Compiler Technology of Programming Languages

Chapter 6
LR Parsing Techniques

Prof. Farn Wang

LR (Shift-Reduce) Parsing Scheme

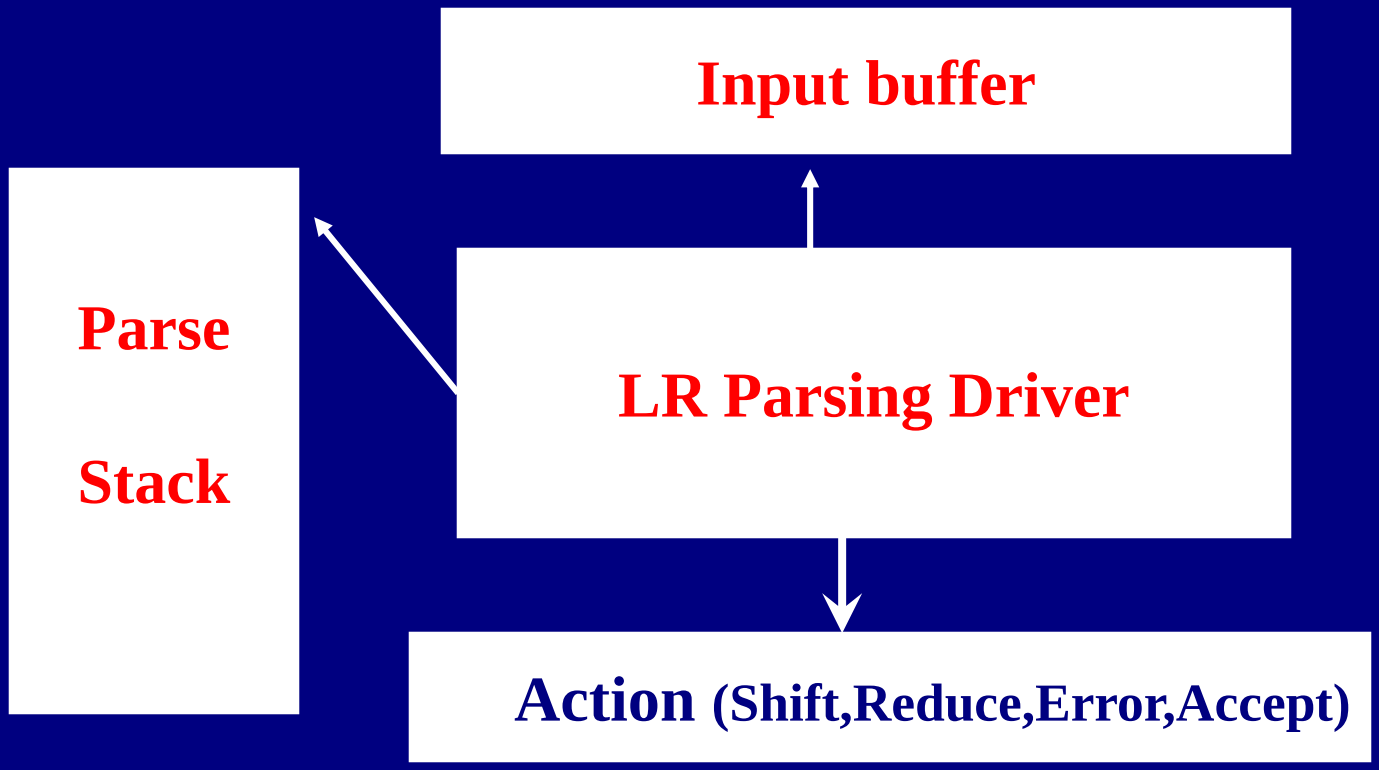


Table Driven LL Parsing

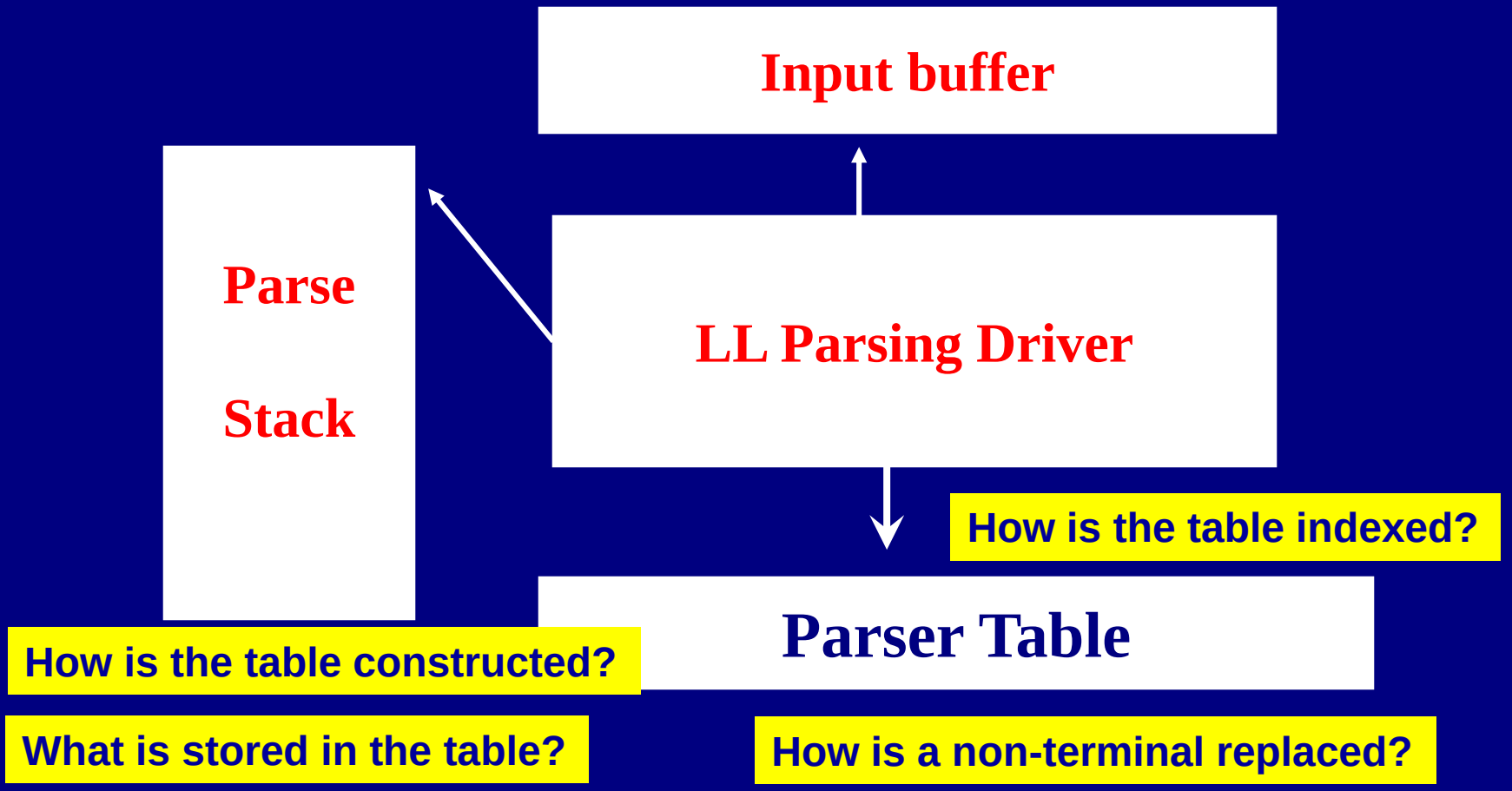


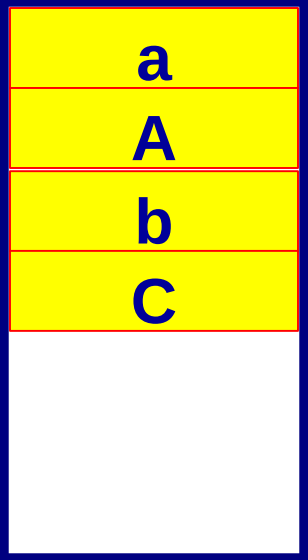
Table Driven LL Parsing

adbc

S → a A b C
A → dA | λ
C → c

Table Driven LL Parsing

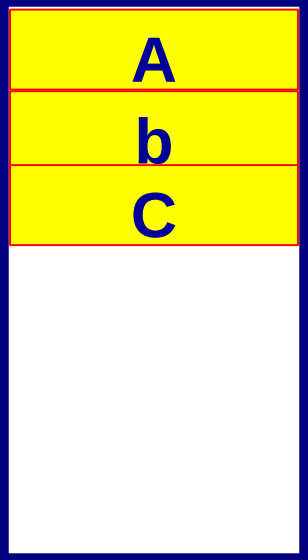
adbc



S → a A b C
A → dA | λ
C → c

Table Driven LL Parsing

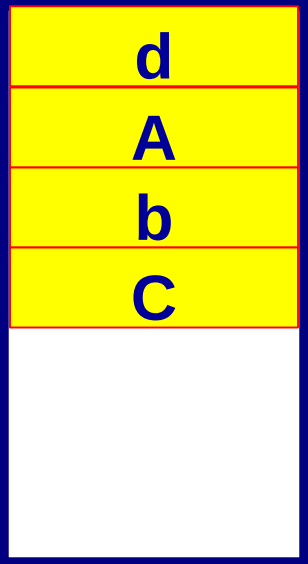
dbc



S → a A b C
A → dA | λ
C → c

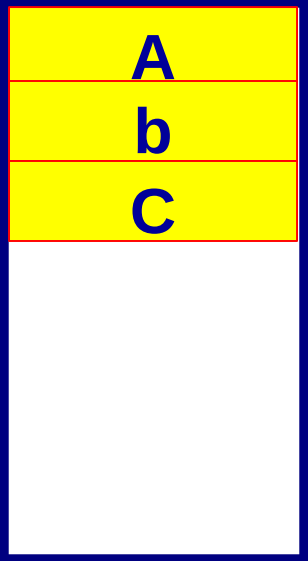
Table Driven LL Parsing

dbc



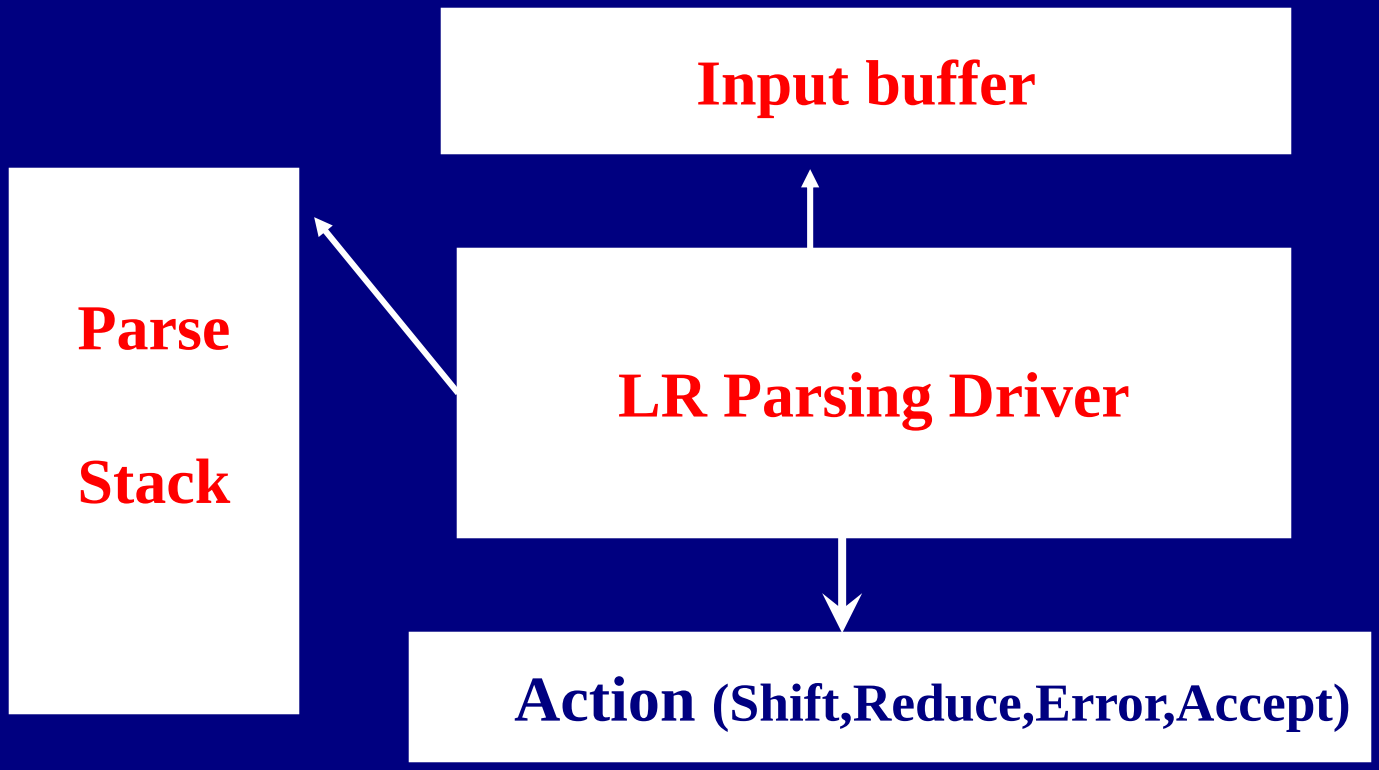
S → a A b C
A → dA | λ
C → c

Table Driven LL Parsing



S → a A b C
A → dA | λ
C → c

LR (Shift-Reduce) Parsing Scheme



LR Parsing

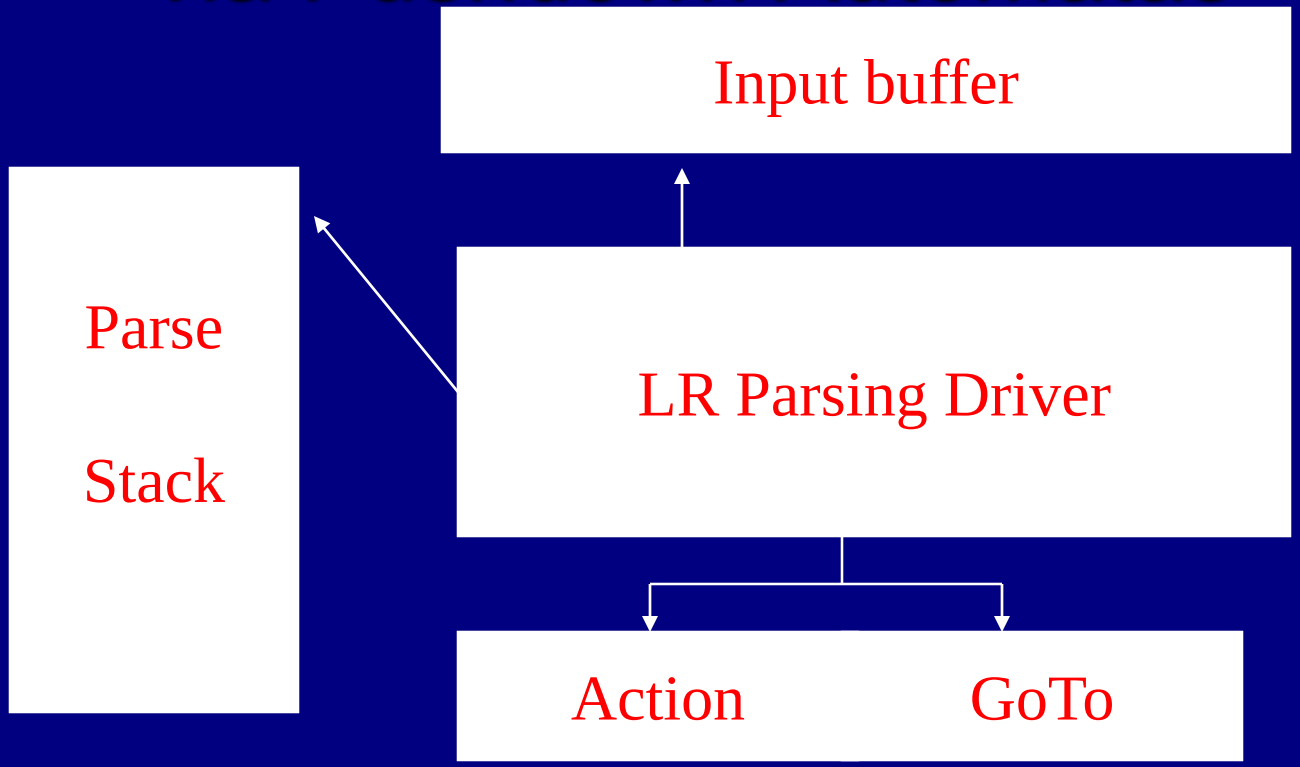
- The key to LR parsing:

  Identify the ***handle*** when it shows up on the top of the parse stack.
  - Not easy especially when λ productions exist
- Parse table driven bottom-up parsing is called LR(k) parsing:
  - L: Left-to-right
  - R: Rightmost derivation in reverse
  - k: number of input symbols of look-ahead to help make shift-reduce decisions.

LR Parsing

- LR parsing is attractive
  - can recognize virtually all programming language constructs
  - the most general non-backtracking parsing method known.
  - more powerful than predictive parsing.
  - efficient LR parser generators available
  - can detect a syntax error as soon as the error token shows up.

LR (Shift-Reduce) Parsing Scheme via Pushdown Automatas



- How is the action table indexed? How about the GoTo table?
- What are the actions associated with each state?
- Each state summarizes the information contained in the stack below it.

LR Parsing Tables

- CFG cannot be recognized by NFA.
- CFG needs be recognized by pushdown automata (PDA).
  - States
    - This is the key to recognize a handle
  - GoTo Table
  - Action Table for stack operation.
    - Shift: like push a token
    - Reduce, replacing RHS by LHS of a production rule.

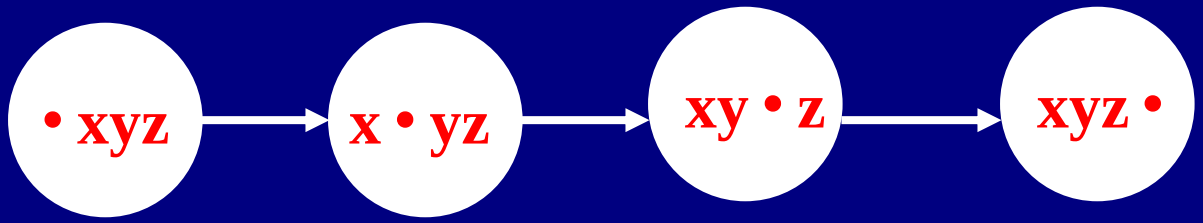
LR(0) Parsing

- Configuration or LR(0) item
  - A production of G with a dot at some position of the right hand side

    A → X1 …. Xi • Xi+1 … Xj
  - The dot marks how much of the production has already been matched.
  - A production A → XYZ yields four items
    - A → • XYZ
    - A → X • YZ
    - A → XY • Z
    - A → XYZ •

LR(0) Items

- Production rules with a " • " in them are called LR(0) items, or just Items.
- Items can be viewed as the states of an ***NFA*** recognizing viable prefixes.



- Everything to the left of the " • " represents symbols already on the stack, and everything to the right of dot represents symbols yet to see.

You may think the dot as a bookmark

– A *configuration set* contains all the items that may be applicable at a given point of parse.

stmt → • id := *expr;*

stmt → • id : *stmt;*

stmt → • id;

represents a situation in which an **id** can be matched as part of any of three productions.

– In general, an item with the dot at the extreme left end is said to *predict* that production and a dot at the extreme right end is said to *recognize* that production.

– An item can be represented as a pair of integer

(**production, position of dot**)

Example

Configuration or Item

S → • aABe

Grammar G

- S → aABe
- A → Abc | b
- B → d

Configuration set or Set of Item

A → • Abc

A → • b

Closure(I)

Symbolic representation of parsing states.

- *All productions induced from I !*
  - **I** is a set of items
- Every item in **I** is added to *closure*(**I**)
- If $A \rightarrow \alpha \bullet B \beta$ is in closure(**I**) and $B \rightarrow \gamma$ is a production, then add the item $B \rightarrow \bullet \gamma$ to **I**.
- Repeat until no more item can be added.

$A \rightarrow \alpha \bullet B \beta$ indicates we might next see a substring derivable from $B \beta$.

So if $B \rightarrow \gamma$ is a production, we might also expect a substring derivable from γ .

Example

Grammar G

- S → aABe
- A → Abc | b
- B → d

Configuration or Item

S → • aABe

NFA state

Configuration set or Set of Item

A → • Abc

A → • b

Set of states

Closure

S → a • ABe

A → • Abc

A → • b

DFA state

goto function

- Actually state-transition function!
- goto(I, X), where
  - I is a set of items and
  - X is a grammar symbol,

  is defined to be the *closure* of the set of all items $[A \rightarrow \alpha X \bullet \beta]$ such that $[A \rightarrow \alpha \bullet X \beta]$ is in **I**.

  It simply means move the dot over X.

  Intuitively, if $A \rightarrow \alpha \bullet X \beta$ is in I, the state indicates α is a viable prefix, so αX will also be a valid viable prefix, and item $A \rightarrow \alpha X \bullet \beta$ will be in the set of the next state.

State set construction (forward)

begin

C := {closure({S' → • S})}; // the initial state

repeat

for each set of item I in C and each symbol X

such that GoTo(I,X) is not empty and not in C,

add GoTo(I,X) to C

until no more sets of items can be added to C

// finally C is the set of symbolic states.

end

Note:

- *G' is the augmented grammar for G. It is G with a new start symbol S' and production S' → S.*
- *The sets-of-items is similar to the subset construction to turn the initial NFA into a DFA.*

算出所有的States

Automata construction

Example

Start with S' → • S

- S → aABe
- A → Abc | b
- B → d

State construction: example

Start with S' → • S

- S → aABe
- A → Abc | b
- B → d

0

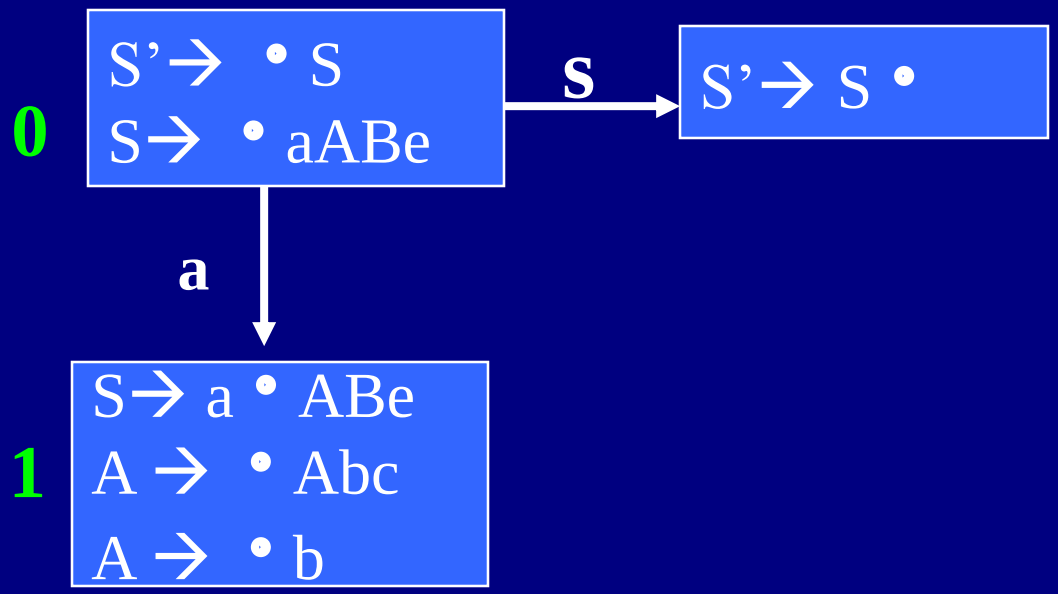
S' → • S
S → • aABe

C := {closure({S' → • S})};

State construction: example

- S → aABe
- A → Abc | b
- B → d

Start with S' → • S

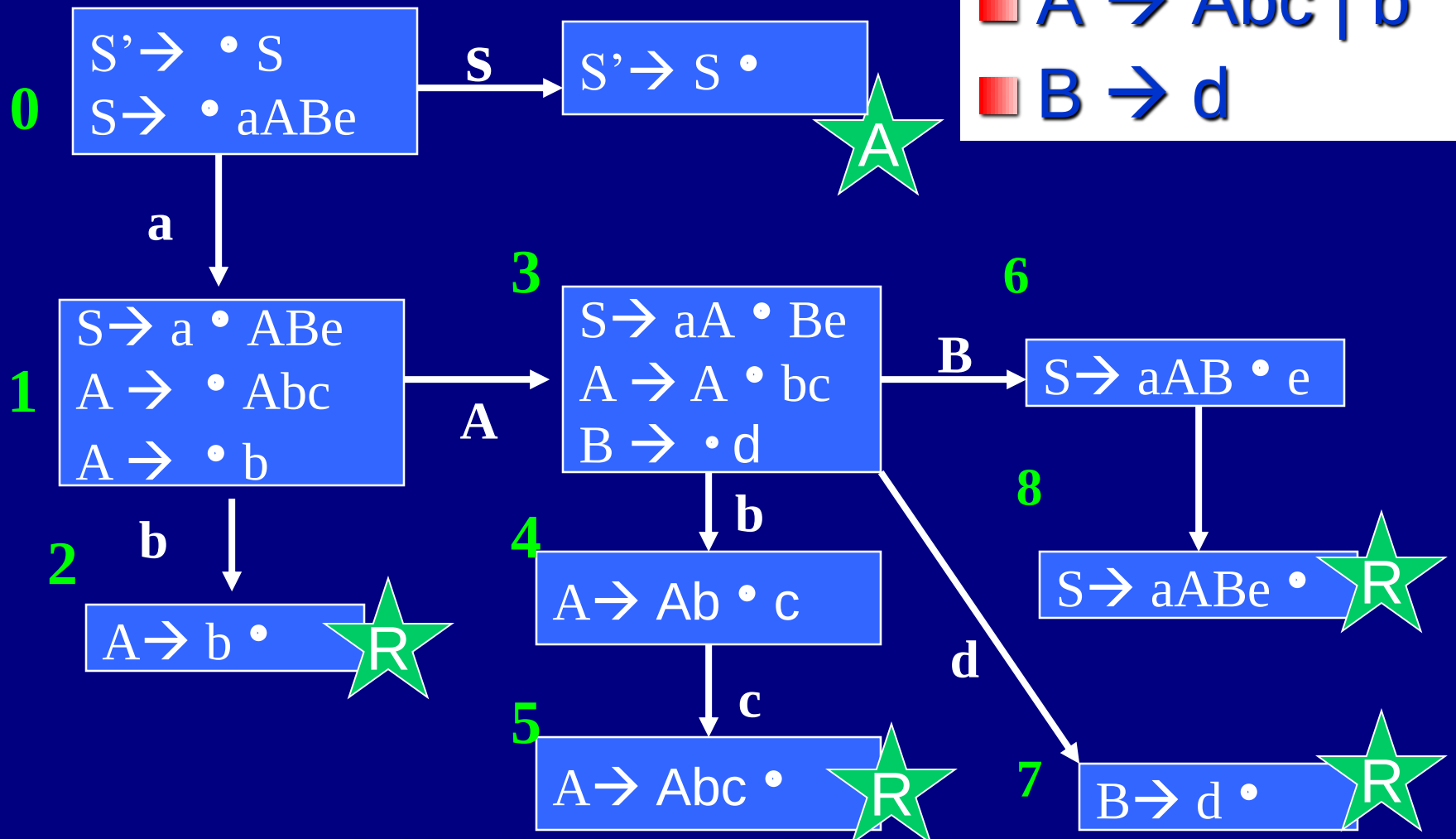


for each set of item I in C and each symbol X such that GoTo(I,X) is not empty and not in C, add GoTo(I,X) to C

State construction: example

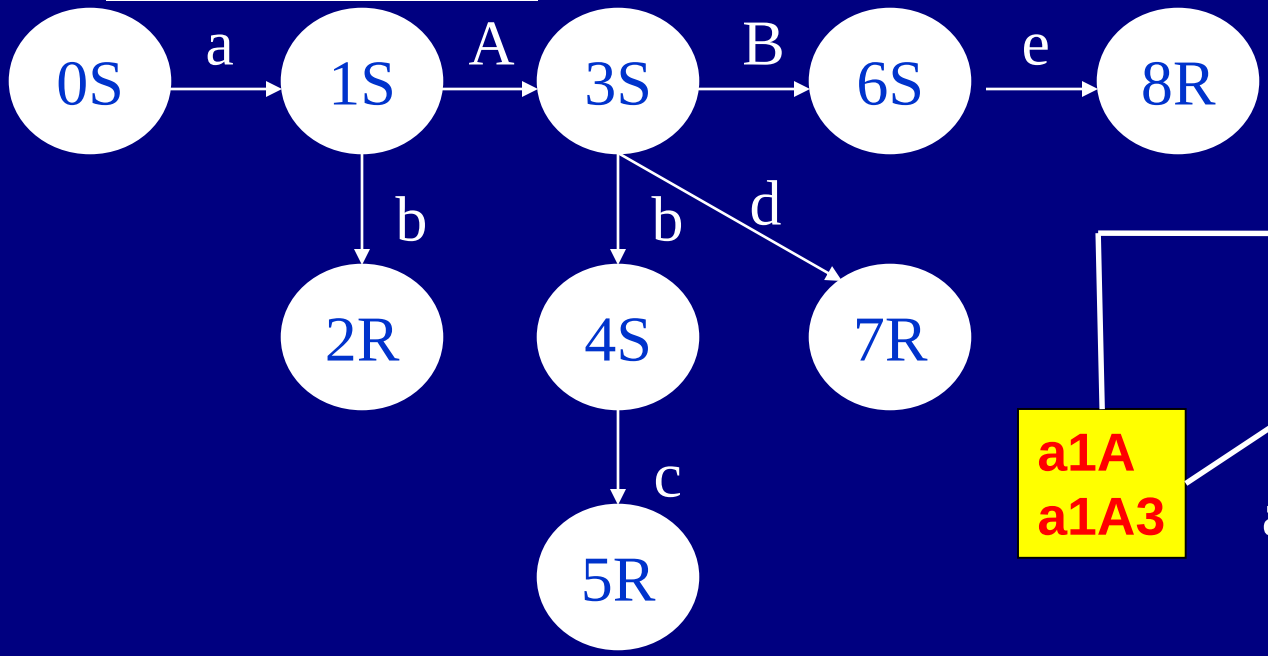
- S → aABe
- A → Abc | b
- B → d

Start with S' → • S



LR(0) Parsing steps: example

S → aABe
A → Abc | b
B → d



Input string: abbcde

Stack	Input
0	abbcde
a1	bbcde
a1b2	bcde
a1A3	bcde
a1A3b4	cde
a1A3b4c5	de
a1A3d7	e
a1A3B6e8	
0S	
Accept	

a1A
a1A3

Another example

- Consider the following G

1 S → E$

2 E → E + T | T

3 T → id | (E)

Closure ({ S → • E $}) == {

S → • E $

E → • E + T

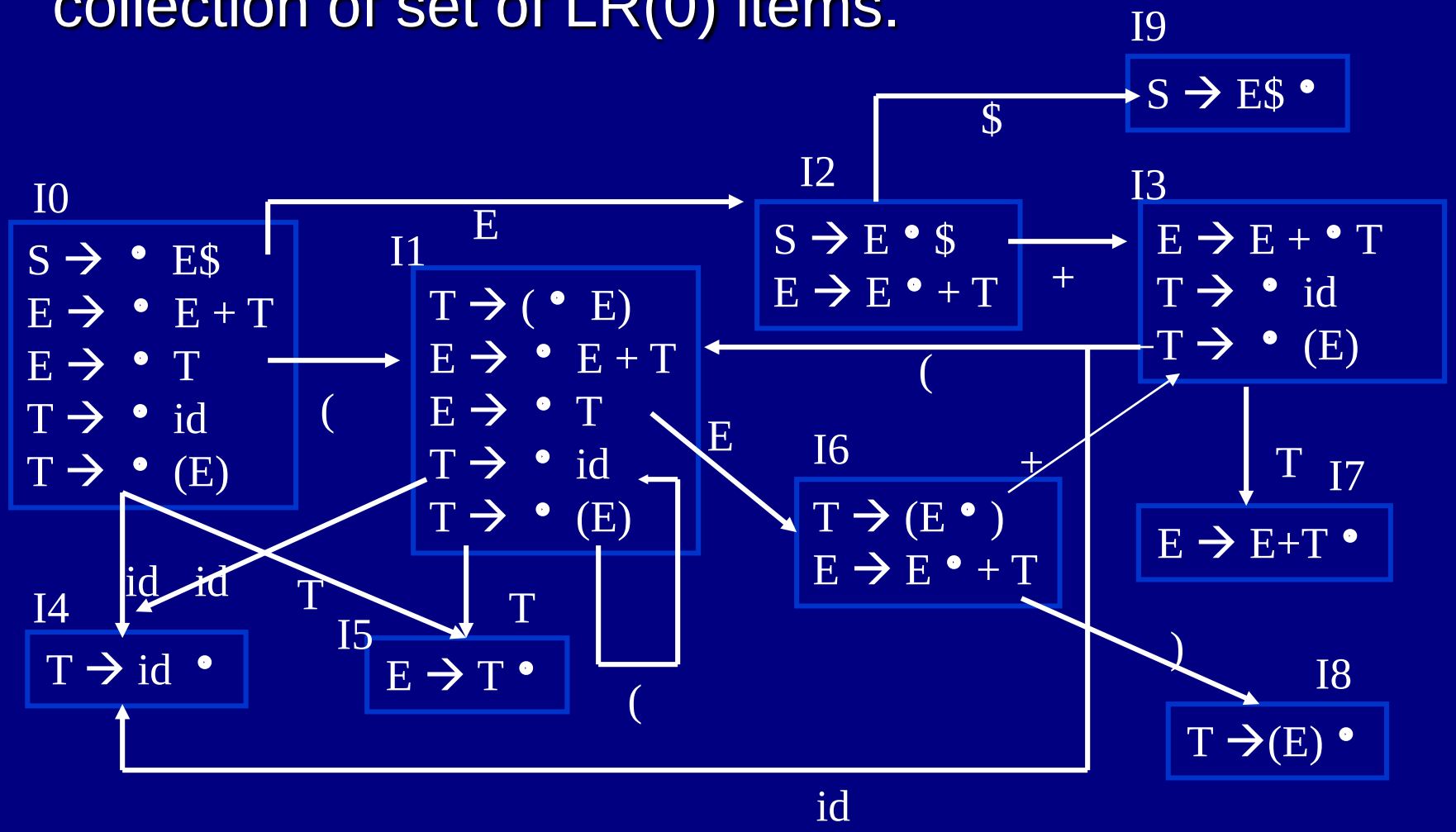
E → • T

T → • id

T → • (E) }

Example: Building CFSM

CFSM (Characteristic Finite State Machine) or collection of set of LR(0) items.



Exercise

- What is the meaning of state I3

  E → E + • T
  T → • id
  T → • (E)

  Viable prefix E+ has been recognized. The parser expects to see **id** or (E) from the input buffer.

- What is the meaning of state I8

  T →(E) •

  Viable prefix (E) has been recognized. We may reduce it to T.

LR(0) Parser Table

states	action	GoTo						
		E	+	(	)	Id	T	$
I0	S	I2		I1		I4	I5	
I1	S	I6		I1		I4	I5	
I2	S		I3					I9
I3	S			I1		I4	I7	
I4	R(3a)							
I5	R(2b)							
I6	S		I3		I8			
I7	R(2a)							
I8	R(3b)							
I9	A							

Action Table

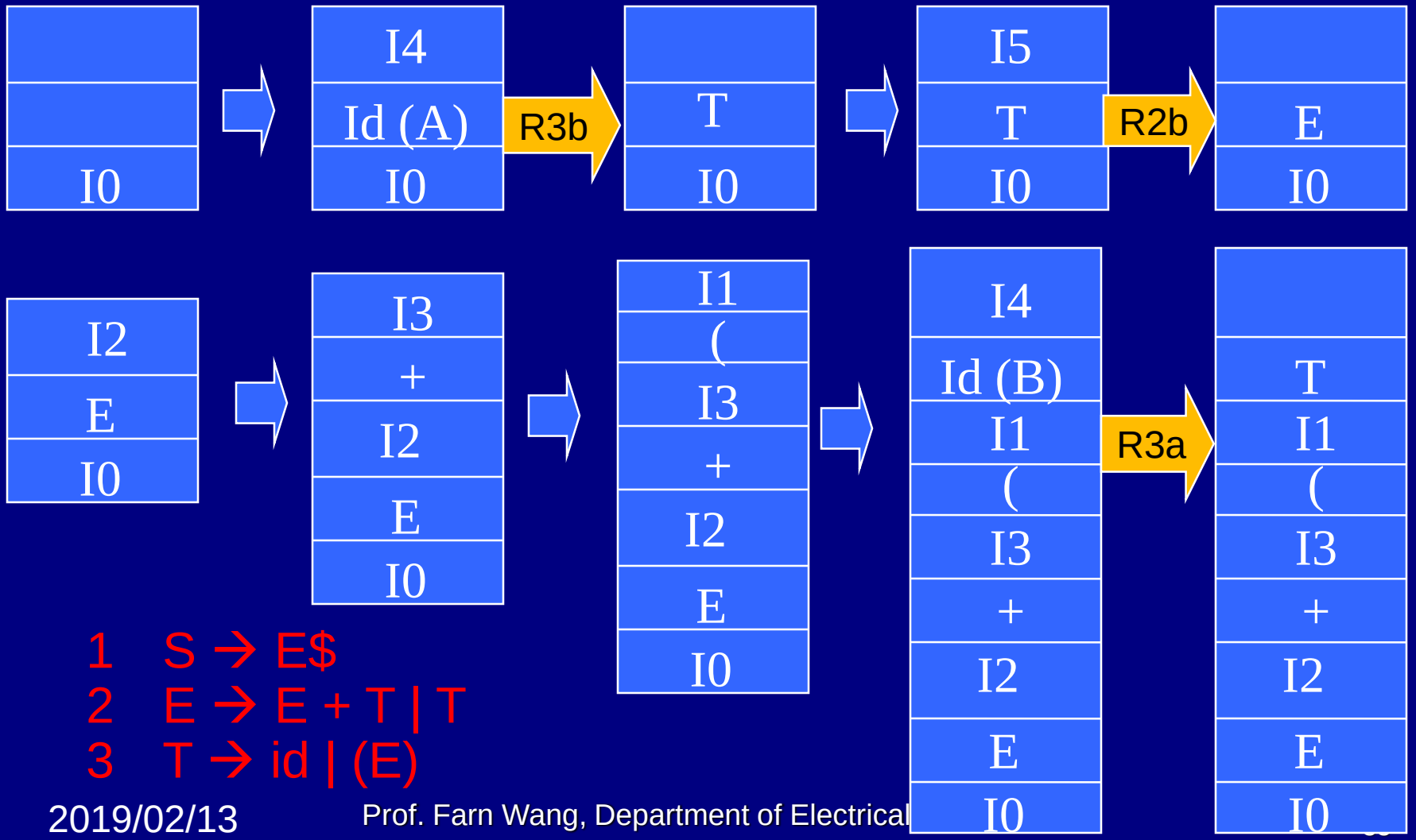
- Handling the stack of PDA
- Deciding whether the CFSM has reached the end of the handle. If it is, a reduction or accept action is indicated, otherwise, a shift action is needed.
- Actions are {shift, reduce1, reduce2, … accept, error}
- Shift-reduce or reduce-reduce conflict may occur to inadequate states.
- Inadequacies are usually resolved by using look-ahead in the action table.

LR(0) Parser Table

states	action	GoTo						
		E	+	(	)	Id	T	$
I0	S	I2		I1		I4	I5	
I1	S	I6		I1		I4	I5	
I2	S		I3					I9
I3	S			I1		I4	I7	
I4	R(3a)							
I5	R(2b)							
I6	S		I3		I8			
I7	R(2a)							
I8	R(3b)							
I9	A							

Parsing with LR(0) table

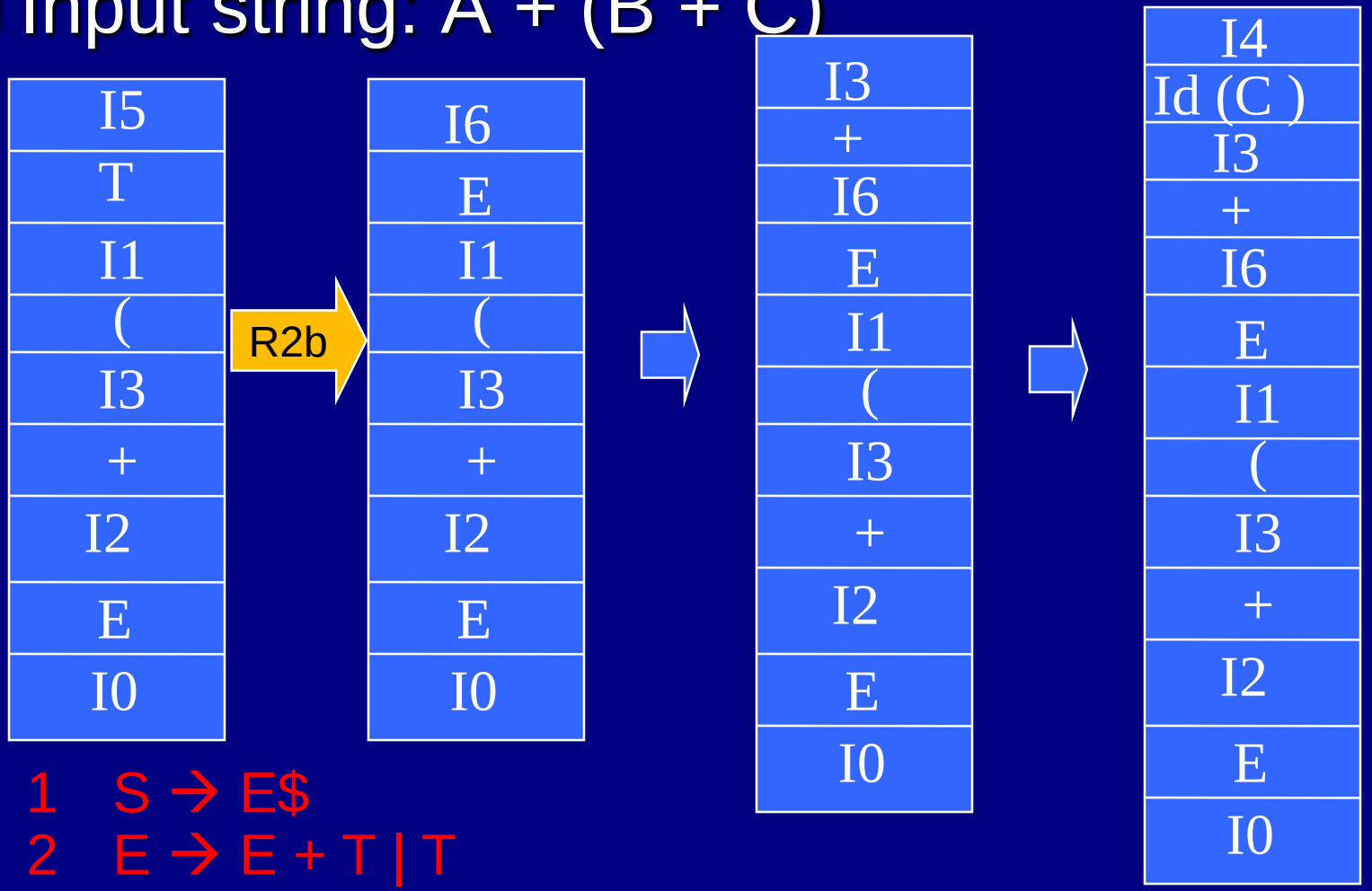
- Input string: A + (B + C)



1 S → E$
2 E → E + T | T
3 T → id | (E)

Parsing with LR(0) table (cont.)

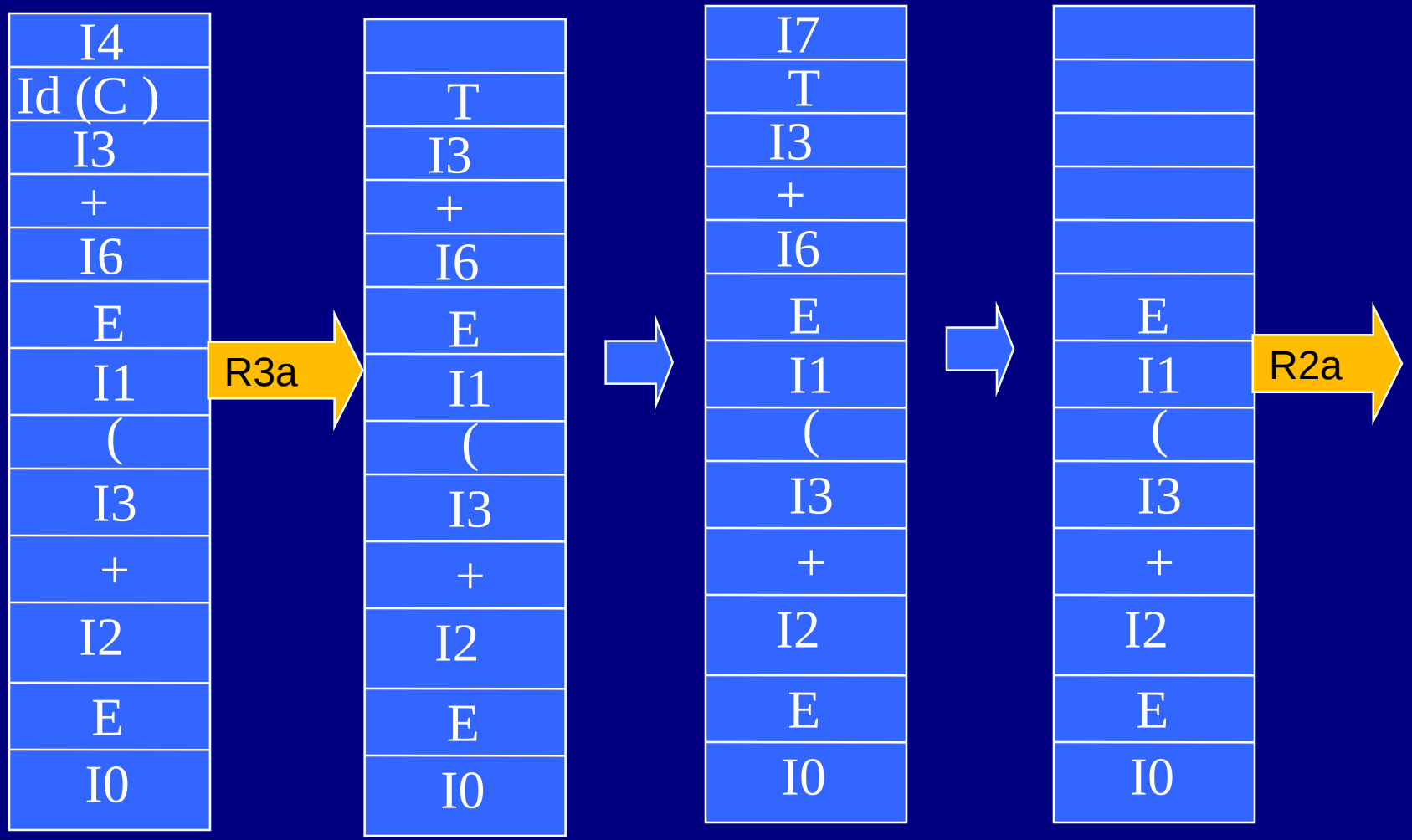
- Input string: A + (B + C)



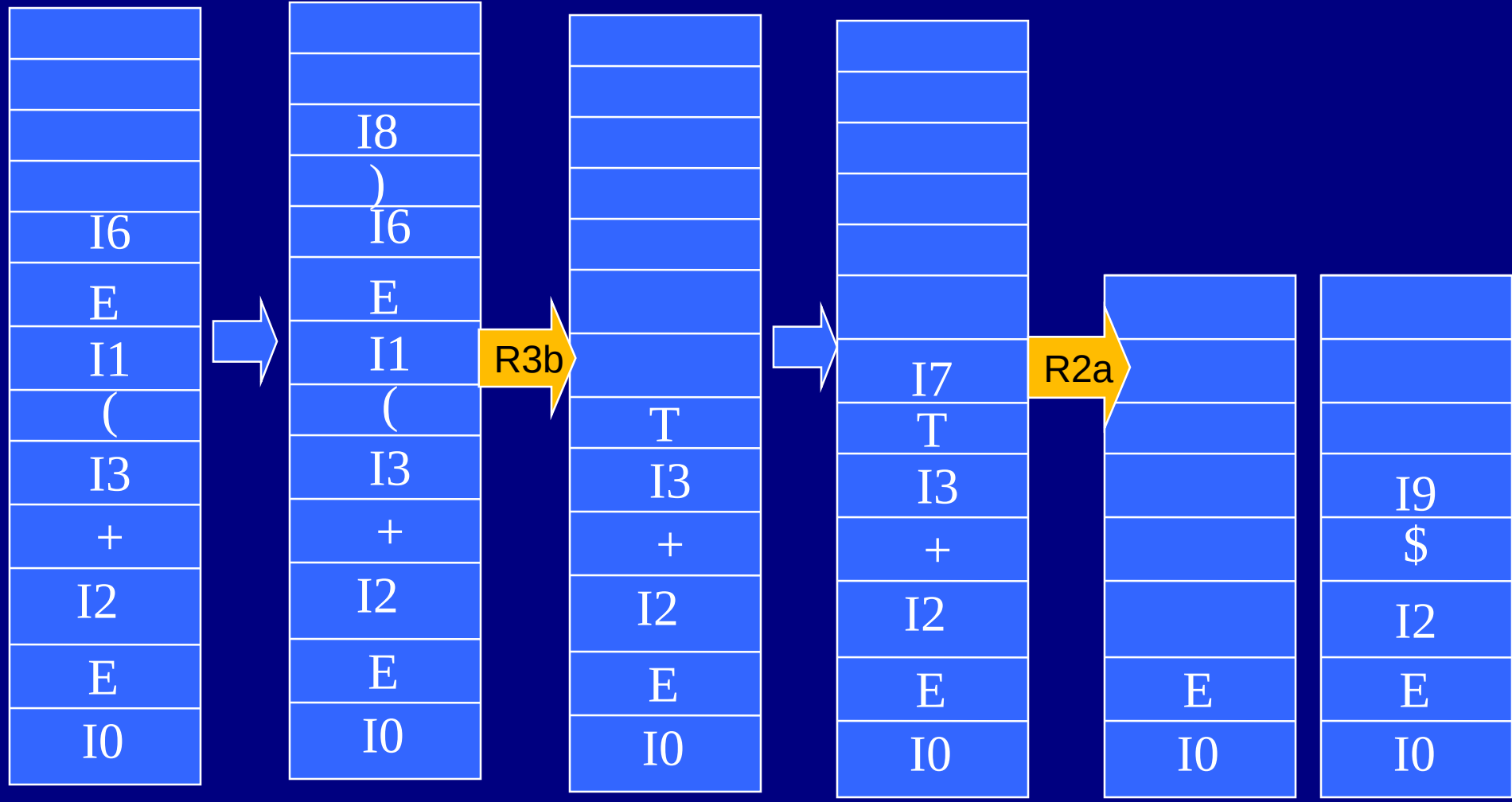
1 S → E$
2 E → E + T | T
3 T → id | (E)

Parsing with LR(0) table (cont.)

- Input string: A + (B + C)



Parsing with LR(0) table (cont.)



Top-Down Vs. Bottom-Up

- Parser Table
  - Top-down: Action table only; indexed by non-terminals and terminals
  - Bottom-up: Action table and GoTo table.
    - Goto table indexed by states and grammar symbols.
    - Action table indexed by state and terminals.
- Stack
  - Top-down: keep grammar symbols on stack.

    A list of what the parser expect to see in the *future*
  - Bottom-up: keep states on stack.

    A record of what the parser has already seen in the *past*
- Action
  - Top-down: pop terminals, expand non-terminals,accept
  - Bottom-up: shift, reduce, accept

Top-Down Vs. Bottom-Up

- Main function
  - Top-down: which production to be used to expand a non-terminal
  - Bottom-up: when can we replace the right-hand side of a production by its left-hand side non-terminal

Recursion in Bottom-Up Parsing

- Left recursion is good for LR
  - but bad for LL.

list → list, num | num

Stack	Input	Action
$	**1, 2, 3**	
$ num	**,2, 3**	**Shift a num**
$list	**,2, 3**	**reduce**
$list, num	,3	shift
$list	,3	reduce
$list, num		shift
$list		reduce

Recursion in Bottom-Up Parsing

- Right recursion needs some stack space

list → num | num, list

Stack	Input	Action
$	1, 2, 3	shift
$ num	,2, 3	shift
$ num, num	,3	shift
$ num, num , num		reduce
$ num, num, list		reduce
$ num, list		reduce
$list		reduce

Quick Review of LR(0)

- LR(0) Item: e.g. E → E + • T
  - an NFA state
- Set-of-Items
  - Closure of a set of NFA states
  - a DFA state
- CFSM (DFA for LR parsing)
- Parse table
  - Action Table
    - shift, reduce, accept, error
  - GoTo Table

LR(0) Parser Table

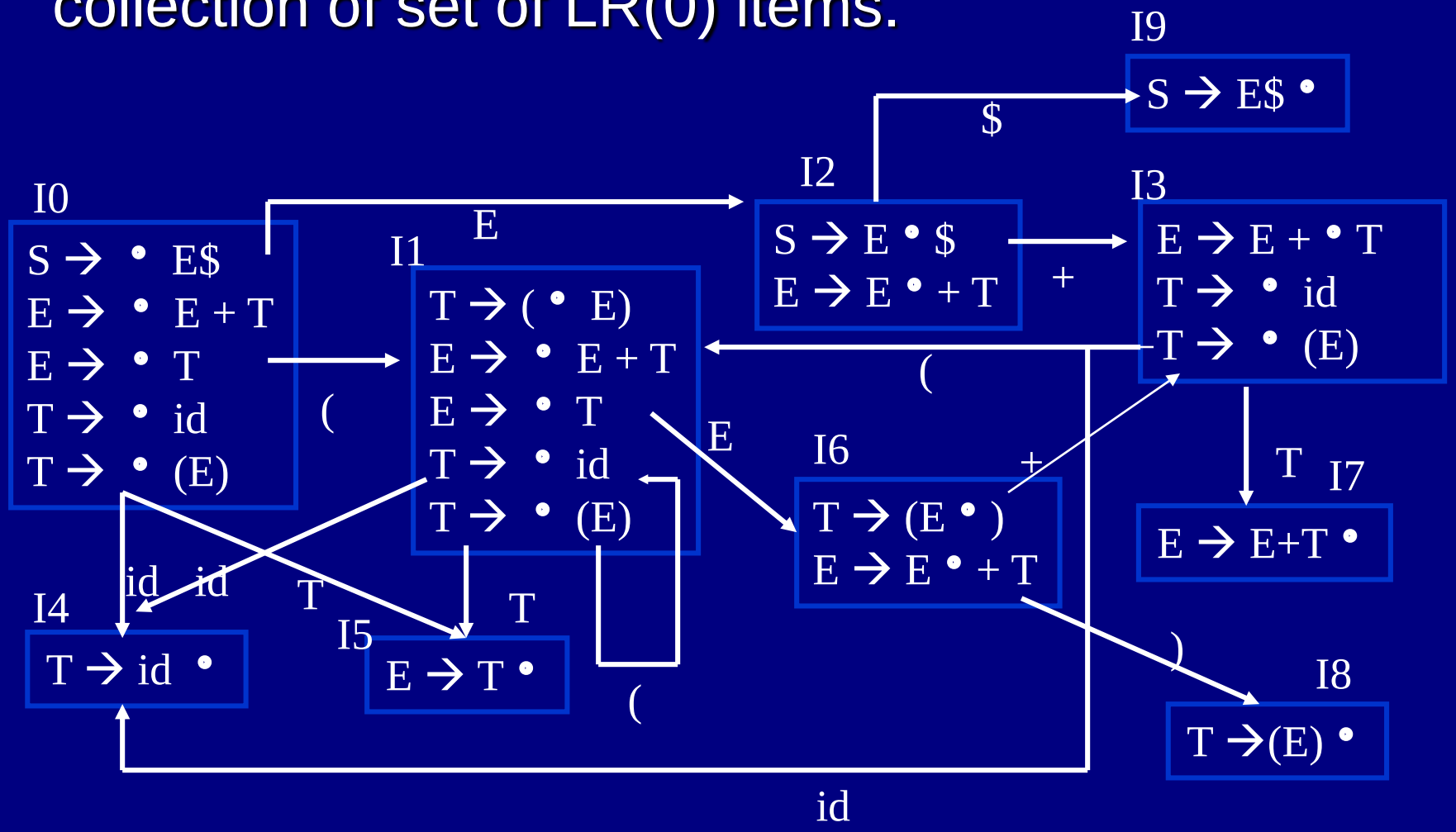
states	action	GoTo						
		E	+	(	)	Id	T	$
I0	S	I2		I1		I4	I5	
I1	S	I6		I1		I4	I5	
I2	S		I3					I9
I3	S			I1		I4	I7	
I4	R(3a)							
I5	R(2b)							
I6	S		I3		I8			
I7	R(2a)							
I8	R(3b)							
I9	A							

LR(0) Parse Table

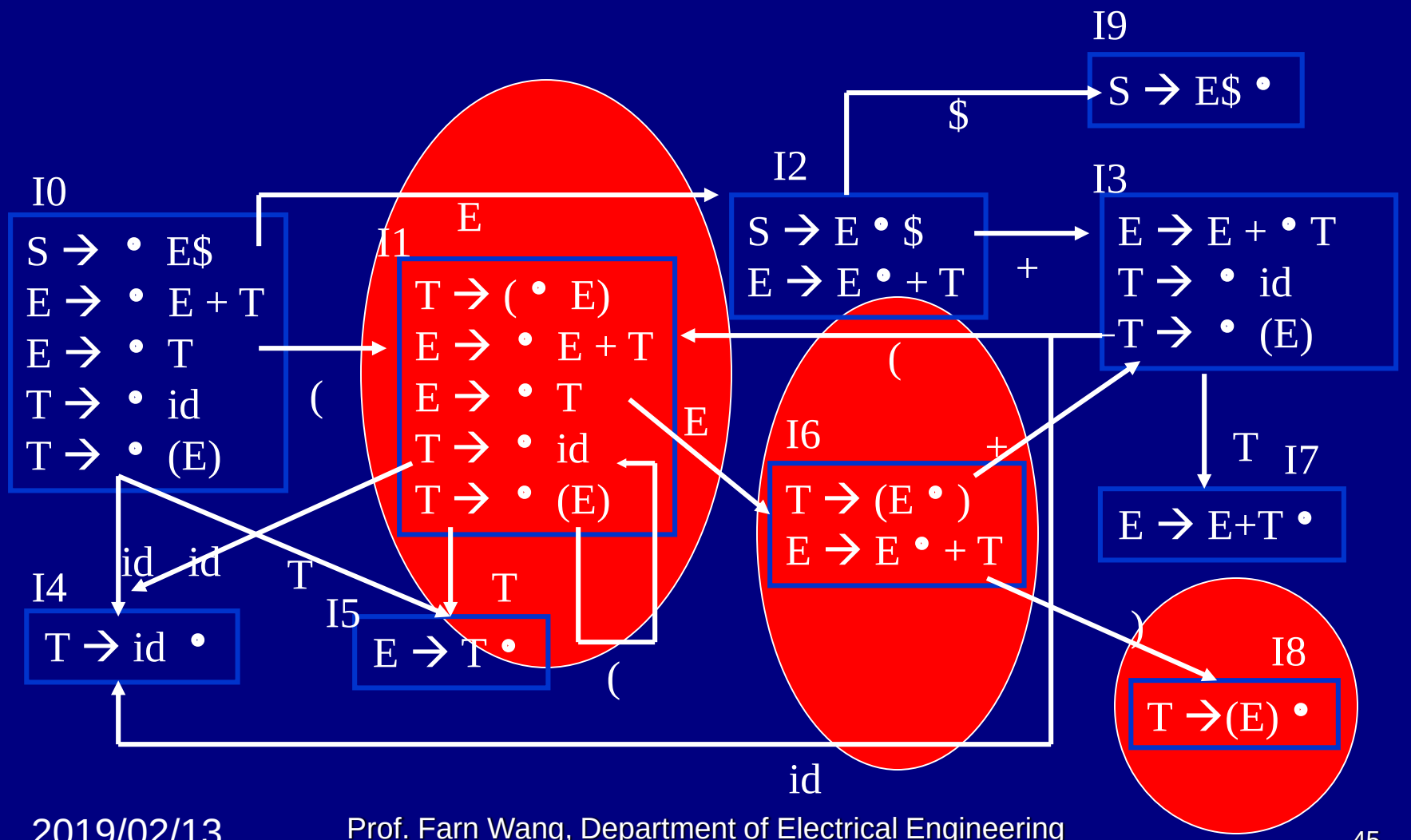
- If a state ***S*** contains an item with " • " at the end, add reduction (R,#) to the action table.
- If a state ***S*** contains a transition labeled "**b**" to state P, add shift (S) to the action table, and add (P) to the GoTo table entry [***S***,**b**].
- If a state ***S*** contains a transition labeled "**A**" to state P, add (P) to the GoTo table entry [***S***,A].

Example: Building CFSM

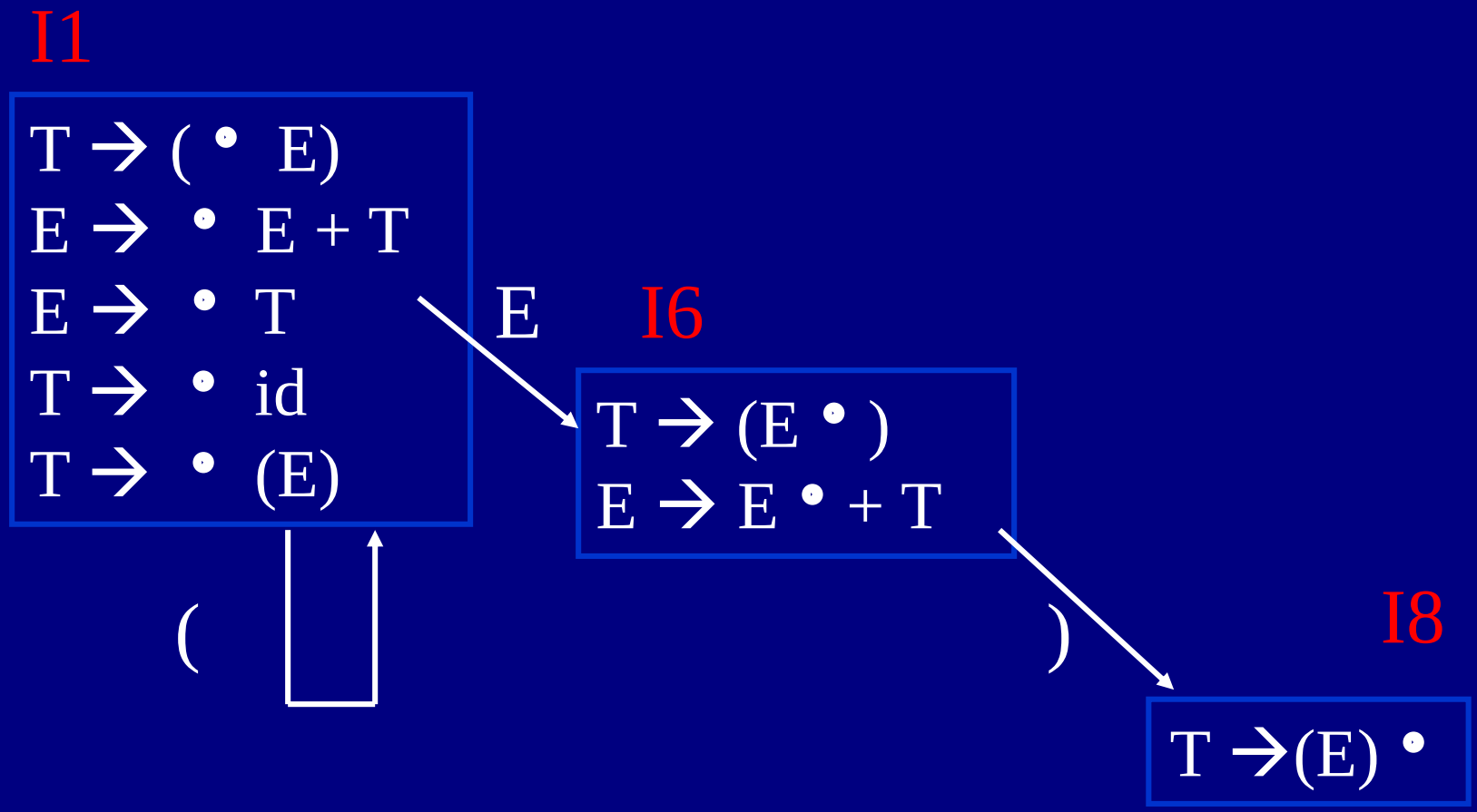
CFSM (Characteristic Finite State Machine) or collection of set of LR(0) items.



Example CFSM



Example CFSM



LR(0) Parser Table

states	action	GoTo						
		E	+	(	)	Id	T	$
I0	S	I2		I1		I4	I5	
I1	S	I6		I1		I4	I5	
I2	S		I3					I9
I3	S			I1		I4	I7	
I4	R(3a)							
I5	R(2b)							
I6	S		I3		I8			
I7	R(2a)							
I8	R(3b)							
I9	A							

Action Conflicts in LR(0) Table

- Suppose the configuration set is as follows:

  $S \rightarrow E \bullet$

  $E \rightarrow E \bullet + T$

  the first item calls for a reduction action

  the second one calls for a shift action
- Such conflicts may be avoided by adding one or more look-ahead symbols
- LR(0) is not practical since it is too easy to have shift-reduce conflicts.

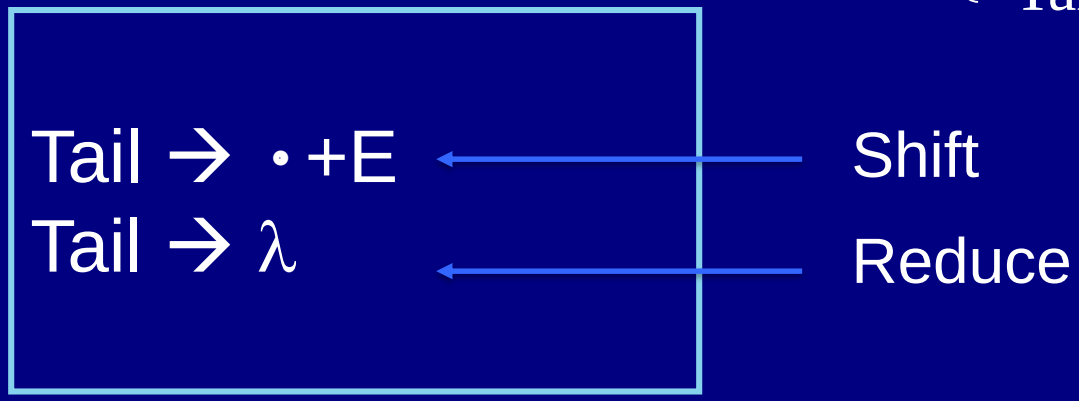
Consider the expression A+B+C, shift is for right associative op and reduction is for left associative op.

Action Conflicts in LR(0) Table

- Is the example CFG in LR(0)?

G_0:
- $E \rightarrow Prefix(E)$
- $E \rightarrow c\ Tail$
- $Prefix \rightarrow b$
- $Prefix \rightarrow \lambda$
- $Tail \rightarrow +E$
- $Tail \rightarrow \lambda$

- Answer is NO !
- Since there is a λ production !
- There will be s state contains items as



LR(0) is not suitable for real PL

1) λ-productions

   A → aB | λ

   both shift and reduce actions are possible

   e.g. C → α • A β

2) Operator precedence

   B+C+D vs. B+C*D

   if we get to a state with B+C •α

   The action should be shift if next token is *, and reduce if the next token is +.

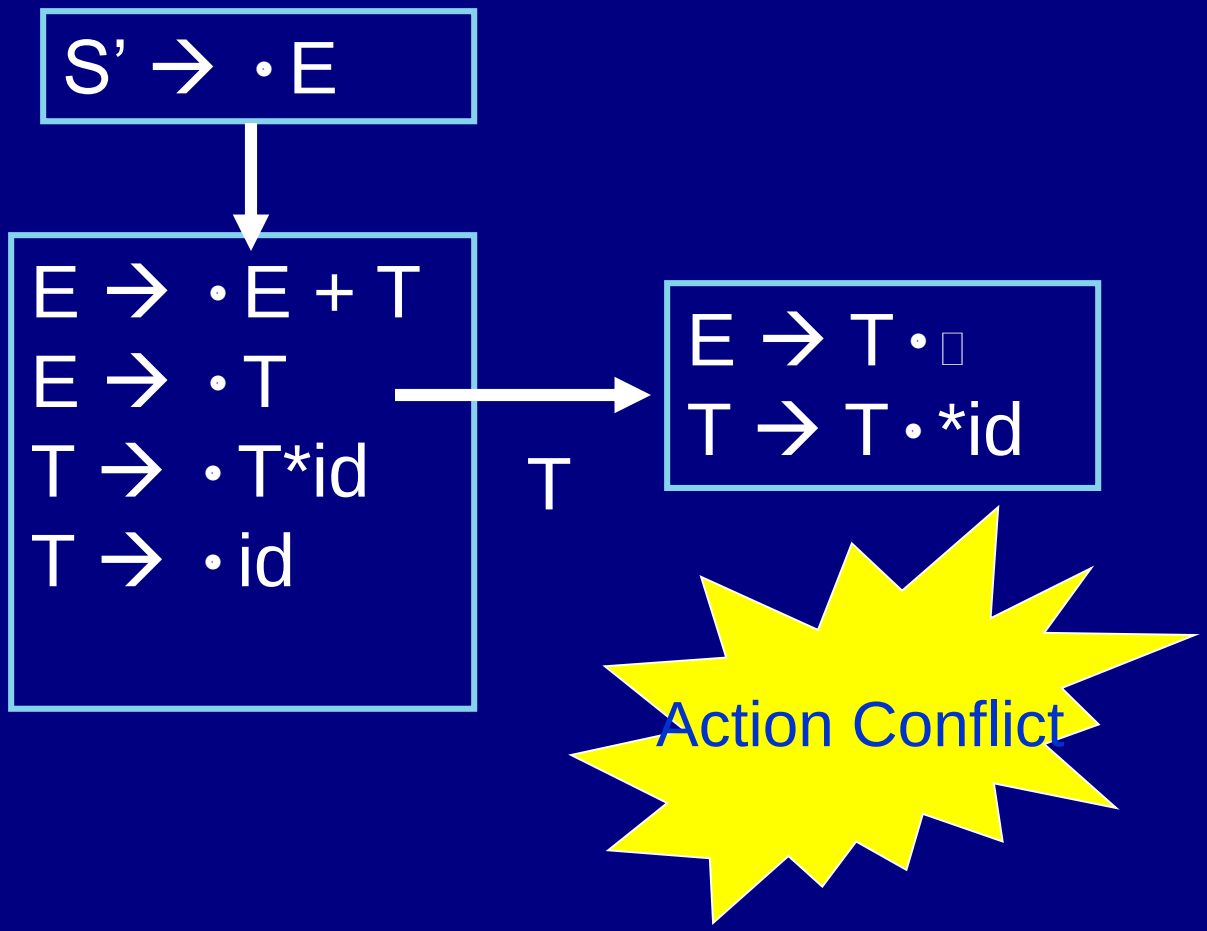
Show a CFG that is not LR(0)

- Show there is a state contains action conflict.

Example:

CFG

E → E + T
E → T
T → T*id
T → id



SLR(1) Parser Table

- SLR(1) is different from LR(0) only in the action table where a reduction of A → α • is called for. In SLR(1), we only reduce when the look-ahead symbol is in the set FOLLOW(A).

E → T •
T → T • *id

We only reduce when the lookahead is + *(i.e. follow(E)=+)*

- SLR(1) is much more useful than LR(0)
- However, the use of FOLLOW set for look-ahead is not precise. This may cause a) R-R conflict and b) incorrect reduction. So LR(1) is still needed.

SLR(1) Parse Table

Action table and GoTo Table can be merged.

- If a state ***S*** contains an item A → α • , with " • " at the end, add reduction (R,#) to the table entry [***S, b***], where ***b*** is in Follow(A).
- If a state ***S*** contains a transition labeled "**X**" to state P, add (S,P) to the table.

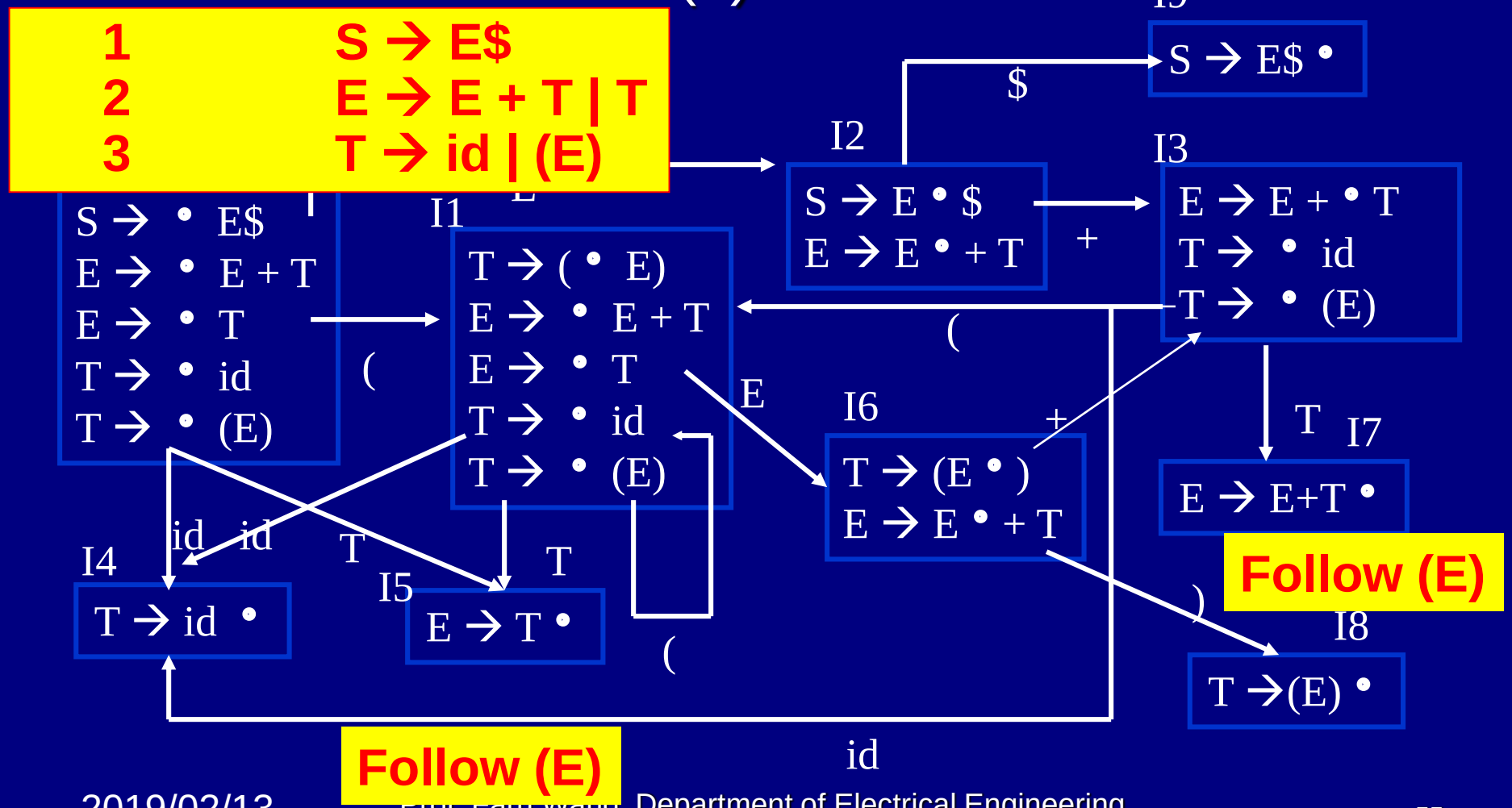
Merged table is now indexed by state and grammar symbols (both non-terminals and terminals). It is the same size as LR(0) table, with less empty entries. SLR parsers were popular in early 70's.

SLR(1) Parser Table

states	Action/GoTo						
	E	+	(	)	Id	T	$
0	S2		S1		S4	S5	
1	S6		S1		S4	S5	
2		S3					S9
3			S1		S4	S7	
4		R(3a)		R(3a)			R(3a)
5		R(2b)		R(2b)			R(2b)
6		S3		S8			
7		R(2a)		R(2a)			R(2a)
8		R(3b)		R(3b)			R(3b)
9							A

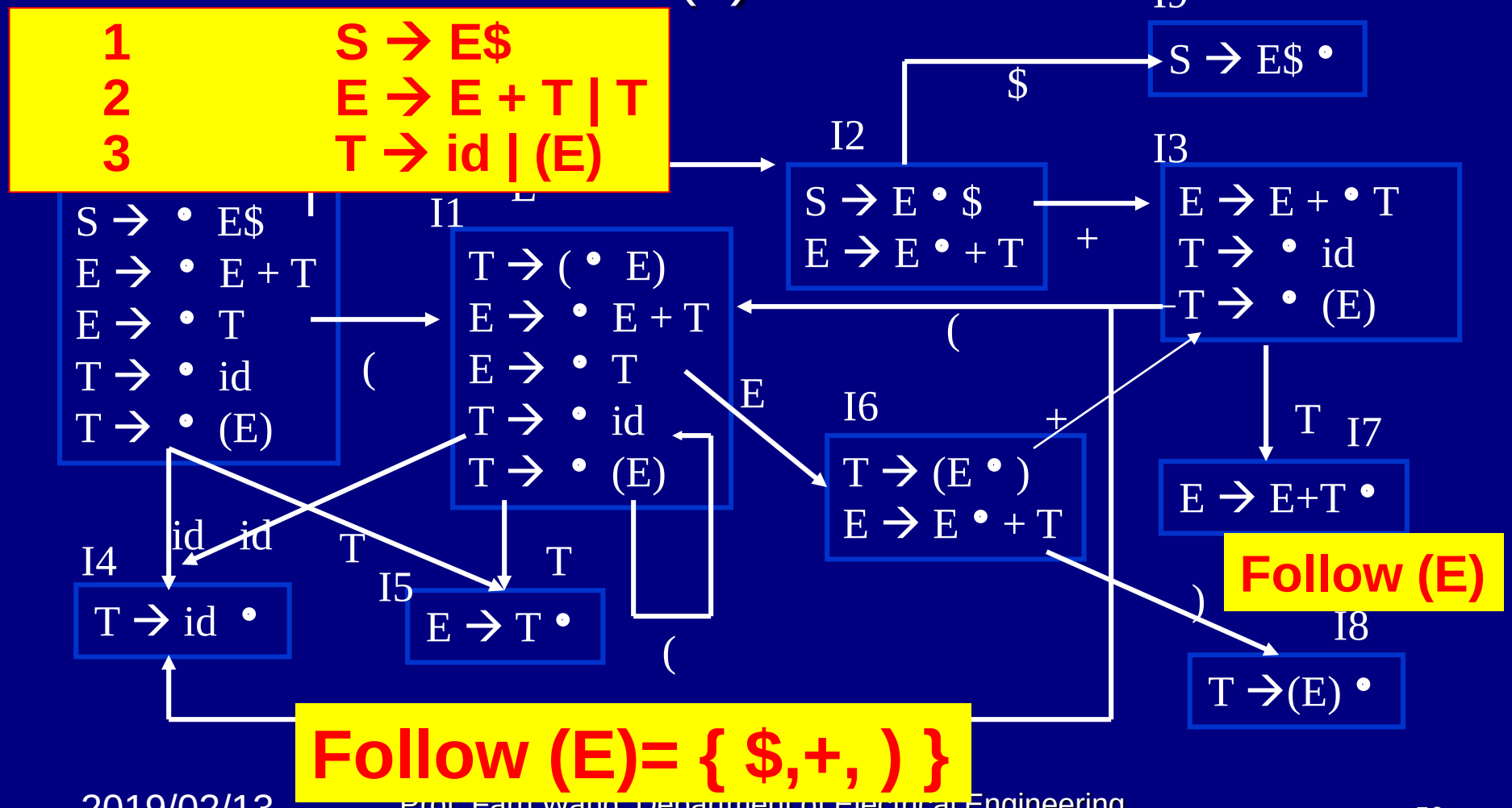
Example: Building CFSM

CFSM (Characteristic Finite State Machine) or collection of set of LR(0) items.



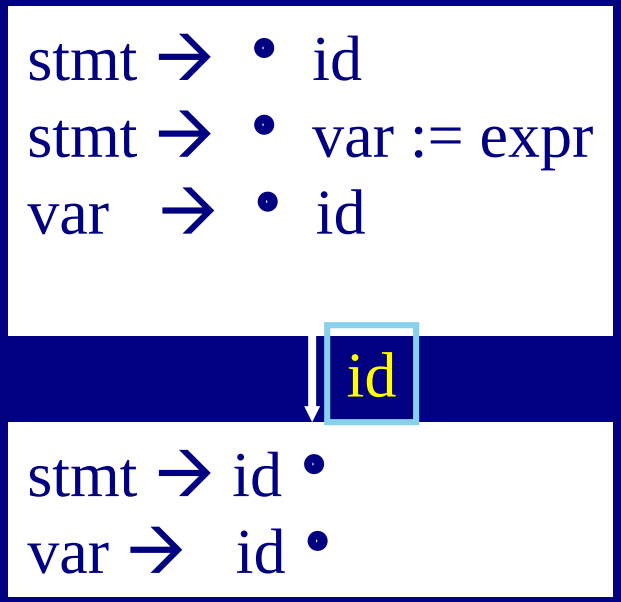
Example: Building CFSM

CFSM (Characteristic Finite State Machine) or collection of set of LR(0) items.



Example of conflicts in SLR(1)

stmt → id
stmt → var := expr
var → id
expr → var

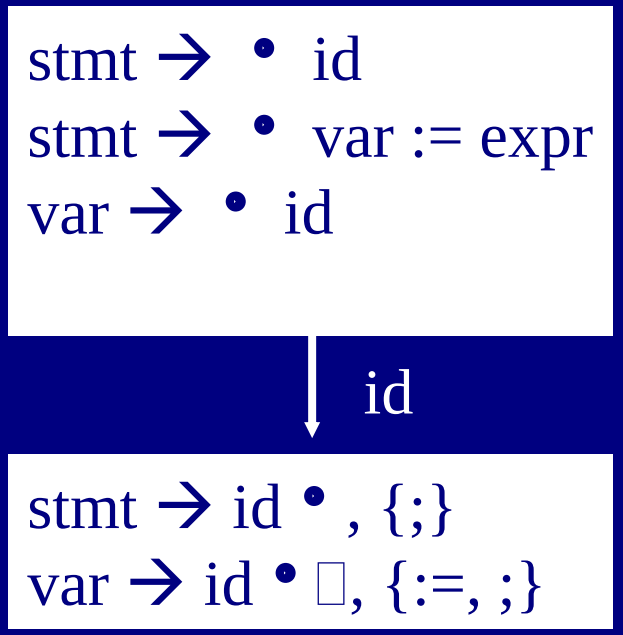


R-R conflict

stmt → id • follow(stmt) = {;}
var → id • follow (var) + ***follow(expr) = follow(stmt) = { ;}***
Since “;” is in the look-ahead set of both productions
SLR(1) can not resolve the conflict
We need something more powerful!!

A Case for More Accurate Lookahead

stmt → id
stmt → var := expr
var → id
expr → var

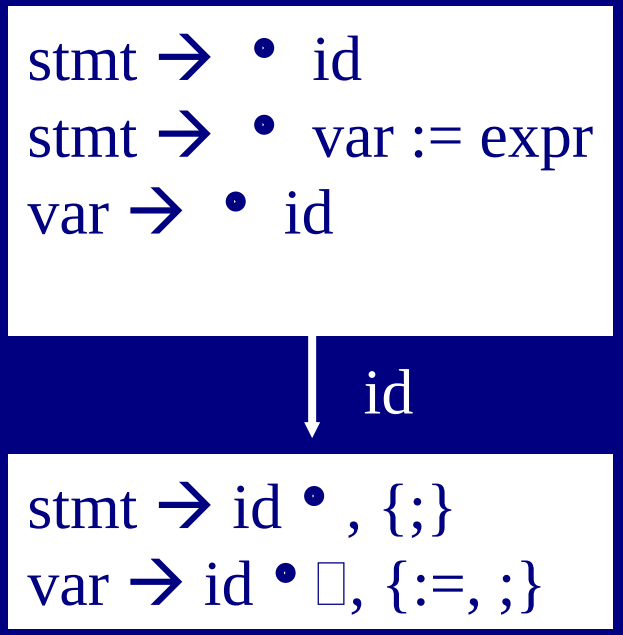


SLR(1)

Since the item var → id started from item **stmt → var := expr,** the semicolon will never follow this item.
The only possible symbol to follow var here will be ":=".
So we need something finer than the simple Follow.

A Case for More Accurate Lookahead

stmt → id
stmt → var := expr
var → id
expr → var



SLR(1)

Observation:

If the item **var → • id** is in the configuration set

{ **stmt → var:= • expr, expr → • var, var → • id** }

then ";" could follow this item.

Context free vs. Context Sensitive

```
Function A (int a)
{
        if (a ==1) call B;
        return;
}
main {
        call A(0);
}
```

Does function A call B?

Context free: Yes
Context sensitive: No

LR(1) Parsing

- An LR(1) item is of the form

  A → X1 … Xi • Xi+1 … Xj, *L*, where *L* is a look-ahead symbol.

  (production, dot position, look-ahead symbol)
- Lookahead symbols have effect only on reduction items, e.g. [A→α, ***a***] calls for a reduction only if the next input symbol is ***a***.
- If a number of LR(1) items differ only in their look-ahead symbols, the following notation can be used:

  A → X1 … Xi • Xi+1 … Xj, {*L1, .. Lm*}
- The ratio of LR(1) items to LR(0) items is about |Vt|, the size of the token set.

Closure and GoTo operation for LR(1)

- Closure

  Repeat

  for each item [$A \rightarrow \alpha \bullet B \beta$, a] in I

  each production $B \rightarrow \gamma$ and

  each terminal b in FIRST(βa)

  such that [$B \rightarrow \bullet \gamma$, b] is not in I, do

  add [$B \rightarrow \bullet \gamma$, b] to I;

  Until no more item can be added

- GoTo (*I*,*X*)

  let J be the set of items [$A \rightarrow \alpha X \bullet \beta$, a] such that [$A \rightarrow \alpha \bullet X \beta$, a] is in I.

  return closure(J)

Example

stmt → id
stmt → var := expr
var → id
expr → var

$A \rightarrow \alpha \cdot B\ \beta,\ a$

stmt → • id, ;
stmt → • var := expr, ;
var → • id, := (*since First(:=expr;)*)

$B \rightarrow \cdot\ \gamma,\ b$
$b = First(\beta a)$

id

stmt → id •, {;}
var → id •, {:=}

LR(1) items avoid the R-R conflict
Now the two reduction items have different look-ahead tokens

LR(1) Parsing Table

1. Construct C = {I0, I1, … ,In}, the collection of sets of LR(1) items.
2. State i is constructed from Ii, the parsing actions are as follows
   a) If [$A \rightarrow \alpha \bullet a \beta$, b] is in Ii, and goto(Ii,*a*) = Ij, then set action[i,*a*] to "shift j", *a* is a terminal.
   b) If [$A \rightarrow \alpha \bullet$, *a*] is in Ii, set action[i,*a*] to "reduce $A \rightarrow \alpha$ "
   c) If [S'→S, $] is in Ii, set action[i,$] to accept.
3. If goto(Ii,A) = Ij, then goto[i,A] = j

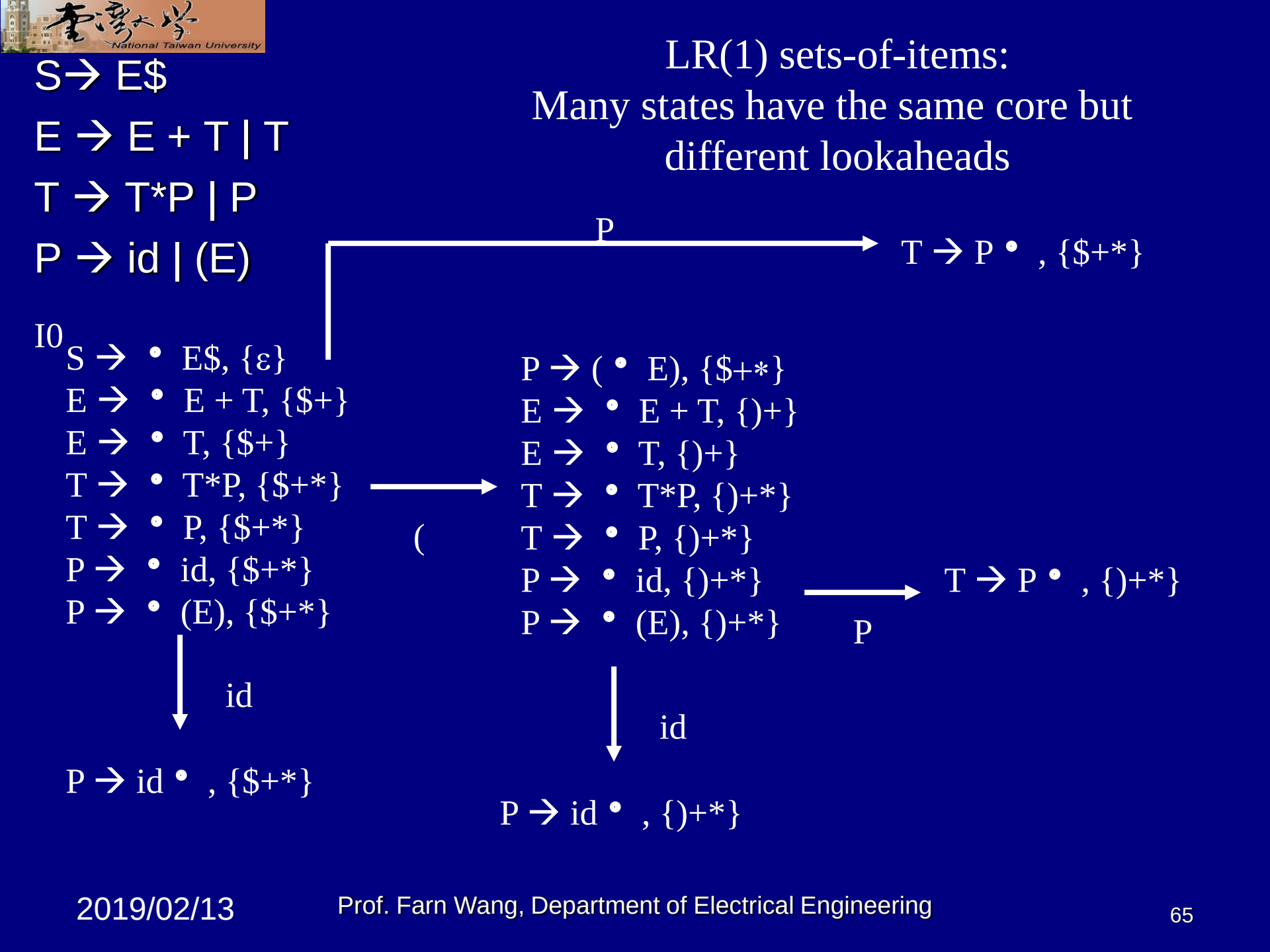
S→ E$

E → E + T | T

T → T*P | P

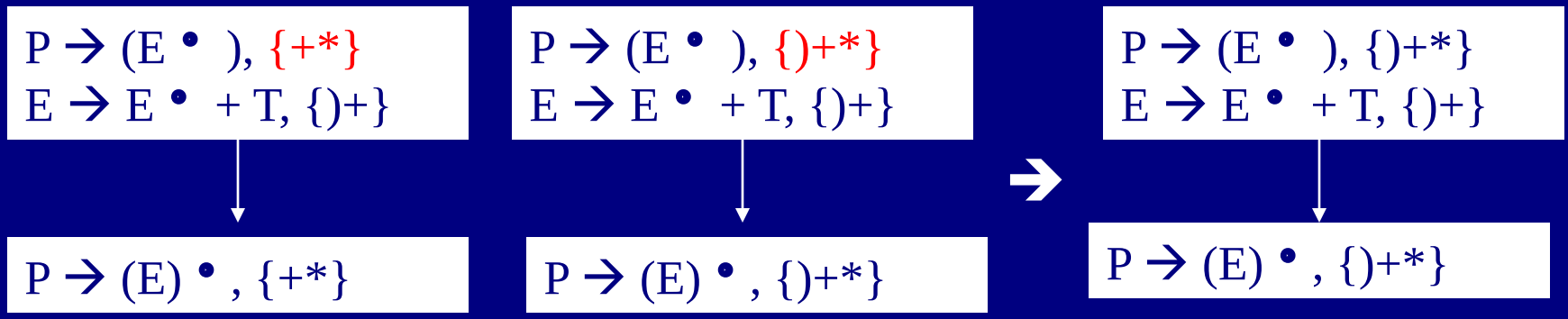
P → id | (E)

LR(1) sets-of-items:
Many states have the same core but different lookaheads



LALR(1)

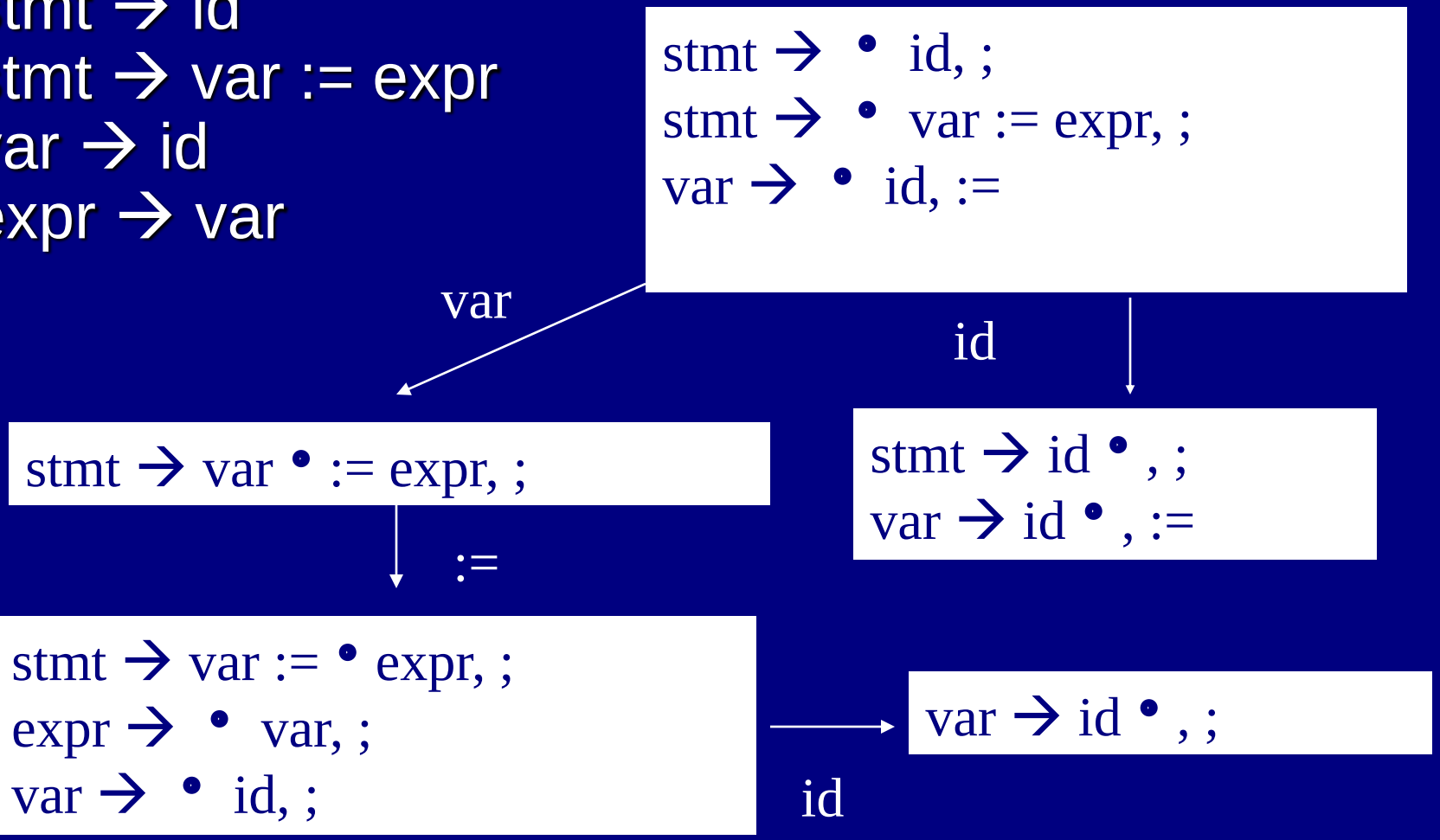
- LR(1) is more powerful, but it generates too many states and very large parse tables.
- In most cases, several states can be merged without losing parsing power.



- LALR(1) parser is an LR(1) parser in which all states that differ only in the look-ahead are merged.

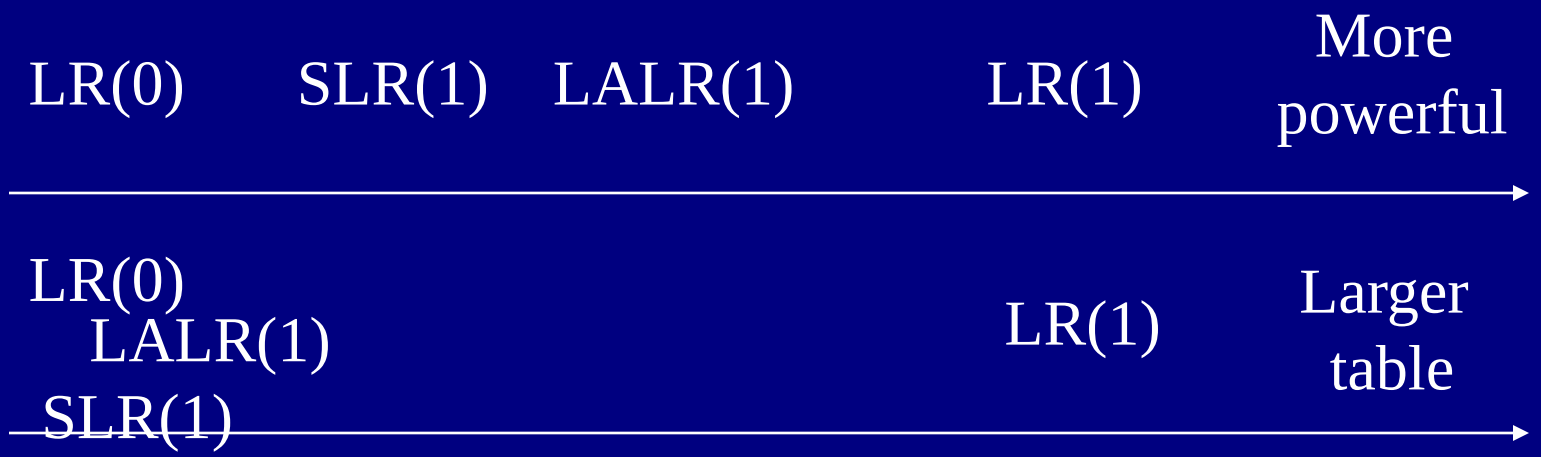
LALR handles previous SLR example

stmt → id
stmt → var := expr
var → id
expr → var

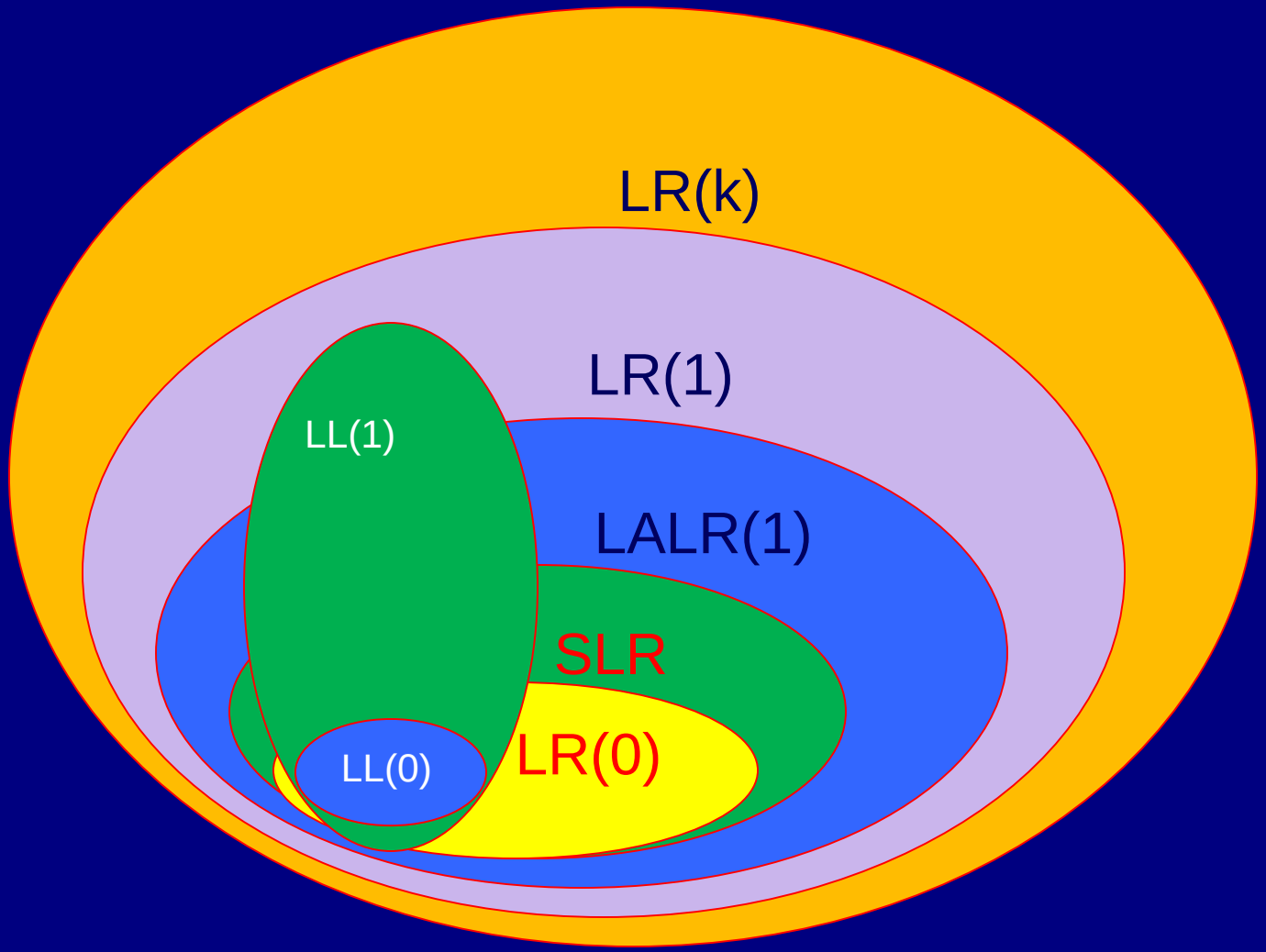


SLR and LALR

- LR(0): most compact parsing table. *Why?*
- LR(1): most powerful, but large table. *Why?*
- LR(0) plus look-ahead using FOLLOW() ➔ SLR
- LR(1) with state merge ➔ LALR



A Hierarchy of Grammar Classes



Compacting LR Parsing Table

- Table compaction techniques introduced in Ch. 3 for DFA can also be applied to parsing tables.
  - Identical actions can be shared (such states may not be merged because they may have different GoTo entries)
  - Sparse table techniques: use a list of pairs (symbol, action).

Error Recovery in LR Parsing

- Errors always detected by action table
- LR(1) parsing will never make a reduction before announcing an error.
- SLR and LALR may make several reductions before announcing an error, but they never shift an error token.
- Panic mode recovery:
  - Scan down the stack to locate a particular non-terminal A
  - Discard input symbols until a symbol can follow A
    - This symbol is usually called sync token such as “;”, “)”
  - Push state goto[s,A] and resume parsing

Midterm Exam (10/26)

- In-class exam
- 150 minutes
- Chapter 1 – 6
  - Introduction
  - AcDc compiler
  - Scanner
  - Lex
  - Top-down Parsing
  - Bottle-up parsing
  - Table driven parsing
- Open Book
- Open Notes
- Laptop / Pad are allowed, but no communications allowed.
- You may access to any public information, including YACC/BISON/Lex/Flex, tools, but not get help from classmates