Compiler Construction
Lent Term 2021
Lecture 6: Deterministic SLR(1) and LR(1) parsing

1. SLR(1) parsing
2. LR(1) parsing.

Timothy G. Griffin
tgg22@cam.ac.uk
Computer Laboratory
University of Cambridge

Our goal: impose deterministic choices on this non-deterministic LR parsing algorithm

c := first symbol of input w$

while(true)

$\alpha :=$ the stack

if $A \rightarrow \beta \bullet c\gamma \in \delta_G(q_0, \alpha)$

then shift c onto the stack

c := next input token;

if $\mathrm{A} \rightarrow \beta \bullet \in \delta_G(q_0, \alpha)$

then reduce : pop β off the stack

and then push A onto the stack;

if $\mathrm{S} \rightarrow \beta \bullet \in \delta_G(q_0, \alpha)$

then accept and exit if no more input;

if none of the above then ERROR

This is non-deterministic since multiple conditions can be true and multiple items can match any condition.

The easy part: NFA → DFA

In general, add new production $S' \to S$, where S is the original start symbol. For the simple term grammar G_2, add production

$$E' \to E$$

which produces the NFA start state

$$q_0 = E' \to \bullet E$$

The DFA start state is then

$$\varepsilon - \text{closure}(\{E' \to \bullet E\}) = \boxed{\begin{array}{l} E' \to \bullet E \\ E \to \bullet E + T \\ E \to \bullet T \\ T \to \bullet T * F \\ T \to \bullet F \\ F \to \bullet (E) \\ F \to \bullet \text{id} \end{array}}$$

The DFA transition function δ

For this DFA

$$\delta(\mathrm{I}, \mathrm{X}) = \varepsilon\text{-closure}(\{\mathrm{A} \rightarrow \alpha \mathrm{X} \bullet \beta \mid \mathrm{A} \rightarrow \alpha \bullet \mathrm{X} \beta \in I\})$$

Many books calls this GOTO(I, X).

and repeat the construction of DFA

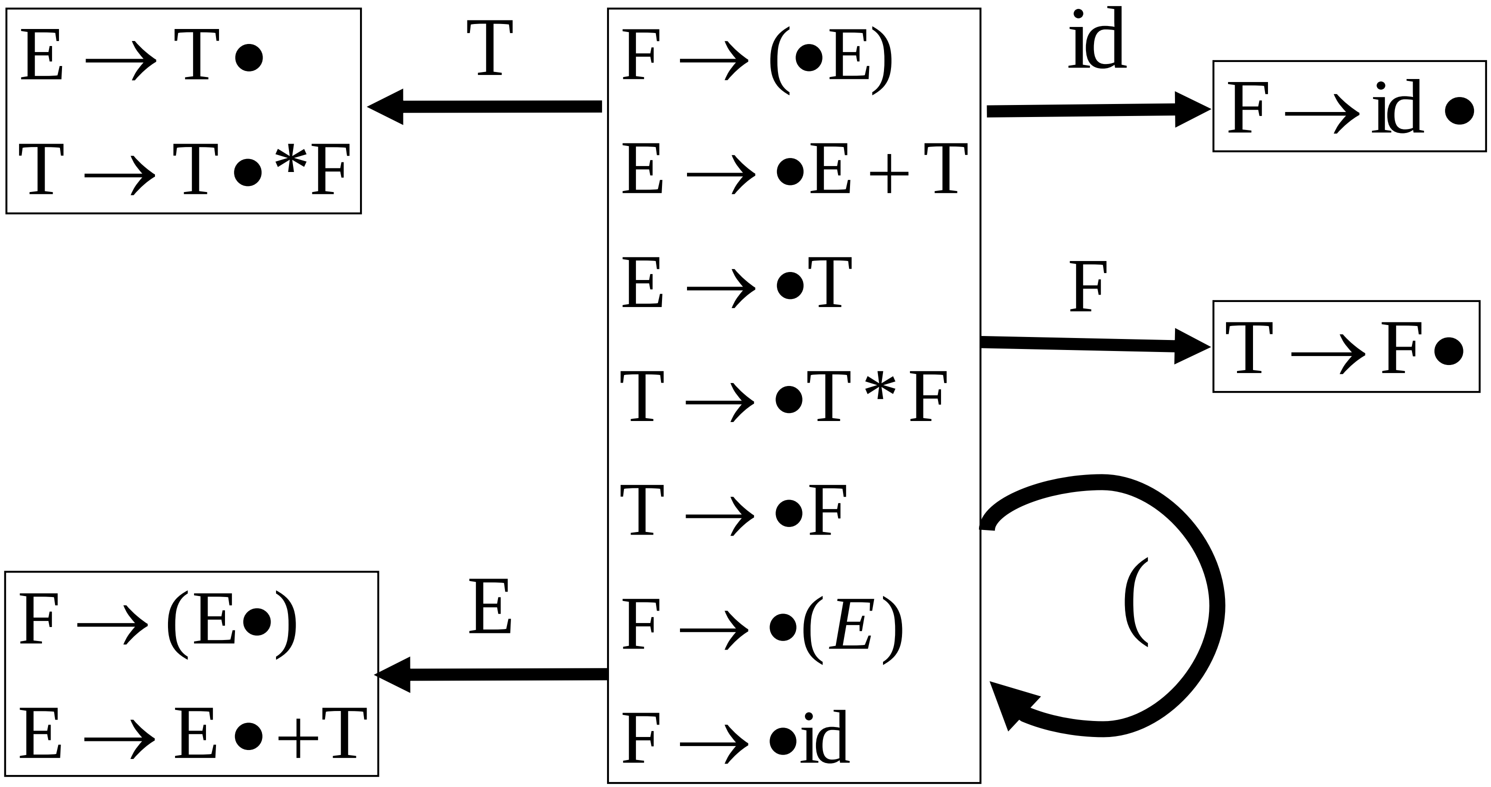
specialised to LR(0) items (using

function called CLOSURE). I see no reason to do

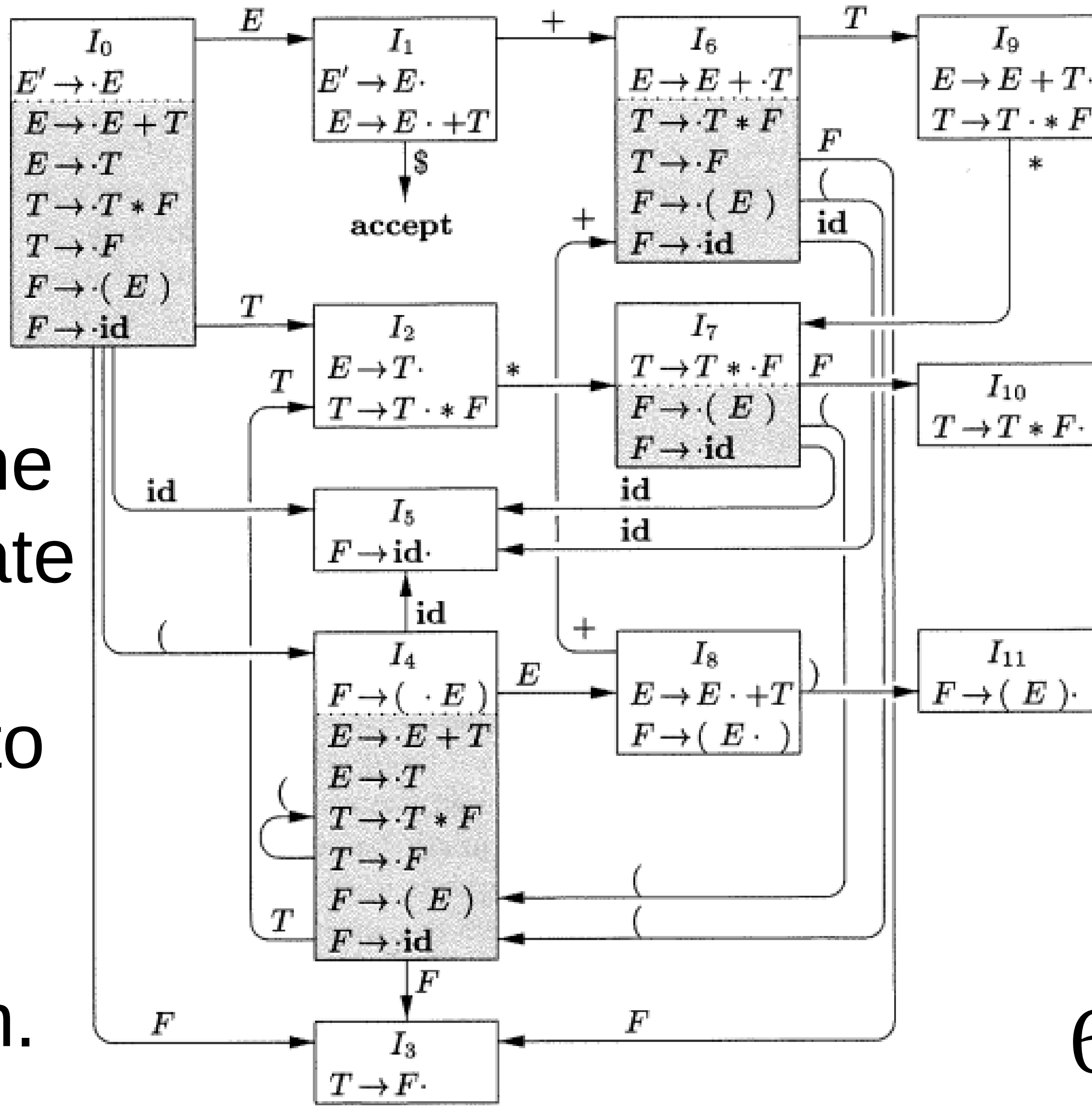
this since we already know how to build a DFA

from an NFA (see Lexing lecture).

A few DFA transitions for grammar G_2



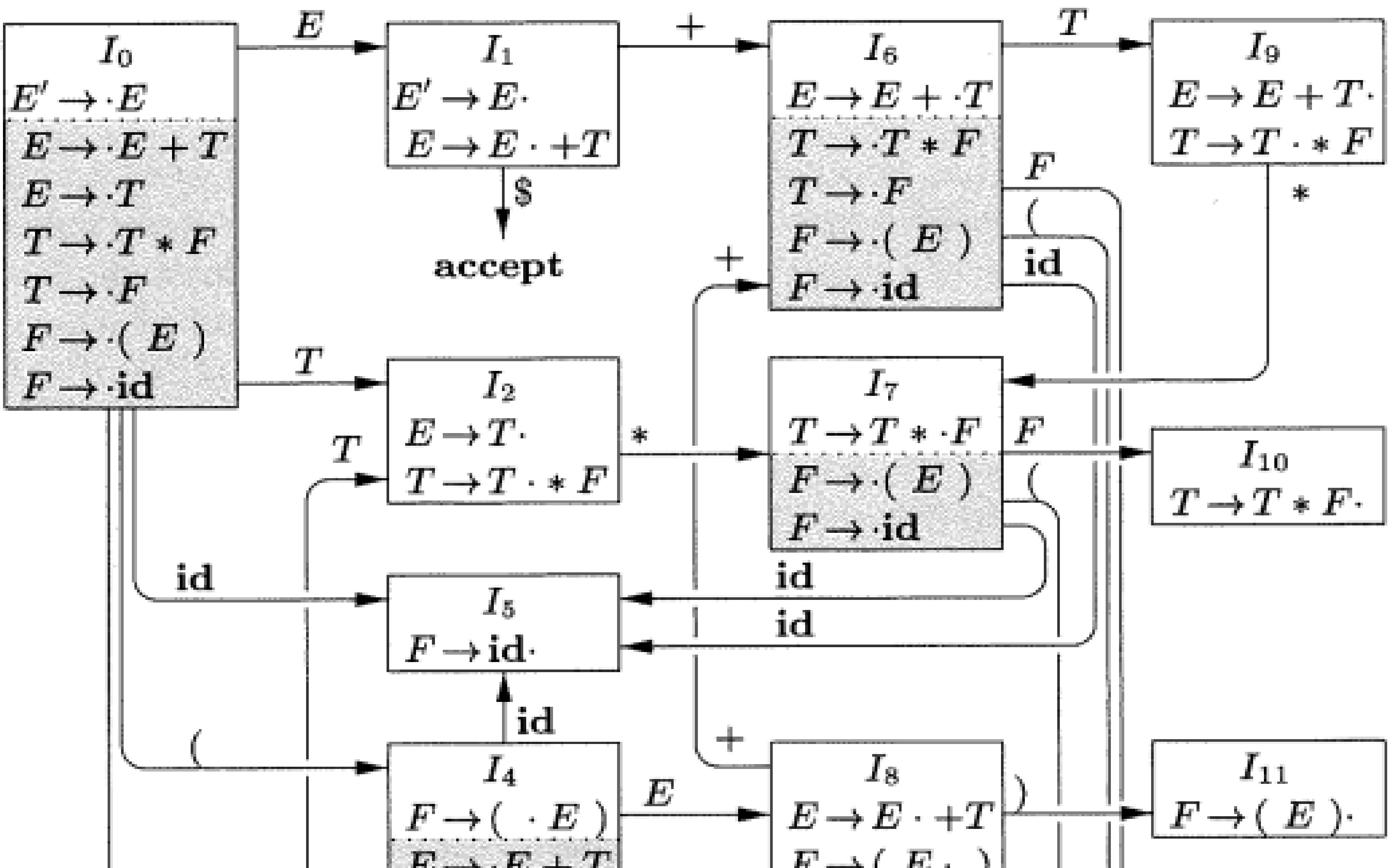
Full DFA for the stack language of G_2



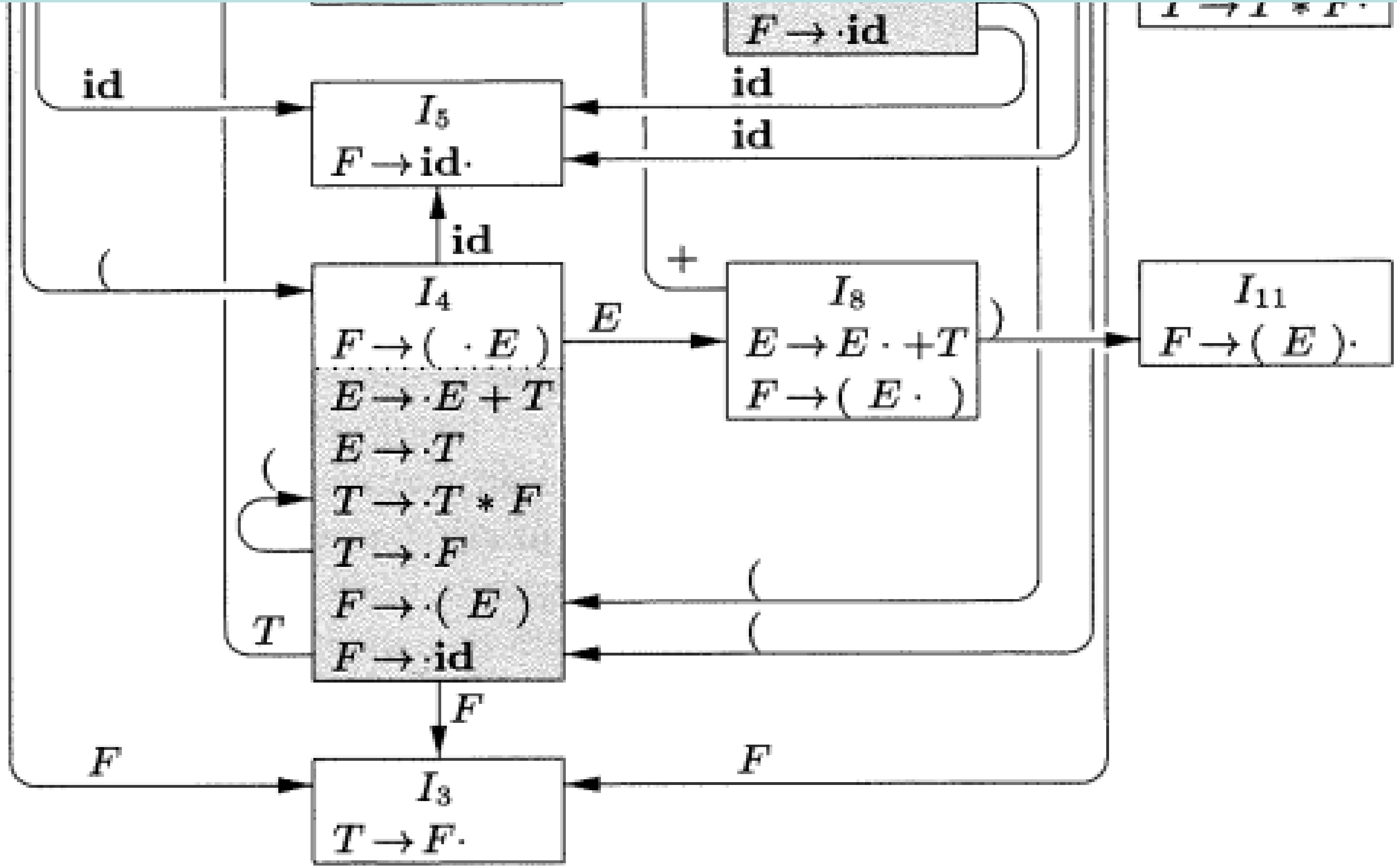
As usual, the ERROR state and transitions to it are not included in the diagram.

From Compilers by Aho, Lam, Sethi, Ullman

(enlarged to improve readability)



(enlarged to improve readability)



How can we avoid shift/reduce conflicts?

Consider I_2

$$\boxed{\begin{array}{l} \quad\quad I_2 \\ E \to T \bullet \\ T \to T \bullet *F \end{array}}$$

This inspires one approach called SLR(1) (Simple LR(1)):

1) Shift using if $*$ is the next token.

2) Reduce with $E \to T$ only if next token is in $\text{FOLLOW}(E) = \{(, +, \$\}$.

Now we can do a DETERMINISTIC SLR(1) parse of *(x+y)*

1) When the stack contains α, the parser is in state $\delta(\mathrm{I}_0, \alpha)$. For example,

$$\delta(\mathrm{I}_0, \mathrm{E}+\mathrm{T}) = \mathrm{I}_9$$

$$\delta(\mathrm{I}_0, (\mathrm{T}*\,) = \mathrm{I}_7$$

$$\delta(\mathrm{I}_0, \mathrm{E}*\mathrm{T}) = \mathrm{ERROR}$$

2) When the current state is I, the next token is c, and $\mathrm{A} \rightarrow \beta \bullet c\gamma \in \mathrm{I}$, then shift t onto stack

3) When the current state is I, the next token is c, $\mathrm{A} \rightarrow \beta\bullet \in \mathrm{I}$, and $\mathrm{c} \in \mathrm{FOLLOW(A)}$, then reduce with production $\mathrm{A} \rightarrow \beta$

Replay parsing of *(x+y)* using SLR(1) actions (FW(X) abbreviates FOLLOW(X))

stack, input	State action	reason
$\$, \quad (x+y)\$$	I_0 shift	$F \to \bullet(E) \in I_0$
$\$(, \quad x+y)\$$	I_4 shift	$F \to \bullet id \in I_4$
$\$(x, \quad +y)\$$	I_5 reduce $F \to id$	$"+" \in FW(F)$
$\$(F, \quad +y)\$$	I_3 reduce $T \to F$	$"+" \in FW(T)$
$\$(T, \quad +y)\$$	I_2 reduce $E \to T$	$"+" \in FW(E)$
$\$(E, \quad +y)\$$	I_8 shift	$E \to E \bullet +T \in I_8$
$\$(E+, \quad y)\$$	I_6 shift	$F \to \bullet id \in I_6$

stack,	input	State	action	reason
$\$(E+y,$	$)\$$	I_5	reduce $F \to id$	$")" \in FW(F)$
$\$(E+F,$	$)\$$	I_3	reduce $T \to F$	$")" \in FW(T)$
$\$(E+T,$	$)\$$	I_9	reduce $E \to E+T$	$")" \in FW(E)$
$\$(E,$	$)\$$	I_8	shift	$E \to (E\bullet) \in I_8$
$\$(E),$	$\$$	I_{11}	reduce $F \to (E)$	$"\$" \in FW(F)$
$\$F,$	$\$$	I_3	reduce $T \to F$	$"\$" \in FW(T)$
$\$T,$	$\$$	I_2	reduce $F \to E$	$"\$" \in FW(F)$
$\$E,$	$\$$	I_1	reduce $E' \to E$	$"\$" \in FW(E')$
$\$E',$	$\$$		accept!	

Better idea: Replace the stack contents with state numbers!

$($
$(id$
$(F$
$(T$
$(E$
$(E+$

0
04
045
043
042
048
0486

$(E+id$
$(E+F$
$(E+T$
$(E$
(E)
F
T
E

04865
04863
04869
048
04 11
03
02
01

LR parsing with DFA states on the stack

$a := \text{first symbol of input } w\$$

while(true)

$\quad s := \text{state at top of stack}$

$\quad \text{if ACTION}[s, a] = \text{shift } t$

$\quad \text{then push } t \text{ on stack}$

$\qquad a := \text{next input token}$

$\quad \text{else if ACTION}[s, a] = \text{reduce } A \to \beta$

$\qquad \text{then pop } |\beta| \text{ states off the stack}$

$\qquad\quad t := \text{state at top of stack}$

$\qquad\quad \text{push GOTO}[t, A] \text{ onto the stack}$

$\quad \text{else if ACTION}[s, a] = \text{accept}$

$\qquad \text{then accept and exit}$

$\quad \text{else ERROR}$

ACTION and GOTO for SLR(1)

If $[\mathrm{A} \to \alpha \bullet a\beta] \in \mathrm{I_i}$ and $\delta(\mathrm{I_i}, a) = \mathrm{I_j}$ then $\mathrm{ACTION[i,a]} = \text{shift j}$

If $[\mathrm{A} \to \alpha \bullet] \in \mathrm{I_i}$ and $\mathrm{A} \neq \mathrm{S'}$

then for all $a \in \mathrm{FOLLOW(A)}$,

$$\mathrm{ACTION[i,a]} = \text{reduce } \mathrm{A} \to \alpha$$

Note: there may still be shift/reduce or reduce/reduce conflicts!

If $[\mathrm{S'} \to \mathrm{S} \bullet] \in \mathrm{I_i}$ then $\mathrm{ACTION[i,\$]} = \text{accept}$

If $\delta(\mathrm{I_i}, \mathrm{A}) = \mathrm{I_j}$ then $\mathrm{GOTO[i, A]} = \mathrm{j}$

(Now do you see why I prefer to use δ rather than GOTO()?)

ACTION and GOTO for SLR(1)

STATE	ACTION						GOTO		
	id	+	*	(	)	$	*E*	*T*	*F*
0	s5			s4			1	2	3
1		s6				acc			
2		r2	s7		r2	r2			
3		r4	r4		r4	r4			
4	s5			s4			8	2	3
5		r6	r6		r6	r6			
6	s5			s4				9	3
7	s5			s4					10
8		s6			s11				
9		r1	s7		r1	r1			
10		r3	r3		r3	r3			
11		r5	r5		r5	r5			

From Compilers by Aho, Lam, Sethi, Ullman

Example parse

	STACK	SYMBOLS	INPUT	ACTION
(1)	0		$\mathbf{id} * \mathbf{id} + \mathbf{id}\,\$$	shift
(2)	0 5	$\mathbf{id}$	$* \mathbf{id} + \mathbf{id}\,\$$	reduce by $F \to \mathbf{id}$
(3)	0 3	F	$* \mathbf{id} + \mathbf{id}\,\$$	reduce by $T \to F$
(4)	0 2	T	$* \mathbf{id} + \mathbf{id}\,\$$	shift
(5)	0 2 7	$T *$	$\mathbf{id} + \mathbf{id}\,\$$	shift
(6)	0 2 7 5	$T * \mathbf{id}$	$+ \mathbf{id}\,\$$	reduce by $F \to \mathbf{id}$
(7)	0 2 7 10	$T * F$	$+ \mathbf{id}\,\$$	reduce by $T \to T * F$
(8)	0 2	T	$+ \mathbf{id}\,\$$	reduce by $E \to T$
(9)	0 1	E	$+ \mathbf{id}\,\$$	shift
(10)	0 1 6	$E +$	$\mathbf{id}\,\$$	shift
(11)	0 1 6 5	$E + \mathbf{id}$	$\$$	reduce by $F \to \mathbf{id}$
(12)	0 1 6 3	$E + F$	$\$$	reduce by $T \to F$
(13)	0 1 6 9	$E + T$	$\$$	reduce by $E \to E + T$
(14)	0 1	E	$\$$	accept

From Compilers by Aho, Lam, Sethi, Ullman

Beyond SLR(1)?

$$G_3 = (N_3, T_3, P_3, S')$$

$$N_3 = \{\mathrm{S'}, \mathrm{S}, \mathrm{L}, \mathrm{R}\}$$

$$T_3 = \{*, =, \mathrm{id}\}$$

$$P_3 : \mathrm{S'} \rightarrow \mathrm{S\$}$$

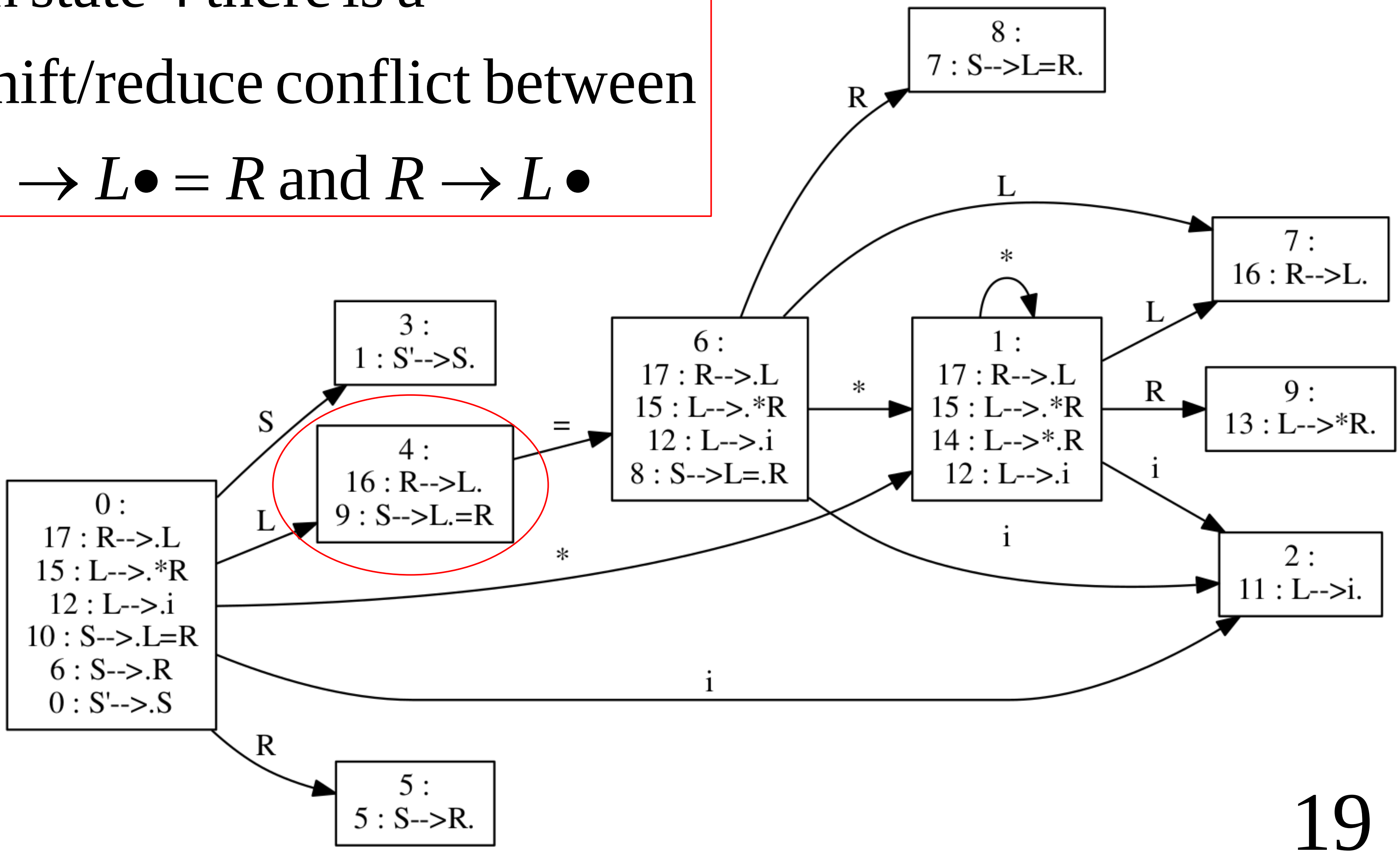
$$\mathrm{S} \rightarrow \mathrm{L} = \mathrm{R} \mid \mathrm{R}$$

$$\mathrm{L} \rightarrow *\mathrm{R} \mid \mathrm{id}$$

$$\mathrm{R} \rightarrow \mathrm{L}$$

LR(0) DFA for grammar G_3

In state 4 there is a shift/reduce conflict between $S \to L\bullet = R$ and $R \to L\,\bullet$



SLR(1) cannot resolve this conflict.

$$[\mathrm{S} \rightarrow L\bullet = R] \in \mathrm{I}_4 \text{ so } \delta(\mathrm{I}_4, \text{"="}) = \mathrm{I}_6$$

$$\text{and so ACTION[4,"="]} = \text{shift 6}$$

$$\text{However, } [\mathrm{R} \rightarrow L\bullet] \in \mathrm{I}_4$$

$$\text{and "="} \in \mathrm{FOLLOW(R)} = \{\text{"="}, \$\},$$

$$\text{so ACTION[4,"="]} = \text{reduce R} \rightarrow L$$

Beyond SLR(1)? LR(1)!

Problems: with SLR(1) there may be shift-reduce or reduce-reduce conflicts when ACTION and GOTO are not uniquely defined.

Either fix the grammar or use a more powerful technique.

LR(1) parsing starts with items of the form

$$[A \to \alpha \bullet \beta, \text{a}]$$

where a is an explicit look-ahead token.

Define an NFA with $LR(1)$ items as states

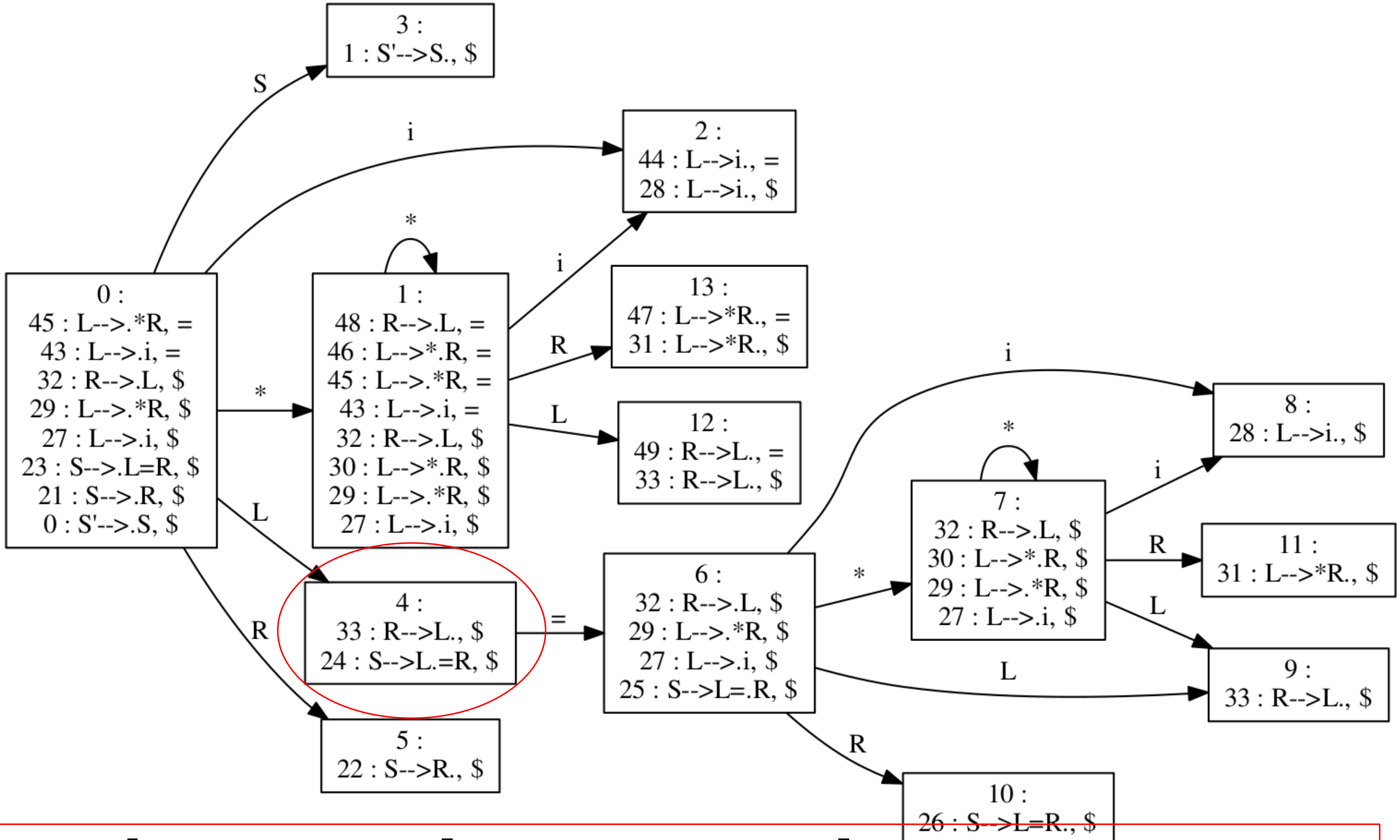
$$\boxed{\mathrm{A} \to \alpha \bullet c\beta,\ a} \xrightarrow{c} \boxed{\mathrm{A} \to \alpha c \bullet \beta,\ a}$$

$$\boxed{\mathrm{A} \to \alpha \bullet B\beta,\ a} \xrightarrow{B} \boxed{\mathrm{A} \to \alpha B \bullet \beta,\ a}$$

For each $b \in \mathrm{FIRST}(\beta a)$:

$$\boxed{\mathrm{A} \to \alpha \bullet B\beta,\ a} \xrightarrow{\varepsilon} \boxed{\mathrm{B} \to \bullet\gamma,\ b}$$

LR(1) DFA for grammar G_3



No ambiguity. Reduce $R \to L$ only if next token is \$. Otherwise shift if next token is $=$.

ACTION and GOTO for LR(1)

If $[A \rightarrow \alpha \bullet a\beta, a] \in I_i$ and $\delta(I_i, a) = I_j$ then $\text{ACTION}[i, a] = \text{shift } j$

If $[A \rightarrow \alpha \bullet, b] \in I_i$ and $A \neq S'$, then

$$\text{ACTION}[i, b] = \text{reduce } A \rightarrow \alpha$$

If $[S' \rightarrow S \bullet, \$] \in I_i$ then $\text{ACTION}[i, \$] = \text{accept}$

If $\delta(I_i, A) = I_j$ then $\text{GOTO}[i, A] = j$

SLR(1) vs LR(1)

$$\begin{aligned}&\mathrm{SLR(1):}\\&\text{If } [\mathrm{A} \rightarrow \alpha \bullet] \in \mathrm{I_i} \text{ and } \mathrm{A} \neq \mathrm{S'}\\&\text{then for all } a \in \mathrm{FOLLOW(A)},\\&\qquad\qquad \mathrm{ACTION[i,a]} = \text{reduce } \mathrm{A} \rightarrow \alpha\end{aligned}$$

$$\begin{aligned}&\mathrm{LR(1):}\\&\text{If } [\mathrm{A} \rightarrow \alpha \bullet, b] \in \mathrm{I_i} \text{ and } \mathrm{A} \neq \mathrm{S'}\text{, then}\\&\qquad\qquad \mathrm{ACTION[i,b]} = \text{reduce } \mathrm{A} \rightarrow \alpha\end{aligned}$$

Note that the look-ahead symbol b is used ONLY for reductions, not for shifts.

SLR(1) vs LR(1)

1. LR(1) is more powerful than SLR(1)
2. The DFA associated with a LR(1) parser may have a very large number of states
3. This inspired an optimisation (collapsing states) resulting in a the class of LALR papers normally implemented as YACC. These parsers have fewer states but can produce very strange error messages.
4. Ocaml’s Menhir is based on LR(1) and claims to overcome many YACC problems.
5. We will not cover LALR parsing.