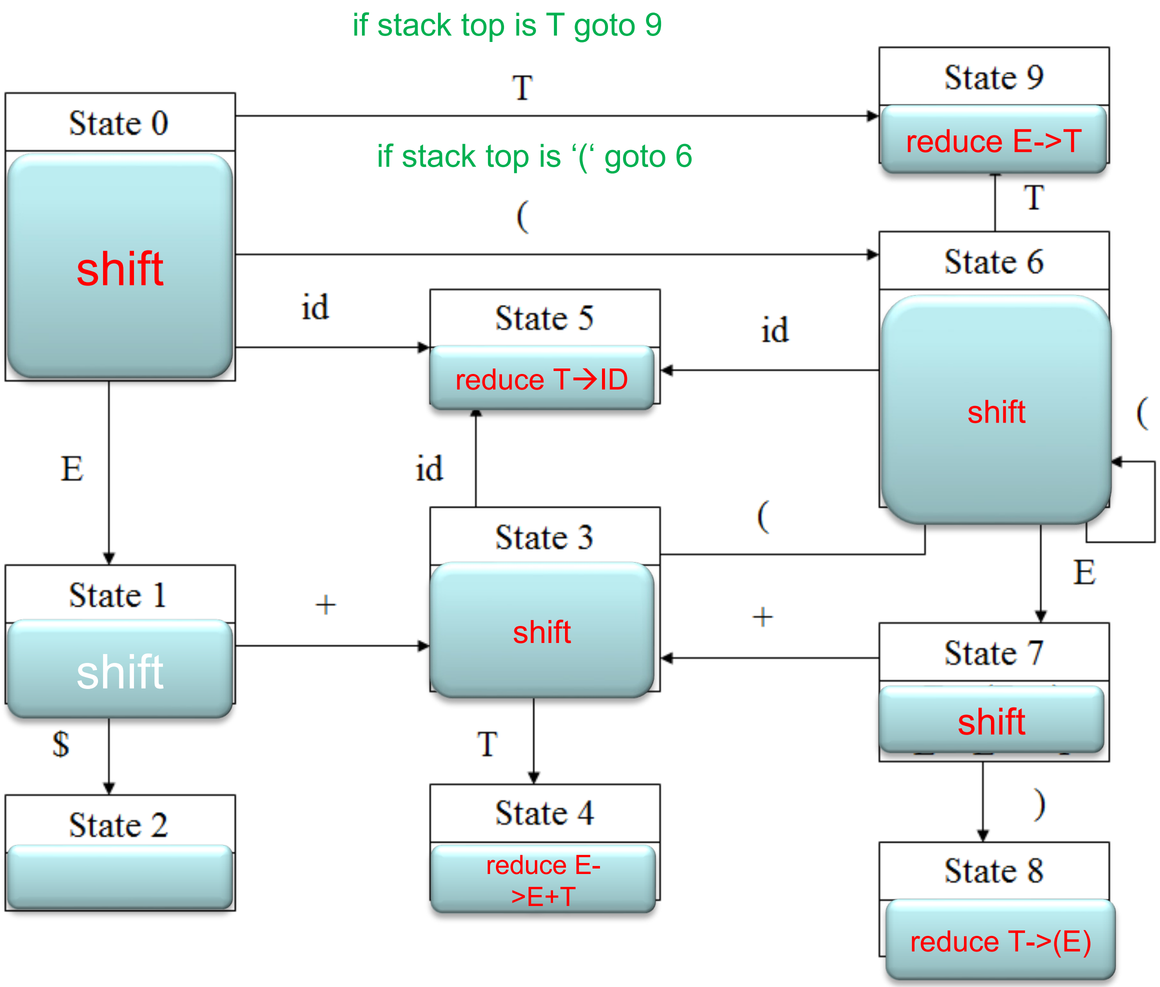
We will cover 4 LR(k) parsers: LR(0), SLR(1), LR(1), and LALR(1) here.

Model of an LR parser



tell the LR parser
to shift token to stack
or reduce a production

Suppose we can build a following smart state transition table



Grammar:

S→E $	r1
E→E+T	r2
| T	r3
T→ID	r4
| (E)	r5

State	Symbol						
	E	T	+	(	)	$	id
0	1	9		6			5
1			3			2	
2		4		6			5
3		4		6			5
4							
5							
6	7	9		6			5
7			3		8		
8							
9							

Let's run an example (ID)

1. starting from state 0, do shift because state 0 tell you so. So stack = $(
2. top(stack) = '(' , so, we go to state 6. state 6 tells you to do shift , so stack = $(ID
3. top(stack) = ID, so we go to state 5. state 5 tells you to do a reduce $T \rightarrow ID$, so stack = $(T
4. When a reduce occurs, go back to previous shift state 6.
5. top(stack) = T, so we go to state 9. state 9 tells you to do a reduce $E \rightarrow T$, so stack = $(E. Again, go back to state 6
6. top(stack) = E, so we go to state 7. state 7 tells you to do a shift, so stack = $(E)
7. top(stack) =), so we go to state 8. state 8 tells you to do a reduce $T \rightarrow (E)$, so stack = $T
8. …..The games go on, until S is reduced….

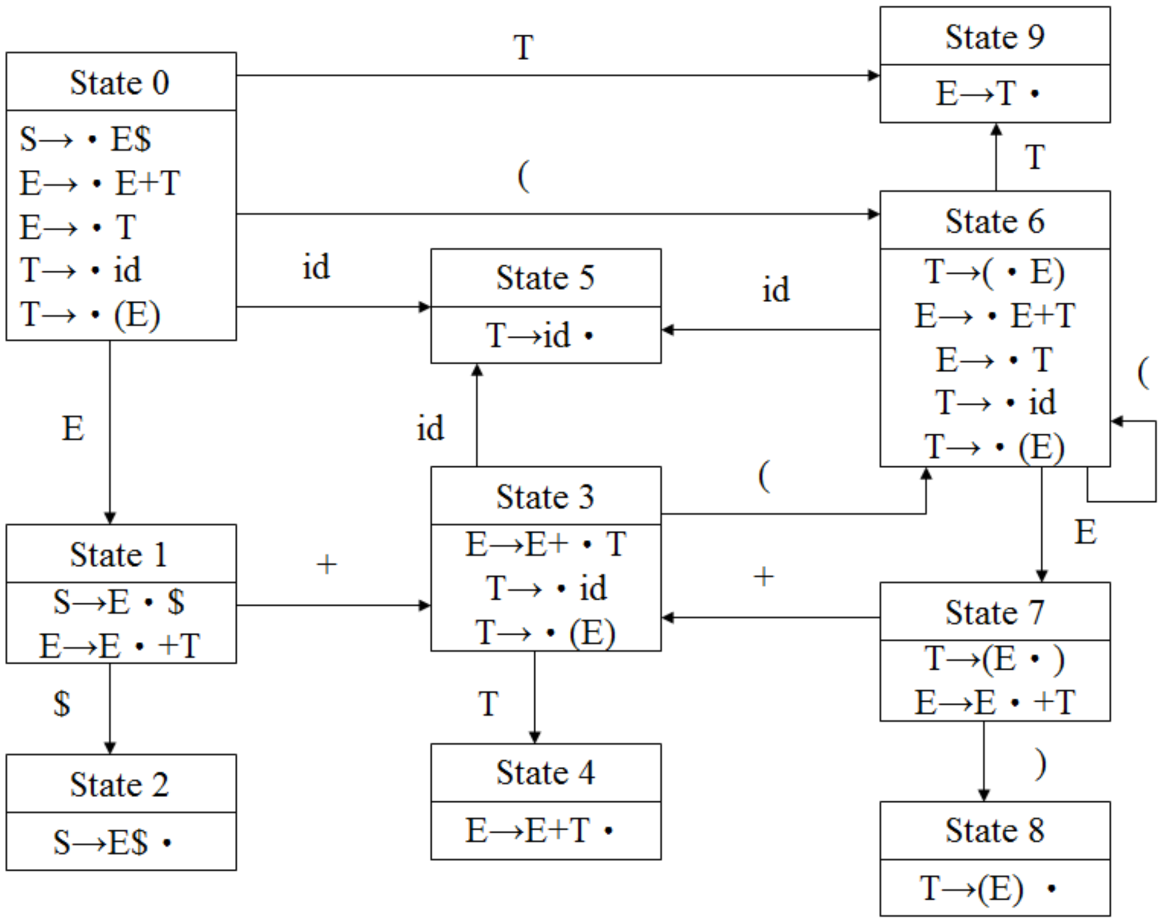
Question is….

- How can we create such a smart transition table to guide the parser choose the right action each time?
- Well, like I told you. in 196x, a computer science Ph.D's topic is to solve the problem.
- The ANSWER of course, we need to derive the tranisition table from the grammar. This is all we have.

醜媳婦總是要見公婆 (利用 grammar 來產生 state transition table

Grammar:

S→E $	r1
E→E+T	r2
| T	r3
T→ID	r4
| (E)	r5



35

LR Parsers (cont.)

In building an LR Parser:

1) Create the Transition Diagram

2) Depending on it, construct:

Go_to Table

Action Table

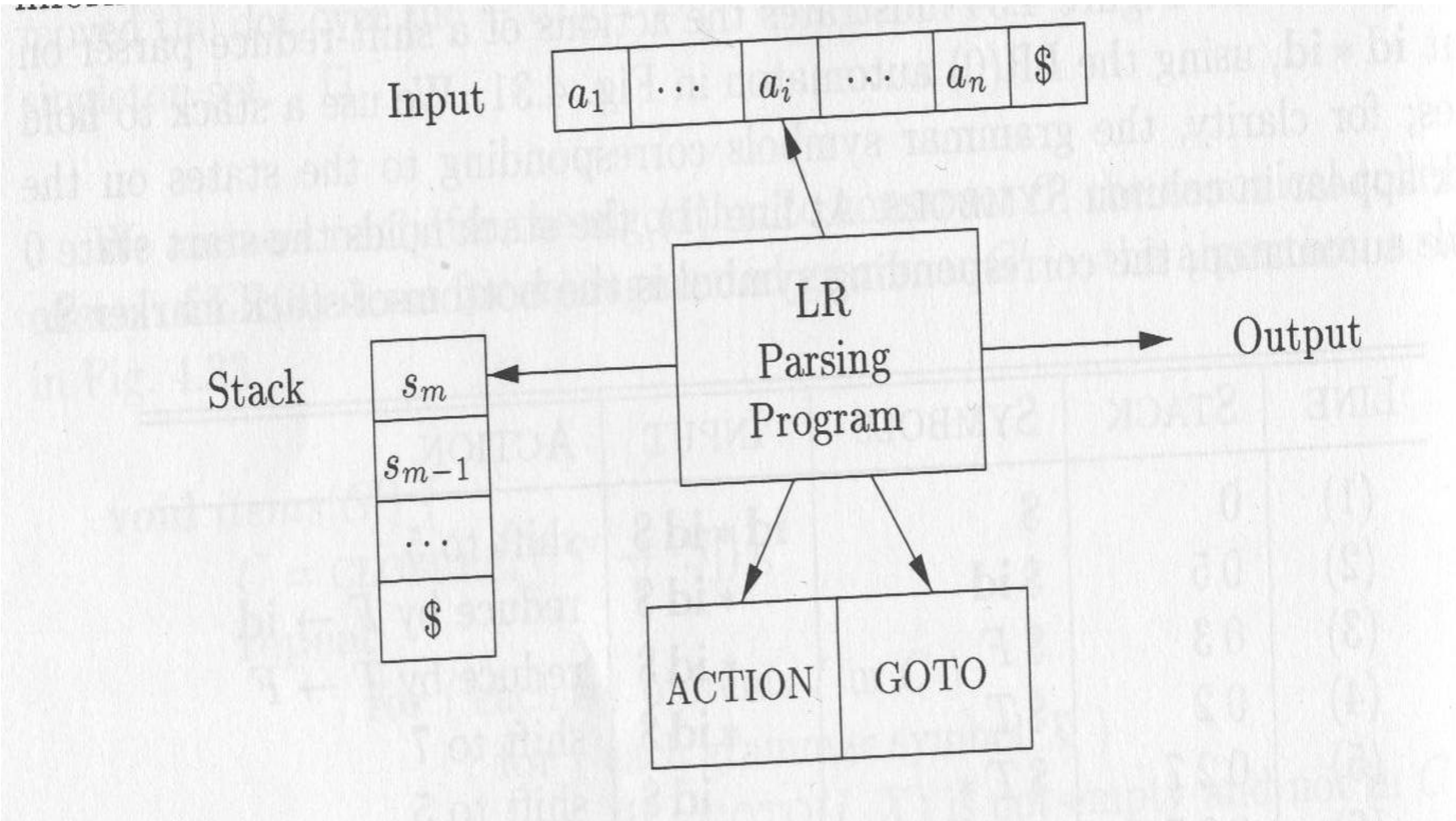
LR Parsers (cont.)

Go_to table defines the
next state after a shift.

Action table tells parser whether to:

- 1) shift (S),
- 2) reduce (R),
- 3) accept (A) the source code, or
- 4) signal a syntactic error (E).

Model of an LR parser



LR Parsers (Cont.)

- An LR parser makes shift-reduce decisions by maintaining states to keep track of where we are in a parse.

- States represent sets of items.

LR(0) Item

- LR(0) and all other LR-style parsing are based on the idea of:

  an *item* of the form:

  $A \rightarrow X1 \ldots Xi \cdot Xi+1 \ldots Xj$

- The *dot symbol* **·**, in an item

  may appear anywhere

  in the right-hand side of a production.
- **It marks how much of the production**

  has already been matched. (See the green part)
- **Remember, this is LR(0) because we do not use any lookahead yet.**

LR (0) Item (Cont.)

- An **LR(0) item** (item for short) of a grammar G is a production of G with a dot at some position of the RHS.

- The production $A \to XYZ$ yields the four items:

$$A \to \cdot XYZ$$
$$A \to X \cdot YZ$$
$$A \to XY \cdot Z$$
$$A \to XYZ \cdot$$

The production $A \to \lambda$ generates only one item, $A \to \cdot$.

LR(0) Item Closure

- If I is a set of items for a grammar G, then CLOSURE(I) is the set of items constructed from I by the 2 rules:

  1) Initially, add every item in I to CLOSURE(I)

  2) If $A \rightarrow \alpha \cdot B\ \beta$ is in CLOSURE(I)

     and $B \rightarrow \gamma$ is a production, then add

     $B \rightarrow \cdot\gamma$ to CLOSURE(I),

     if it is not already there.

  Apply this until no more new items can be added.

LR(0) Closure Example

$E' \rightarrow E$

$E \rightarrow E + T \mid T$

$T \rightarrow T * F \mid F$

$F \rightarrow (E) \mid id$

I is the set of one item $\{E' \rightarrow \cdot E\}$.

Find CLOSURE(I)

LR(0) Closure Example (Cont.)

First, $E' \to \cdot E$ is put in CLOSURE(I) by rule 1.

Then, E-productions with dots at the left end:

$E \to \cdot E + T$ and $E \to \cdot T$.

Now, there is a T immediately to the right of a dot in $E \to \cdot T$, so we add $T \to \cdot T * F$ and $T \to \cdot F$.

Next, $T \to \cdot F$ forces us to add:

$F \to \cdot(E)$ and $F \to \cdot id$.

Another Closure Example

$S \rightarrow E\ \$$
$E \rightarrow E + T \mid T$
$T \rightarrow ID \mid (E)$

$$\text{closure}(S \rightarrow \cdot E\$) = \{S \rightarrow \cdot E\$,\ E \rightarrow \cdot E+T,\ E \rightarrow \cdot T,\ T \rightarrow \cdot ID,\ T \rightarrow \cdot (E)\}$$

The five **items** above forms an **item set** **called state s_0.**

OK, 從前面的說明中，我們知道 transition table 的一個狀態是由這些items 所構成。但是數十年前研究 compiler 的電腦科學家怎麼會想出來這一套技術?

Closure (I)

```
SetOfItems Closure(I) {
   J=I
   repeat
         for (each item A → α·B β in J)
               for (each production B → γ of G)
                     if (B → · γ is not in J)
                           add B → · γ to J;
   until no more items are added to J;
   return J;
} // end of Closure (I)
```

Goto Next State

- Given an item set (state) s,

  we can compute its *next state*, s’,
  under a symbol X,

  that is, Go_to (s, X) = s’

Goto Next State (Cont.)

$E' \rightarrow E$

$E \rightarrow E + T \mid T$

$T \rightarrow T * F \mid F$

$F \rightarrow (E) \mid id$

S is the item set (state):

$\mathbf{E \rightarrow E \cdot + T}$

Goto Next State (Cont.)

S' is the next state that Goto(S, +) goes to:

$E \to E + \cdot T$

$T \to \cdot T * F$ (by closure)

$T \to \cdot F$ (by closure)

$F \to \cdot(E)$ (by closure)

$F \to \cdot id$ (by closure)

We can build all the states of the Transition Diagram this way.

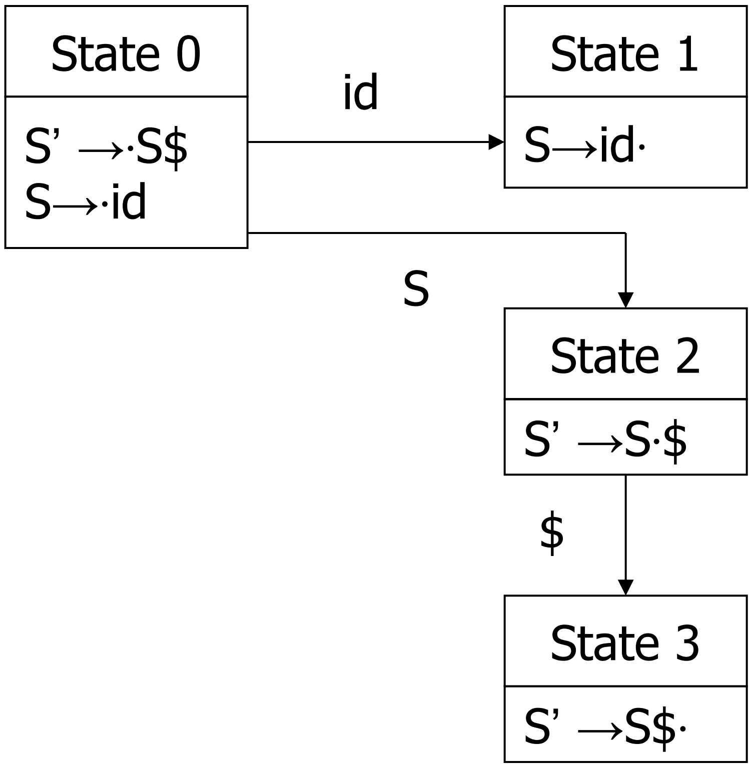
An LR(0) Complete Example

Grammar:

$$S' \rightarrow S\ \$$$

$$S \rightarrow ID$$

LR(0) Transition Diagram



LR(0) Transition Diagram (Cont.)

Each state in the Transition Diagram,

either signals a shift
(·moves to right of a terminal)

or signals a reduce
(reducing the RHS **handle** to LHS)

LR(0) Go_to table

State	Symbol		
	ID	$	S
0	1		2
1			
2		3	
3			

The blanks above indicate errors.

LR(0) Action table

State	0	1	2	3
Action	S	R2	S	A

- S for shift
- A for accept
- R2 for reduce by Rule 2
- Each state has only one action.

LR(0) Parsing

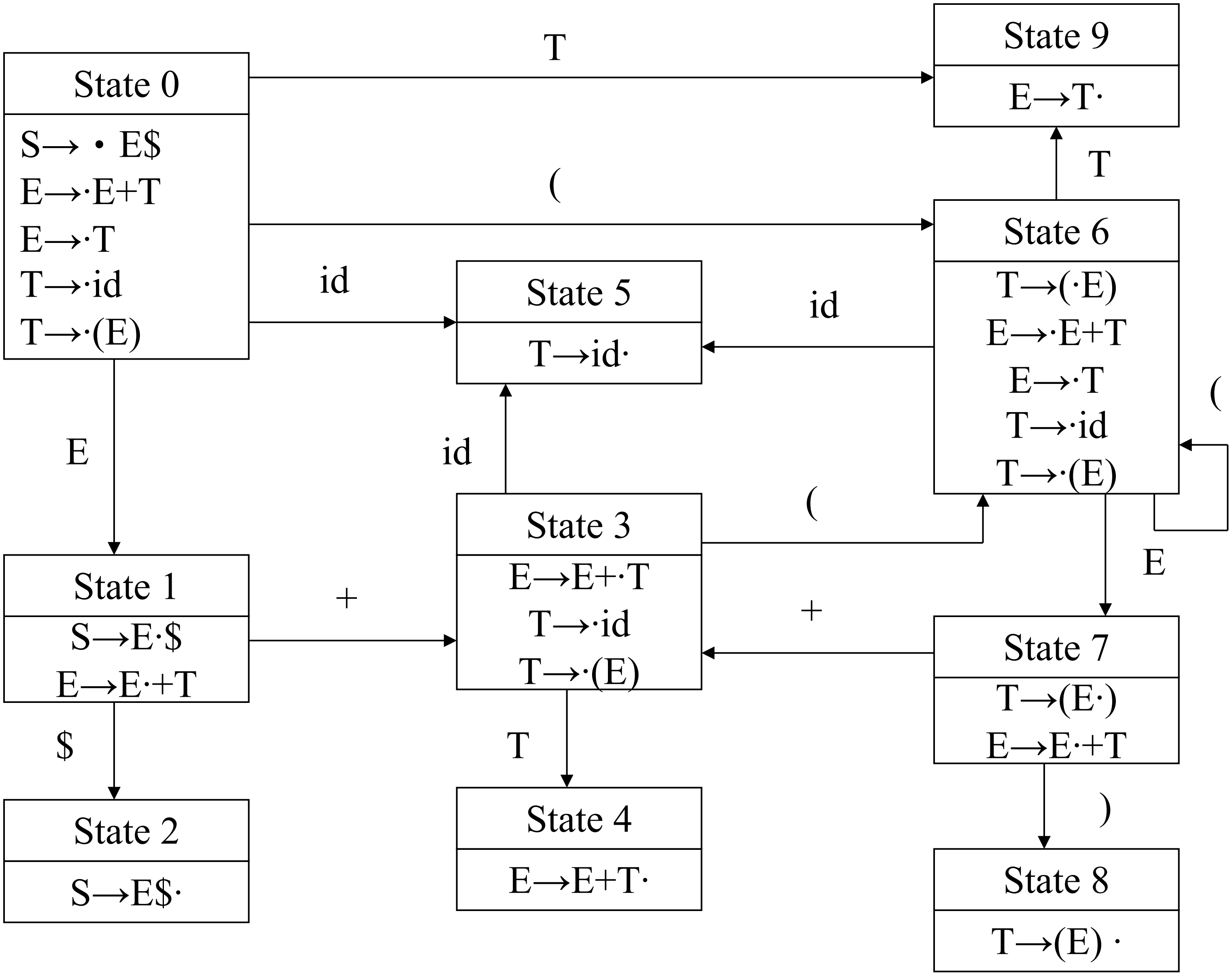
Stack	Input	Action
S0	id $	shift
S0 id S1	$	reduce r2
S0 S S2	$	shift
S0 S S2 $ S3		reduce r1
S0 S’		accept

Another LR(0) Example

Grammar:

$S \longrightarrow E\ \$$	r1
$E \longrightarrow E+T$	r2
$\quad\mid T$	r3
$T \longrightarrow ID$	r4
$\quad\mid (E)$	r5

LR(0) Transition Diagram



LR(0) Go_to table

State	Symbol						
	E	T	+	(	)	$	id
0	1	9		6			5
1			3			2	
2		4		6			5
3		4		6			5
4							
5							
6	7	9		6			5
7			3		8		
8							
9							

LR(0) Action table

State:	0	1	2	3	4	5	6	7	8	9
Action:	S	S	A	S	R2	R4	S	S	R5	R3

LR(0) Parsing

Stack	Input	Action
S0	id + id $	shift
S0 id S5	+ id $	reduce r4
S0 T S9	+ id $	reduce r3
S0 E S1	+ id $	shift
S0 E S1 + S3	id $	shift
S0 E S1 + S3 id S5	$	reduce r4
S0 E S1 + S3 T S4	$	reduce r2
S0 E S1	$	shift
S0 E S1 $ S2		reduce r1
S0 S		accept

NOTES

- OK, so far, remember, we simply go to a state, look at the TOP OF STACK, and make a transition. That is why it is LR(0)
- We never use the lookahead from the input yet.
- If we consider more information from input, we should get a more powerful parser.
- What do we mean by powerful parser? The answer is: It creates less shift/reduce and reduce/reduce conflicts why generates transition table. Some conflicts can be automatically resolved if lookahead is considered.

Grammar that is not LR(0)

$S \rightarrow E$

$E \rightarrow 1\ E$

$E \rightarrow 1$

Simple LR(1), SLR(1), Parsing

SLR(1) has the same Transition Diagram and Goto table as LR(0)

BUT with different Action table

because it looks ahead 1 token.

SLR(1) Look-ahead

- SLR(1) parsers are built first by constructing Transition Diagram, then by computing Follow set as SLR(1) look-aheads. Oh yes, we need the follow set again.

- The ideas is:
  A **handle** (RHS) should NOT be reduced to N
  if the look ahead token is NOT in follow(N)

SLR(1) Look-ahead (Cont.)

$S \rightarrow E\ \$$	r1
$E \rightarrow E + T$	r2
$\quad \mid\ T$	r3
$T \rightarrow ID$	r4
$T \rightarrow (\ E\ )$	r5

Follow (S) = { $}

Follow (E) = {), +, $}

Follow (T) = {), +, $}

For your reminder

First and Follow (Cont.)

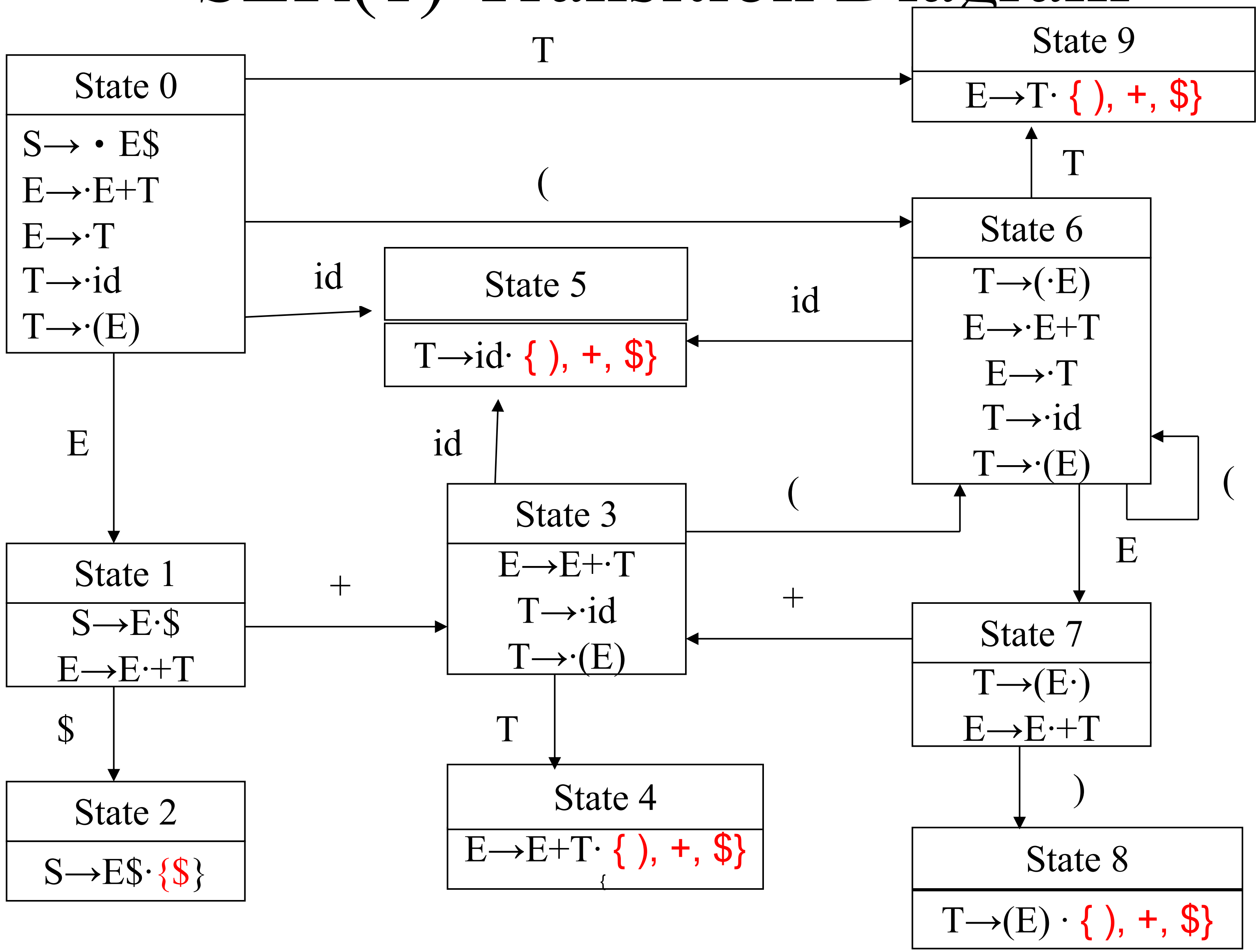
- To compute FOLLOW(B) for non-terminal B:
  - 1. Place $ in FOLLOW(S), where S is the start symbol, and $ is the input right end-marker.
  - 2. if there is a production $A \rightarrow \alpha\, B\, \beta$, then everything in FIRST(β) except λ is in FOLLOW(B).
  - 3. (a) if there is a production $A \rightarrow \alpha\, B$,

    (b) or $A \rightarrow \alpha\, B\, \beta$, where FIRST($\beta$) contains λ,

    then everything in FOLLOW(A) is in FOLLOW(B).

Use the follow sets as look-aheads in reduction.

SLR(1) Transition Diagram



SLR(1) Goto table

	ID	+	(	)	$	E	T
0	5					1	6
1		3			2		
2							
3	5		7				4
4							
5							
6							
7	5		7			8	6
8		3		9			
9							

SLR(1) Action table, which expands LR(0) Action table

	ID	+	(	)	$
0	S		S		
1		S			S
2					R1
3	S		S		
4		R2		R2	R2
5		R4		R4	R4
6		R3		R3	R3
7	S		S		
8		S		S	
9		R5		R5	R5

For your comparison this is LR(0) action table which does concern one lookahead

LR(0) Action table

State:	0	1	2	3	4	5	6	7	8	9
Action:	S	S	A	S	R2	R4	S	S	R5	R3

NOTE

- OK, to be honest, the example's LR(0) have no reduce/reduce conflicts. So, when LR(0) action table is expanded to SLR(1), of course there is no reduce/reduce conflicts neither.
- This is no surprise. In other words, the example (from the text books) are not good.

A shift/reduce example from an SLR(1) grammar

- The SLR(1) grammar below causes a shift-reduce conflict:

  r1,2 $S \rightarrow A \mid xb$

  r3,4 $A \rightarrow aAb \mid B$

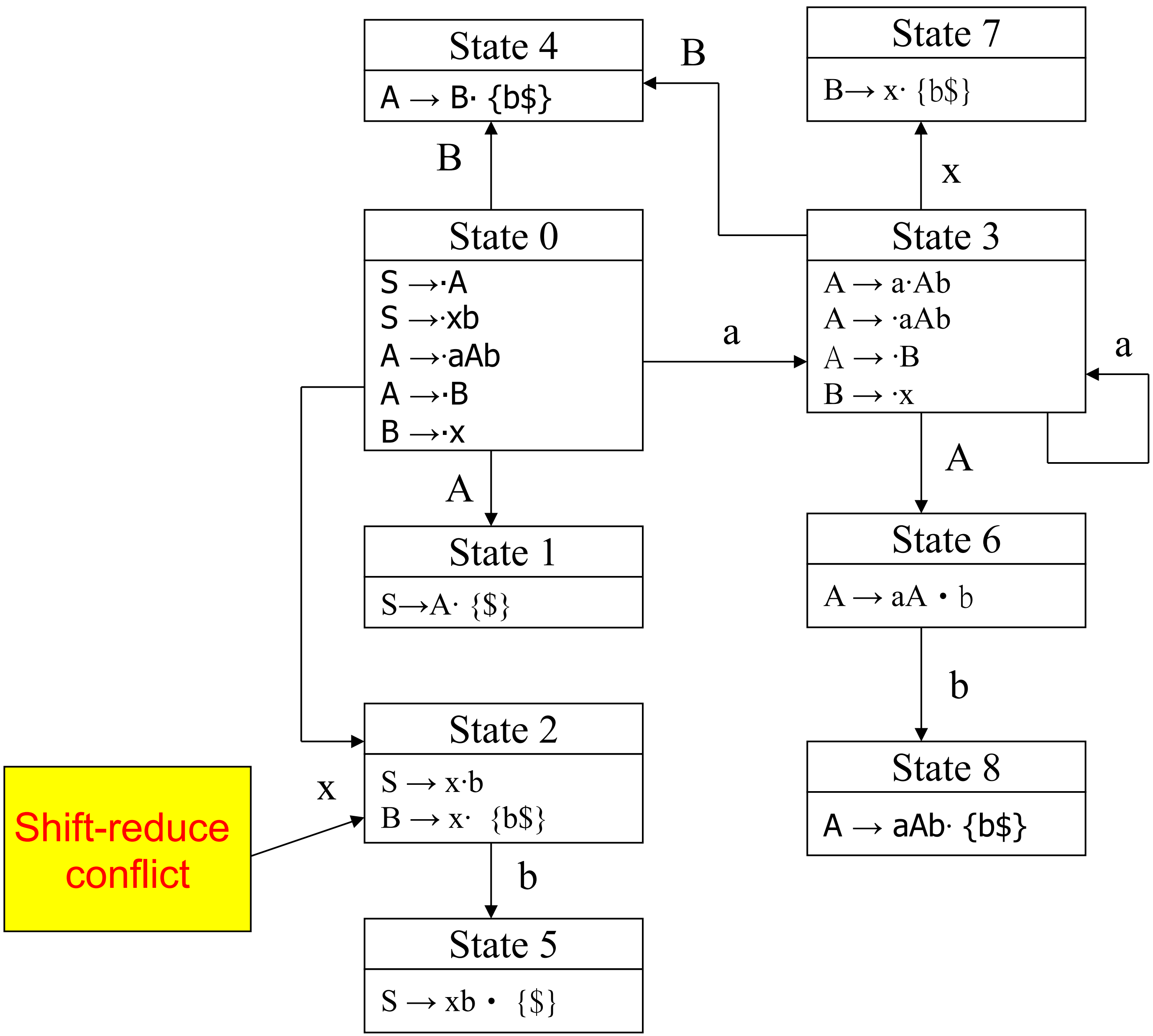
  r5 $B \rightarrow x$

Use follow(S) = {\$},

follow(A) = follow(B) = {b \$}

in the SLR(1) Transition Diagram next.

SLR(1) Transition Diagram



SLR(1) Go_to table

	0	1	2	3	4	5	6	7	8
A				6					
B	4			4					
a	3			3					
b			5				8		
x	2			7					

SLR(1) Action table

state token	0	1	2	3	4	5	6	7	8
b			R5/S		R4		S	R5	R3
$		R1	R5		R4	R2		R5	R3
a	S			S					
x	S			S					

State 2 **(**R5/S) causes shift-reduce **conflict:**

When handling ‘b’, the parser doesn’t know whether to reduce by rule 5 (R5) or to shift (S).

Solution: Use more powerful LR(1)

LR(1) Parsing

The reason why the FOLLOW set does not work as well as one might wish is that:

It replaces the look-ahead of a single item of a rule N in a given LR state by:

the whole FOLLOW set of N,

which is the **union** of all the look-aheads of all alternatives of N in all states.

(my quotes: this slide basically tell you SLR(1) is useless in practice)

Solution: Use LR(1)

LR(1) Parsing

LR(1) item sets are more discriminating:

A look-ahead set is kept with each separate item, to be used to resolve conflicts when a reduce item has been reached.

This greatly increases the strength of the parser, but also the size of its tables.

LR(1) item

An LR(1) item is of the form:

$A \rightarrow X1 \ldots Xi \cdot Xi{+}1 \ldots Xj, l$

where l belongs to $Vt \cup \{\lambda\}$

l is look-ahead 偷看 tape head symbol

Vt is vocabulary of terminals

λ is the look-ahead after end marker \$

LR(1) item look-ahead set

Rules for look-ahead sets:

1) initial item set: the look-ahead set of the initial item set S_0 contains only one token, the end-of-file token ($), the only token that follows the start symbol.

2) other item set:

Look ahead set

First(β) + σ

Given $P \rightarrow \alpha \cdot N \beta \ \{\sigma\}$, we have

$N \rightarrow \cdot \gamma$ **{FIRST($\beta\{\sigma\}$) }** in the item set.

First{(β(σ)}

LR(1) look-ahead

The LR(1) look-ahead set FIRST(β{σ}) is:

If β can produce λ (β →* λ),

FIRST(β{σ}) is:

FIRST(β) plus the tokens in {σ}, excludes λ.

else

FIRST(β{σ}) just equals FIRST(β);

a P b

a α N β b

λ

$P \to \alpha \bullet N \beta, \{\sigma\}$

$N \to \bullet \gamma$

First(β)

λ

An LR(1) Example

Given the grammar below,

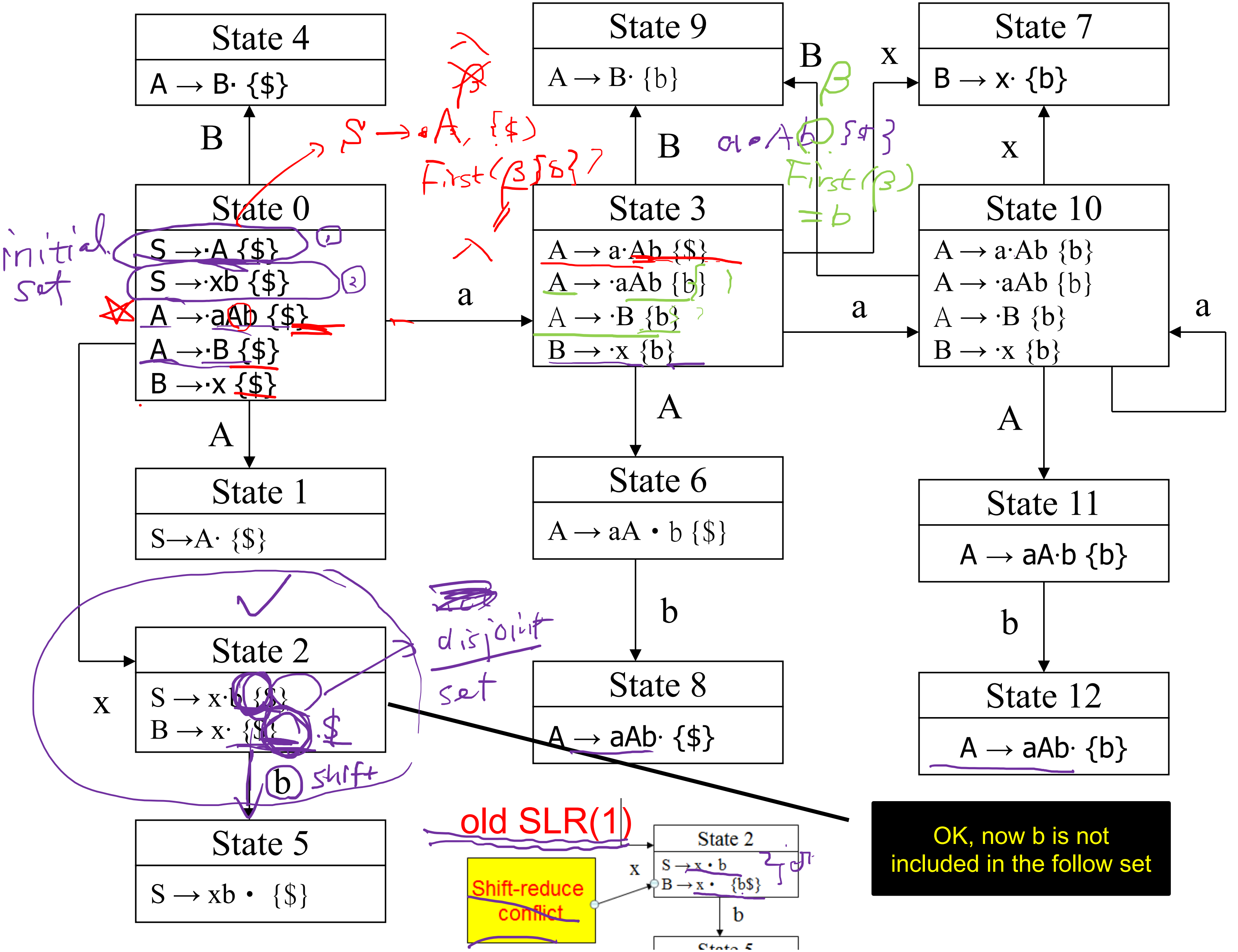
create the LR(1) Transition Diagram.

r1,2 $S \rightarrow A \mid xb$

r3,4 $A \rightarrow aAb \mid B$

r5 $B \rightarrow x$

LR(1) Transition Diagram



LR(1) Go_to table

	0	1	2	3	4	5	6	7	8	9	10	11	12
A	1			6							11		
B	4			9							9		
a	3			10							10		
b			5				8					12	
x	2			7							7		

2개

LR(1) Action table

State / token	0	1	2	3	4	5	6	7	8	9	10	11	12
$		R1	R5		R4	R2			R3				
b			S				S	R5		R4		S	R3
a	S			S							S		
x	S			S							S		

The states are from 0 to 12 and
the terminal symbols include $,b,a,x.

LR(1) Parsing

- LR(1)'s problem is that:
  The **LR(1) Transition Diagram**
    contains *so many states* that
  the **Go_to and Action tables**
    become *prohibitively large*.

- Solution: Use LALR(1) (look-ahead LR(1))
  to reduce table sizes.

NOTES

- YES, it is weird. We improve SLR(1) into LR(1), which is powerful, however, resource intensive.
- Imaging when you compiler a program, the compiler need 1G of memory to proceed. This is not good.

Look-ahead LR(1), LALR(1), Parsing

bison/yacc

L

LR(1) ~~LALR~~ LALR(1)

- LALR(1) parser can be built by first constructing an LR(1) transition diagram and then **merging states**.

概念上

- It differs with LR(1) *only* in its ***merging look-ahead* components of the items with *common core*.**

LALR(1) Parsing (Cont.)

- Consider states s and s' below in LR(1):

  s : $A \rightarrow a\cdot$ {b}　　　s' : $A \rightarrow a\cdot$ {c}

  　　$B \rightarrow a\cdot$ {d}　　　　　$B \rightarrow a\cdot$ {e}

  s and s' have common core :

  $A \rightarrow a\cdot$

  $B \rightarrow a\cdot$

  So, we can merge the two states :

  $A \rightarrow a\cdot$ {b,c}

  $B \rightarrow a\cdot$ {d,e}

  disjoint set

LALR(1) Parsing (Cont.)

For the grammar:

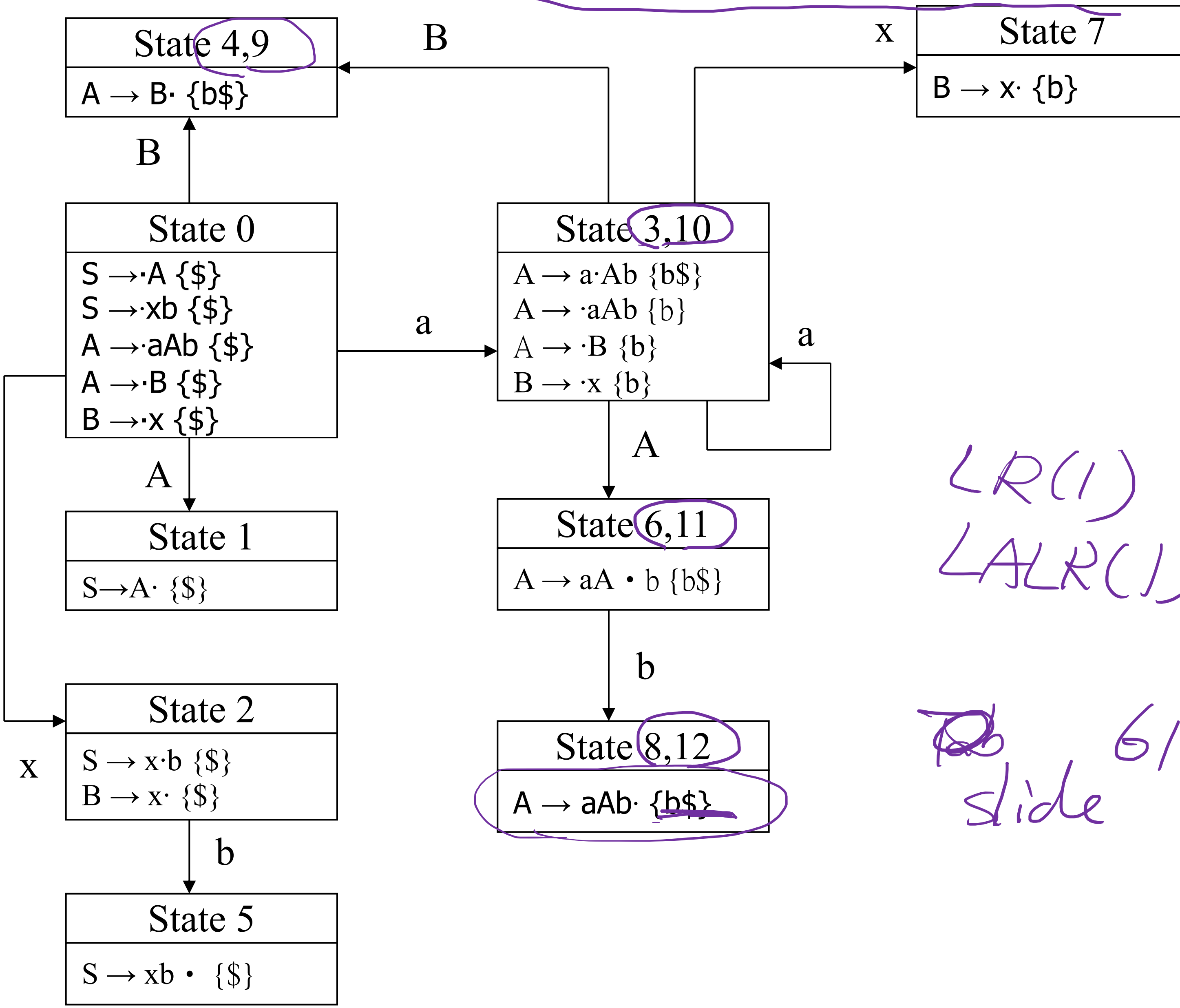
r1,2 $S \rightarrow A \mid xb$

r3,4 $A \rightarrow aAb \mid B$

r5 $B \rightarrow x$

Merge the states in the LR(1) Transition Diagram to get that of LALR(1).

LALR(1) Transition Diagram



LR(1)
LALR(1)

61
slide

Merging States

LALR(1) State	LR(1) States with Common Core
State 0	State 0
State 1	State 1
State 2	State 2
State 3	**State 3, State 10**
State 4	**State 4, State 9**
State 5	State 5
State 6	**State 6, State 11**
State 7	State 7
State 8	**State 8, State 12**

LALR(1) Go_to table

	0	1	2	3	4	5	6	7	8
A	1			6					
B	4			4					
a	3			3					
b			5				8		
x	2			7					

LALR(1) Action table

	0	1	2	3	4	5	6	7	8
$		R1	R5		R4	R2			R3
b			S		R4		S	R5	R3
a	S			S					
x	S			S					

An Example of 4 LR Parsings

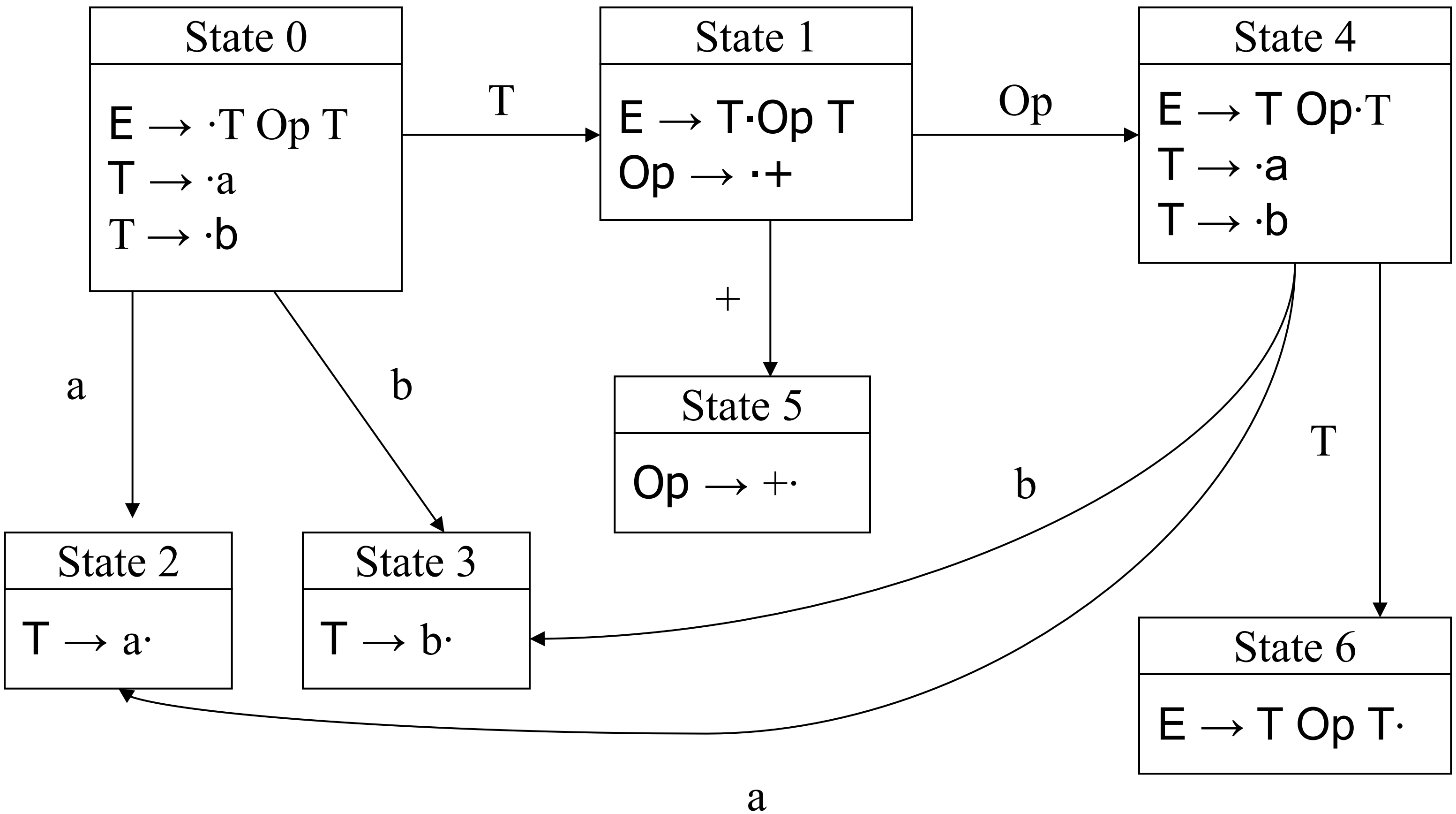
Given the grammar below:

$$\left.\begin{array}{lll} r1 & E & \rightarrow T\ Op\ T \\ r2 & T & \rightarrow a \\ r3 & & \mid\ b \\ r4 & Op & \rightarrow + \end{array}\right\} 4$$

write 1) state transition diagram
2) action table
3) goto table

for 1) LR(0), 2) SLR(1), 3) LR(1) and 4) LALR(1)
4 bottom-up parsing methods, respectively.

LR(0) transition diagram



LR(0) Action table

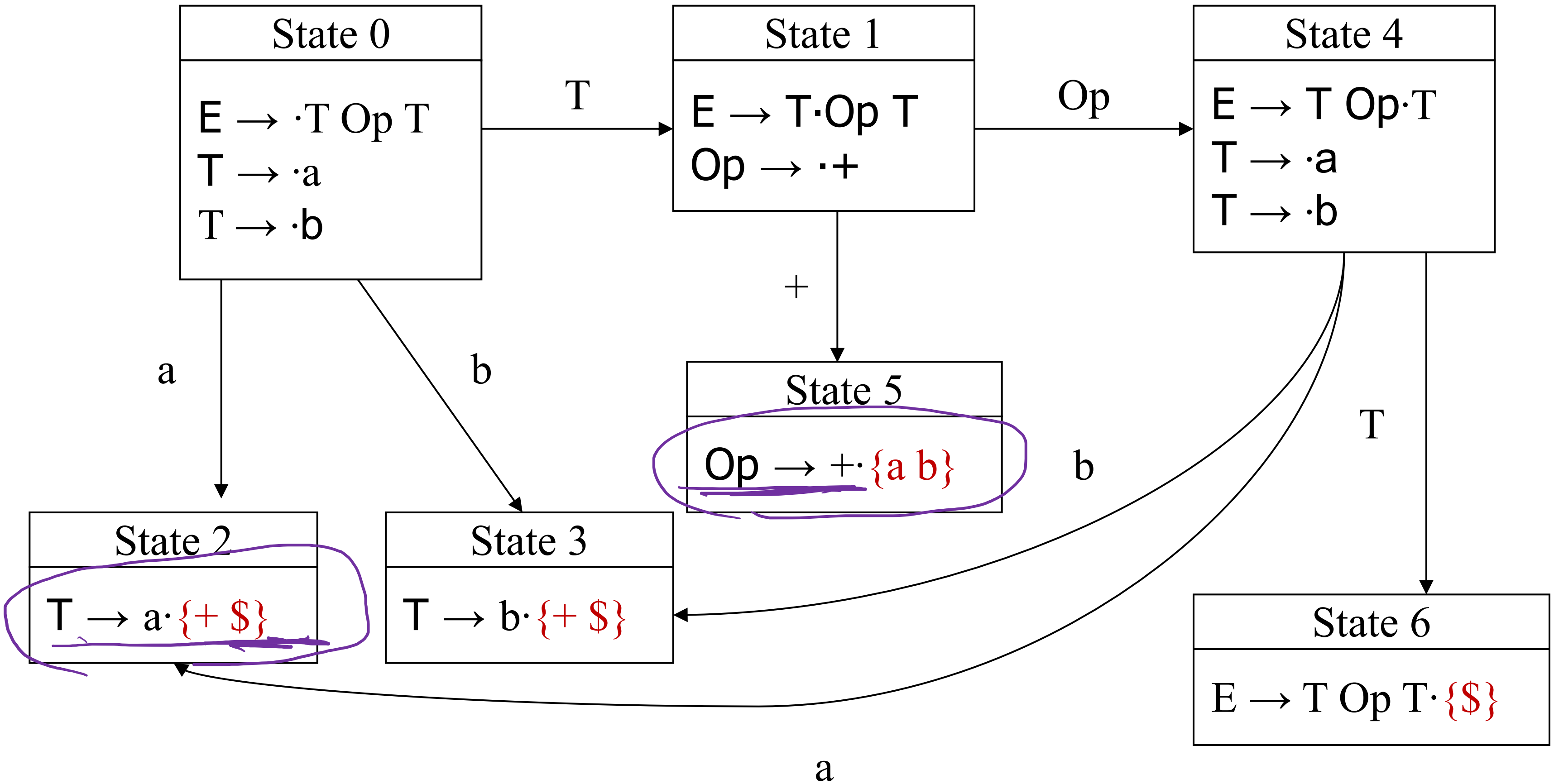
State	0	1	2	3	4	5	6
Action	S	S	R2	R3	S	R4	A

LR(0) Go_to table

	E	T	Op	a	b	+
0		1		2	3	
1			4			5
2						
3						
4		6		2	3	
5						
6						

SLR(1) Transition Diagram

Simply add Follow(N) as look-ahead to the state that is about to do N reduction.



SLR(1) Action table

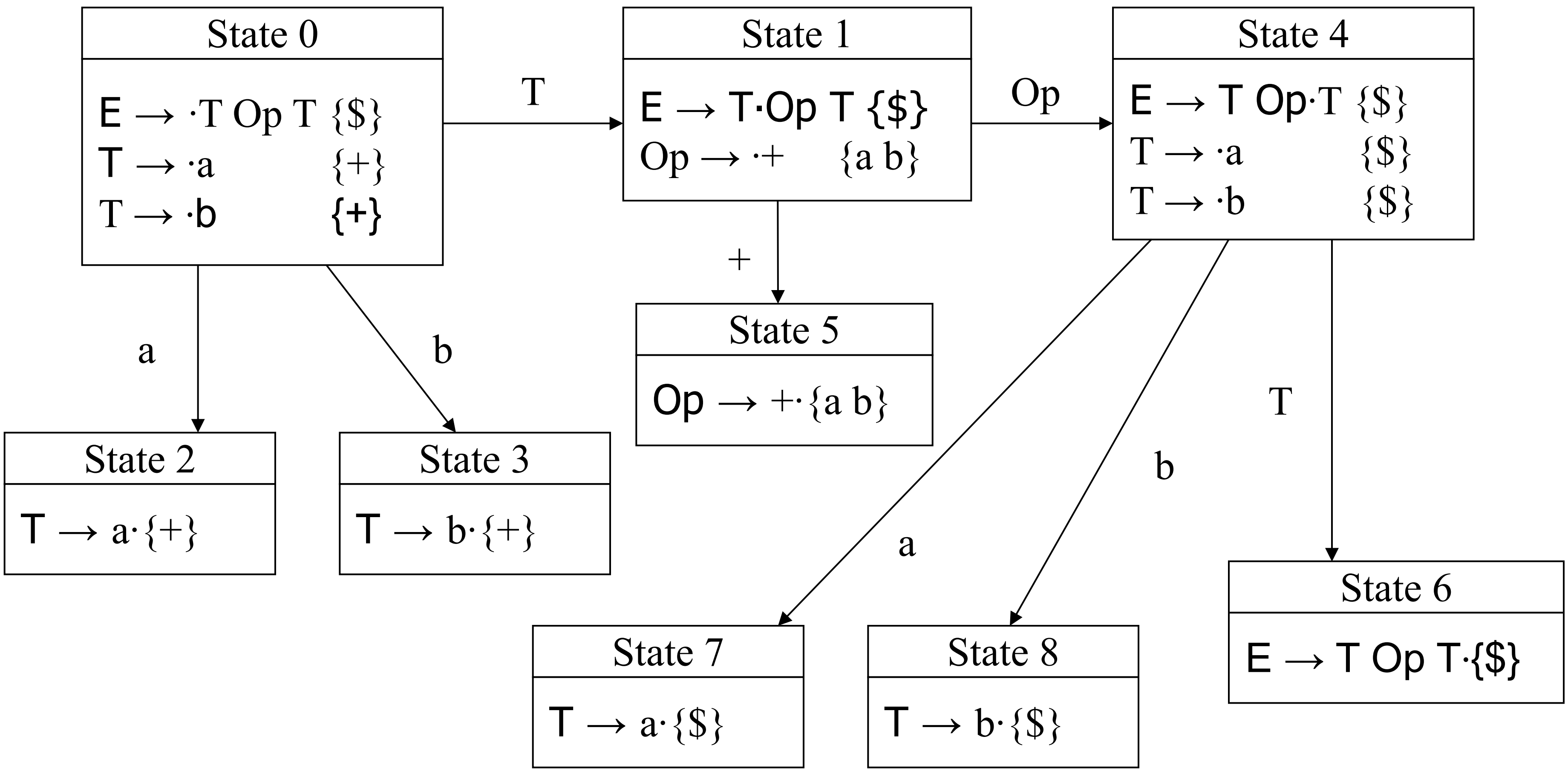
State	0	1	2	3	4	5	6
a	S				S	R4	
b	S				S	R4	
+		S	R2	R3			
$			R2	R3			A

SLR(1) Go_to table

The SLR(1) goto table is the same as that of LR(0).

LR(1) Transition Diagram

Add look-ahead sets when about to reduce.



LR(1) Action table

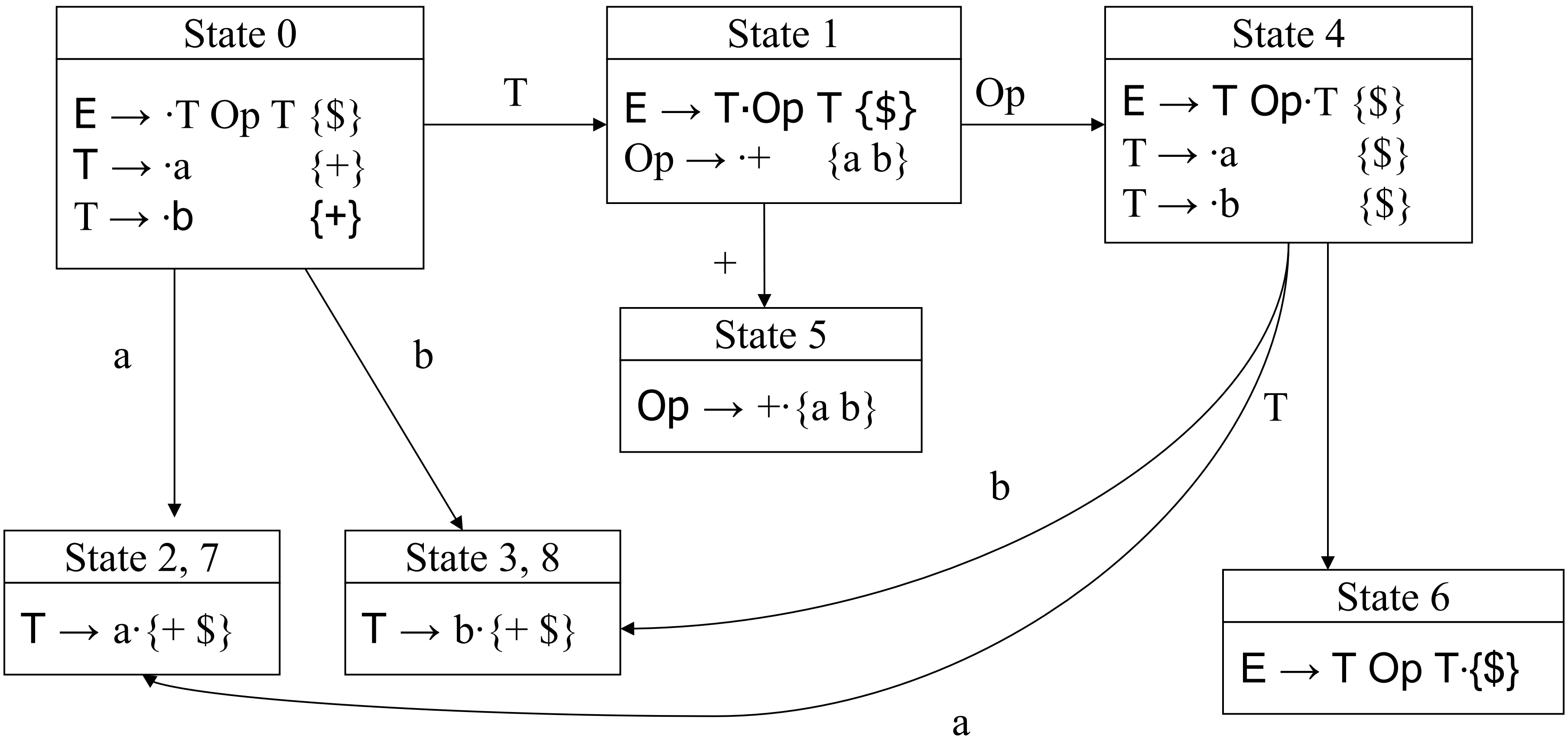
State	0	1	2	3	4	5	6	7	8
a	S				S	R4			
b	S				S	R4			
+		S	R2	R3					
$							A	R2	R3

LR(1) Go_to table

	E	T	Op	a	b	+
0		1		2	3	
1			4			5
2						
3						
4		6		7	8	
5						
6						
7						
8						

LALR(1) Transition diagram

Merge states with common core:



LALR(1) Action table

State	0	1	4	5	6	2, 7	3, 8
a	S		S	R4			
b	S		S	R4			
+		S				R2	R3
$					A	R2	R3

LALR(1) Go_to table

	E	T	Op	a	b	+
0		1		2, 7	3, 8	
1			4			5
4		6		2, 7	3, 8	
5						
6						
2, 7						
3, 8						

Think: when parsing a $ b,

LALR(1) will be less powerful than LR(1).

Final Notes

- OK, finally, you know what Bison/YACC is doing.
- You give it a LALR(1) grammar, it parses your grammar and then generate goto_table, action table (with lookahead), and the code to execute the pushdown automata. These elements form a parser.
- When you input a stream of tokens, the parser read input and execute the pushdown automata smartly, efficiently, with the tables.

How to resolve conflicts

- OK, sometimes your grammar generates shift/reduce or reduce/reduce conflicts. How can we do about it?

```
state 97

   29 selection-stmt: IF LFT_BRKT expression RGT_BRKT statement .
   30               | IF LFT_BRKT expression RGT_BRKT statement . ELSE statement

    ELSE  shift, and go to state 100

    ELSE      [reduce using rule 29 (selection-stmt)]
    $default  reduce using rule 29 (selection-stmt)
```

- The answer: rewrite the grammar so that it still can do the same thing but the conflicts can be resolved.

Example – The Hanging-Else problem

```
Id = {Letter}{AlphaNumeric}*

<Statement> ::= if Id then <Statement>
              | if Id then <Statement> else <Statement>
              | Id ':=' Id
```

- A shift/reduce error

Causes

The ambiguity of the grammar can be seen with a very simple piece of source code:

```
if Enrolled then if Studied then Grade:=A else Grade:=B
```

The sample source code could be interpreted two distinct ways by the grammar. The first interpretation would bind the "else" to the first "if".

```
if Enrolled then if Studied then Grade:=A else Grade:=B
```

The second interpretation would bind the "else" to the second 'if" statement:

```
if Enrolled then if Studied then Grade:=A else Grade:=B
```

Fortunately, there are two approaches you can take to resolve the problem.

Solution #1

This approach modifies the grammar such that the scope of the "if" statement is explicitly stated. Another terminal is added to the end of each "if" statement, in this case an "end". A number of programming languages use this approach; the most notable are: Visual Basic and Ada.

```
Id = {Letter}{AlphaNumeric}*


<Statement> ::= if Id then <Statement> end
              | if Id then <Statement> else <Statement> end
              | Id ':=' Id
```

As seen below, the ambiguity of the original grammar has been resolved.

```
if Enrolled then if Studied then Grade:=A end else Grade:=B end
if Enrolled then if Studied then Grade:=A else Grade:=B end end
```

Solution #2

This solution resolves the hanging-else problem by restricting the "if-then" statement to remove ambiguity. Two levels of statements are declared with the second, "restricted", group only used in the "then" clause of a "if-then-else" statement. The "restricted" group is completely identical the the first with one exception: only the "if-then-else" variant of the if statement is allowed.

In other words, no "if" statements without "else" clauses can appear inside the "then" part of an "if-then-else" statement. Using this solution, the "else" will bind to the last "If" statement, and still allows chaining. This is the case with the C/C++ programming language family.

```
Id = {Letter}{AlphaNumeric}*


<Statement> ::= if Id then <Statement>
              | if Id then <Then Stm> else <Statement>
              | Id ':=' Id


<Then Stm>  ::= if Id then <Then Stm> else <Then Stm>
              | Id ':=' Id
```

Unfortunately, this adds a number of rules, but it is ultimately the price you pay for such a

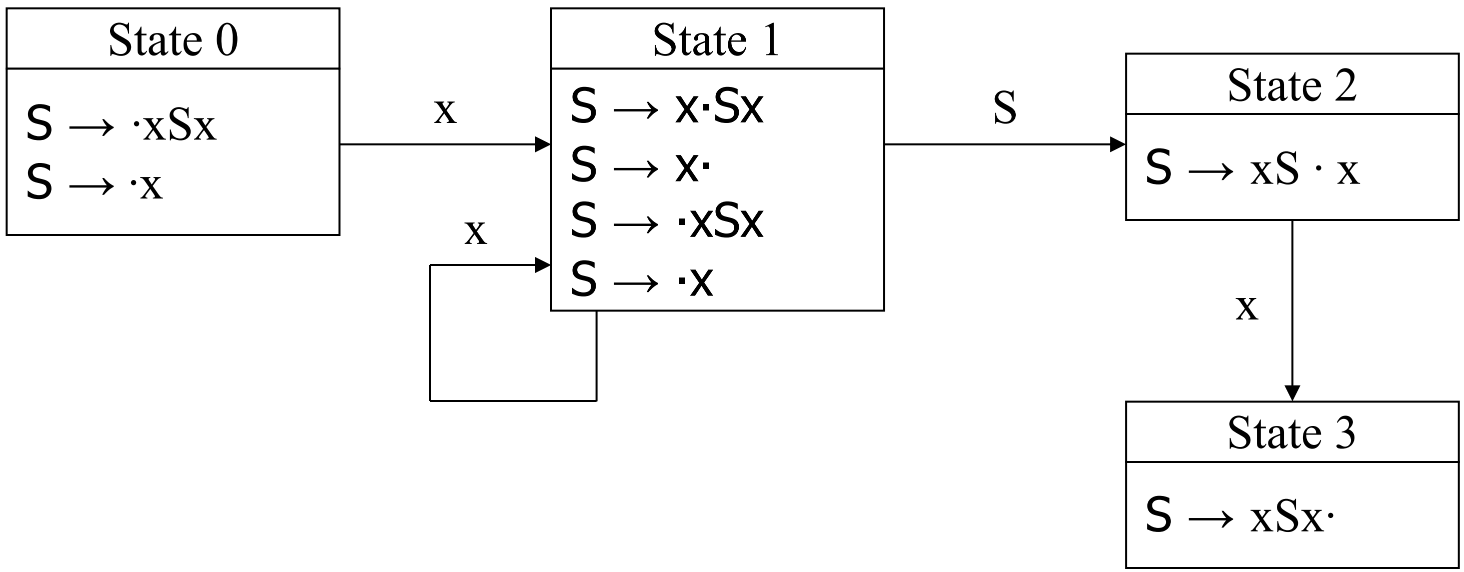
Homework

Construct the Transition Diagram, Action table and Go_to table for:

LR(0), SLR(1), LR(1), and LALR(1) respectively for the grammar below:

$$S \rightarrow xSx$$
$$\mid x$$

Homework Answer
LR(0) Transition Diagram



LR(0) Action/Go_to tables

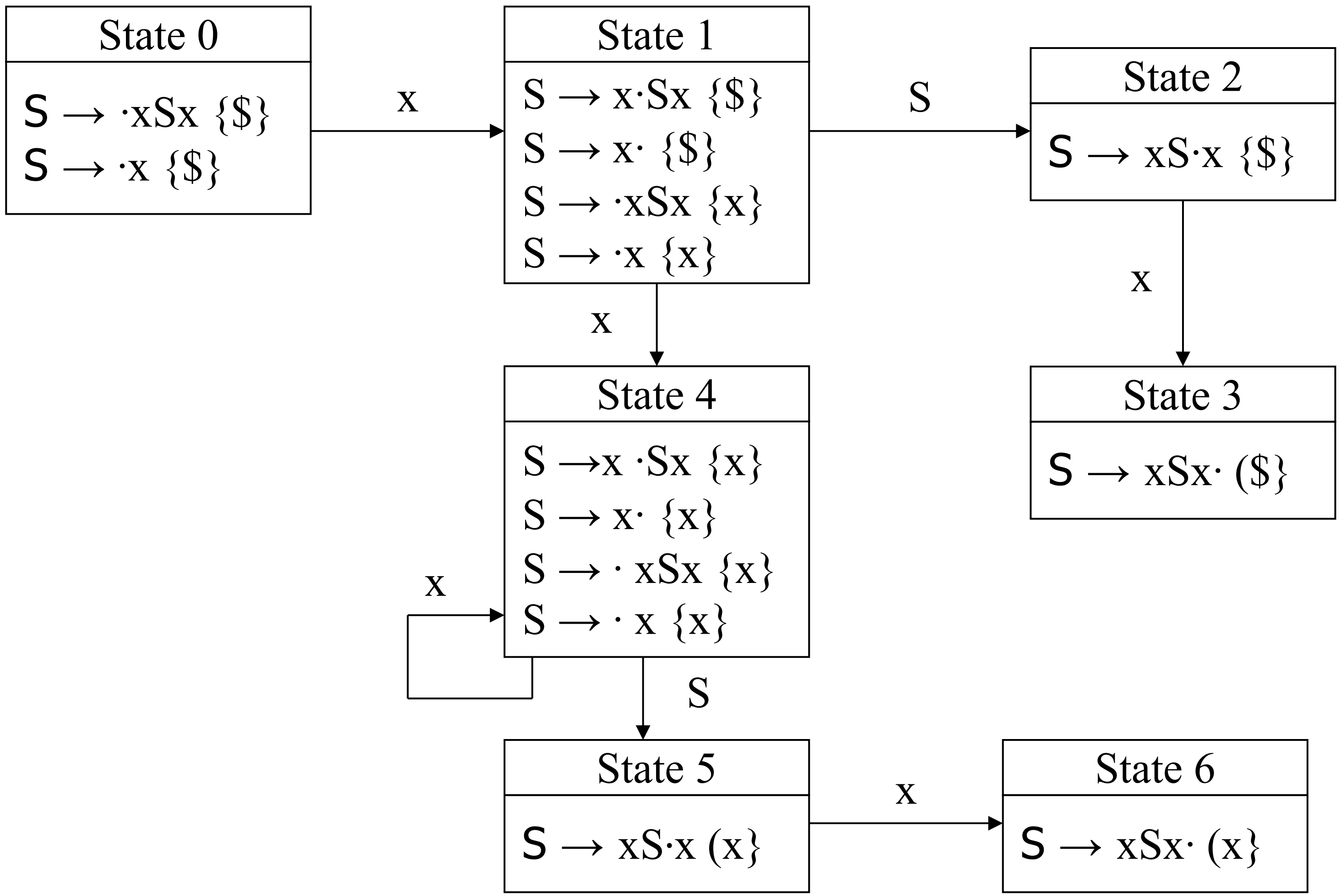
Action table

State	0	1	2	3
Action	S	S/R	S	A

Go_to table

	x	S
0	1	
1	1	2
2	3	
3		

LR(1) Transition Diagram



LR(1) Action, Go_to tables

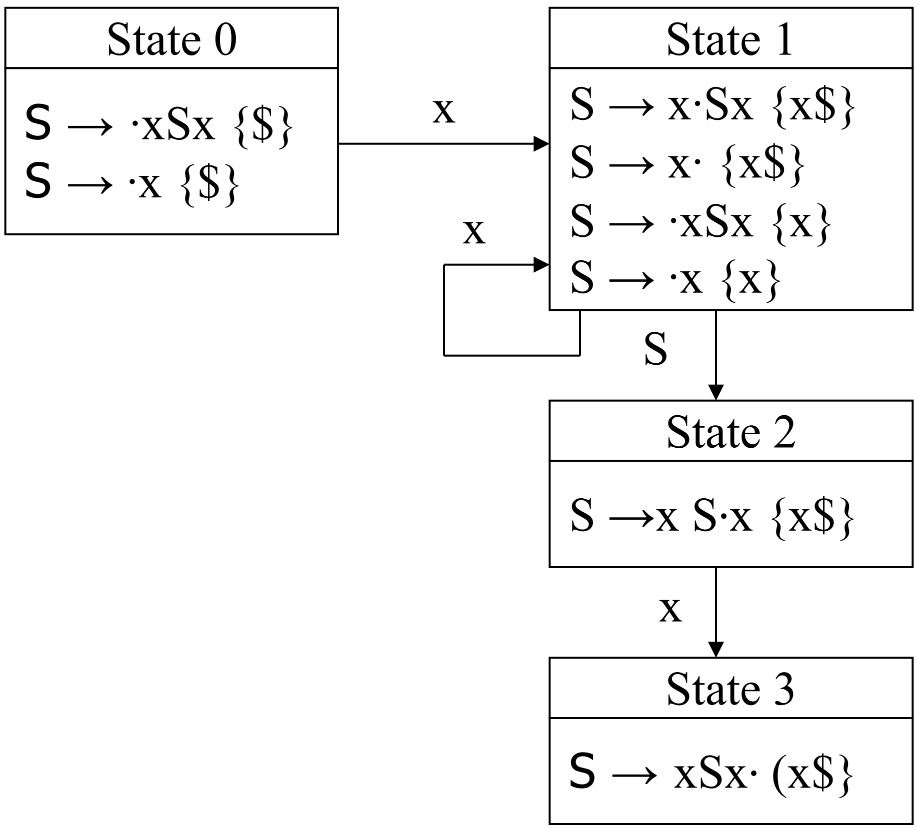
Action table

	x	$
0	S	
1	S	R2
2	S	
3		R1
4	S/R2	
5	S	
6	R1	

Go_to table

	x	S
0	1	
1	4	2
2	3	
3		
4		5
5	6	
6		

SLR(1) Transition Diagram



SLR(1) Action, Go_to tables

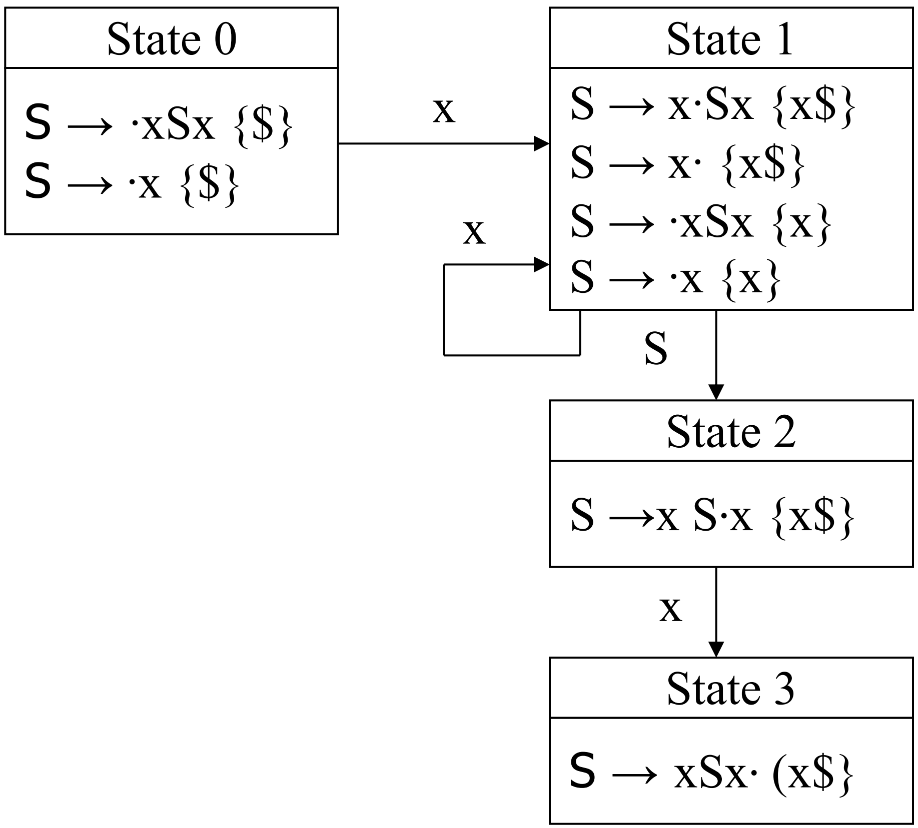
Action table

	x	$
0	S	
1	S/R2	R2
2	S	
3	R1	R1

Go_to table

	x	S
0	1	
1	1	2
2	3	
3		

LALR(1) Transition Diagram



LALR(1) Action, Goto tables

The LALR (1) action table and go_to table are the same as those in SLR(1).