Constructing SLR states

- LR(0) state machine
  - encodes all strings that are valid on the stack
  - each valid string is a configuration, and hence corresponds to a state of the LR(0) state machine
  - each state tells us what to do (shift or reduce?)

Constructing SLR states

- How to find the set of needed configurations
  - What are the valid handles that can appear at the front of the input?
  - Begin with item $S' \rightarrow . Start$, calculate related items (closure)
  - Determine following states (what states can be reached on a single input or nonterminal)
  - Construct closure of each resulting state

Closure of a Set of Items

closure (items *i*, grammar *g*)

c = *i*

repeat

for each item $X \to \alpha\ .\ Y\ \beta$ in *c* and

each production $Y \to \gamma$ from *g* s.t.

$Y \to .\ \gamma$ is not in *c*

add $Y \to .\ \gamma$ to *c*

until no further changes to *c*

return *c*

Closure Example

- Grammar

  E → T + E | T

  T → int * T | int | (E)
- Initial item S′ → . E
- *c* = { S′ → . E }
  - First pass { S→.E }
    - Add E → . T+E   *c* = { S′→.E,E→.T+E }
    - Add E → . T   *c* = { S′→.E,E→.T+E,E→.T }
  - Second pass *c* = { S′→.E,E→.T+E,E→.T }
    - Add T → . int * T *c* = { S′→.E,E→.T+E,E→.T,T→.int*T }
    - Add T → . int   *c* = { S′→.E,E→.T+E,E→.T,T→.int*T,T→.int }
    - Add T → . (E)   *c* = { S′→.E,E→.T+E,E→.T,T→.int*T,T→.int ,T→.(E) }
  - Third pass *c* = { S′→.E,E→.T+E,E→.T,T→.int*T,T→.int ,T→.(E) }
    - no change

Closure Example

- Closure results in a new state
- Closure of { S′ → . E } is

  { S′→.E,E→.T+E,E→.T,T→.int*T,
  T→.int ,T→.(E) }

S′ → . E	1
E → . T	
E → .T + E	
T → .(E)	
T → .int * T	
T → .int	

New States – the goto Function

- To determine possible states reachable from existing state use goto function
- ***goto***(*state,stack element*) is the closure of the set of items that result from shifting *stack element* in *state*
- For state { S′→.E,E→.T+E,E→.T,T→.int*T,
  T→.int ,T→.(E) }

  set of items resulting from shifting *int* are:

  { T→int. , T→int . *T}

Goto Function

goto (items *i*, stackel *J*, grammar *g*)

initial= set of items $X \to \alpha\ J\ .\ \beta$

such that $X \to \alpha\ .\ J\ \beta$ is in *i*

return closure(*initial*, *g*)

Producing Set of States

calc_states (grammar *g*)

sts = { closure({[S'->. Start]},*g*) }

repeat

for each state *s* in *sts* and

each stack element *e*

such that goto(*s*,*e*) is not empty and

not in *sts*

add goto(*s*,*e*) to *sts*

until no more states can be added to *sts*

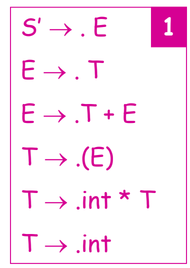
Defining SLR States Example

- Grammar

  E → T + E | T

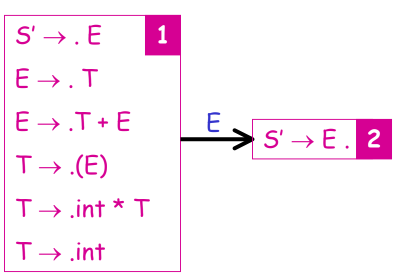
  T → int * T | int | (E)

- Initial state:

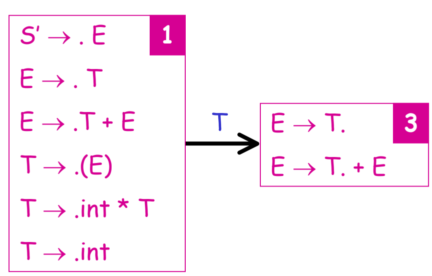


States Example (cont)

From initial state (1) on E we get state 2



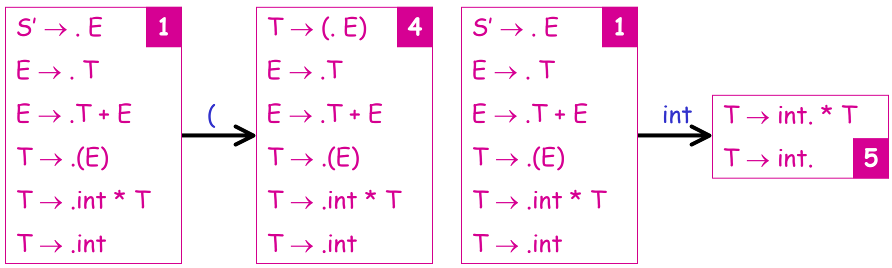
From state 1 on T we get state 3



States Example (cont)

From state 1 on (we get state 4

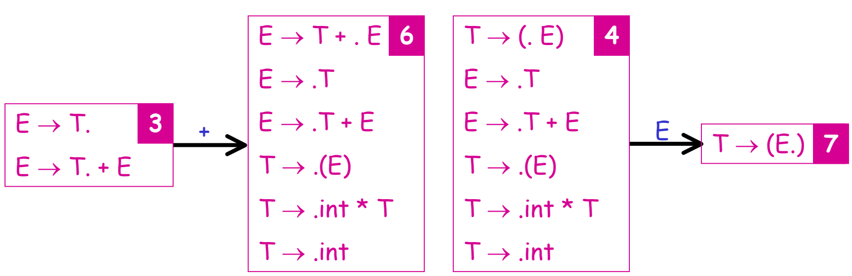
From state 1 on int we get state 5



States Example (cont)

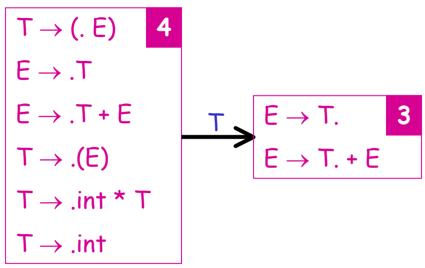
From state 3 on + we get state 6

From state 4 on E we get state 7

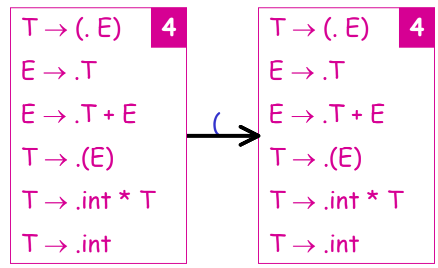


States Example (cont)

From state 4 on T we get state 3

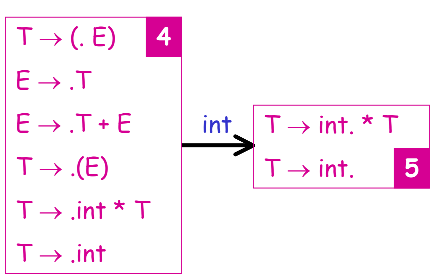


From state 4 on (we get state 4

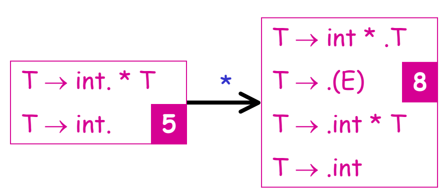


States Example (cont)

From state 4 on int we get state 5

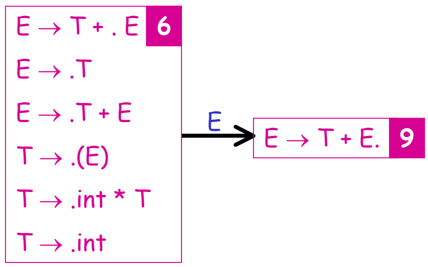


From state 5 on * we get state 8

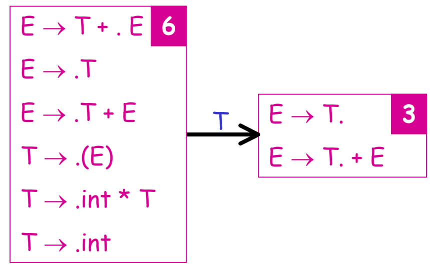


States Example (cont)

From state 6 on E we get state 9

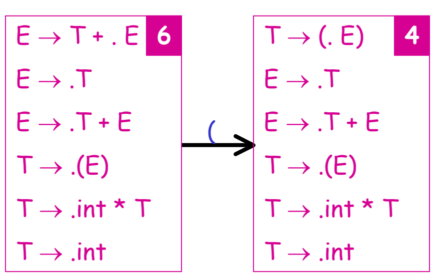


From state 6 on T we get state 3

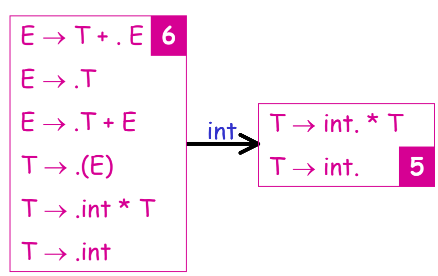


States Example (cont)

From state 6 on (we get state 4



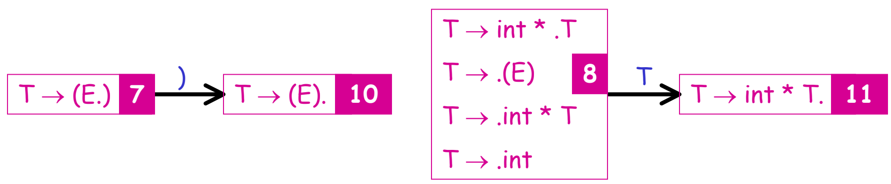
From state 6 on int we get state 5



States Example (cont)

From state 7 on) we get state 10

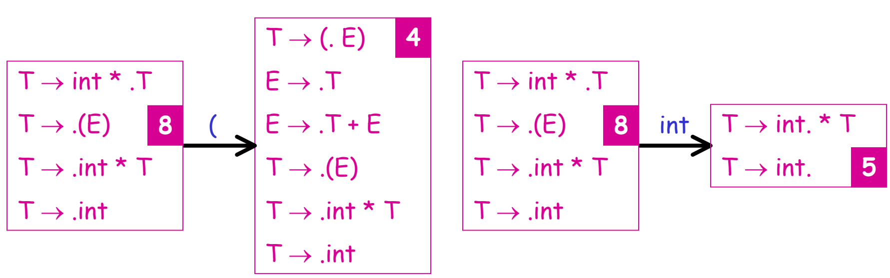
From state 8 on T we get state 11



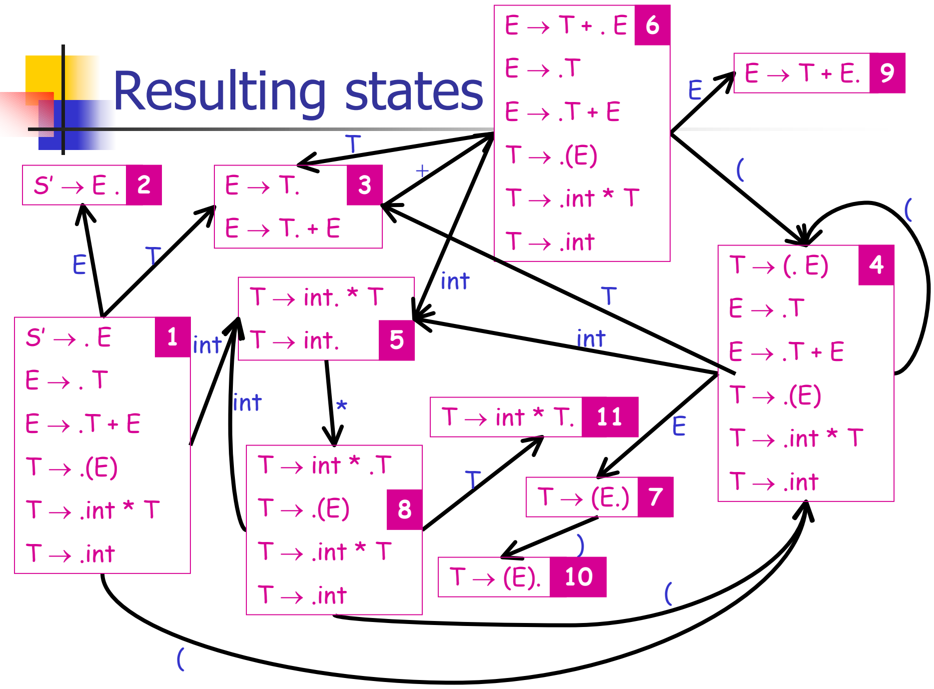
States Example (cont)

From state 8 on (we get state 4

From state 8 on int we get state 5



Resulting states



Constructing an SLR Parse Table

- Add an extra production S′ → Start to the grammar
- Construct set of LR(0) States, the initial state is the one containing S′ → . Start
- For each transition A $\rightarrow^{X}$ B in the set of states add an action shift B in column X for row A
- For each item [Y → α .] part of state A, set the action in row A to reduce Y → α for each column X in FOLLOW(Y)
- Empty table items are errors
- Grammar is not SLR if more than one entry for any table item

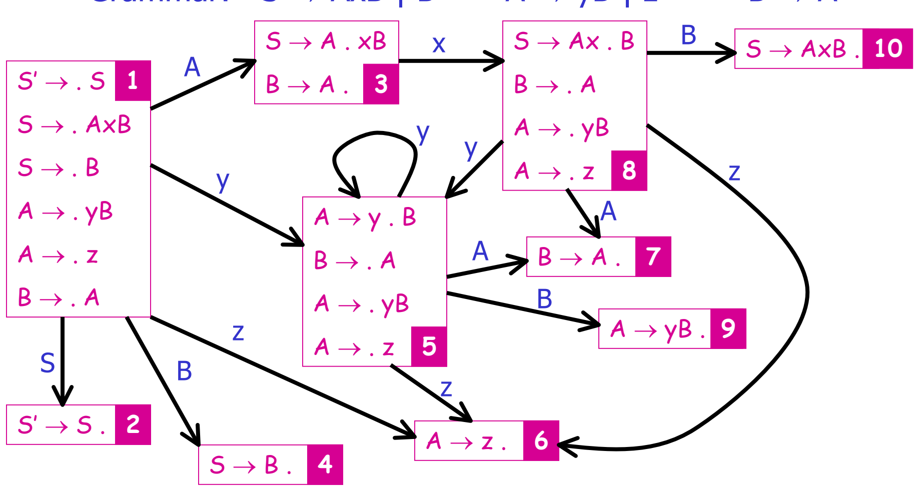
Resulting SLR Parse Table

	int	*	+	(	)	$	E	T
1	s5			s4			s2	s3
2						acc		
3			s6		r2	r2		
4	s5			s4			s7	s3
5		s8	r4		r4	r4		
6	s5			s4			s9	s3
7					s10			
8	s5			s4				s11
9					r1	r1		
10			r5		r5	r5		
11			r3		r3	r3		

1: $E \to T + E$
2: $E \to T$
3: $T \to int * T$
4: $T \to int$
5: $T \to (E)$

Another Example

Grammar: $S \to AxB \mid B$ $A \to yB \mid z$ $B \to A$



Corresponding Parse Table

	x	y	z	$	S	A	B
1		s5	s6		s2	s3	s4
2				acc			
3	s8, r5			r5			
4				r2			
5		s5	s6			s7	s9
6	r4			r4			
7	r5			r5			
8		s5	s6			s7	s10
9	r3			r3			
10				r1			

Follow(S) = {$}
Follow(A) = {x,$}
Follow(B) = {x,$}

1: $S \rightarrow AxB$
2: $S \rightarrow B$
3: $A \rightarrow yB$
4: $A \rightarrow z$
5: $B \rightarrow A$

Limits of SLR Parsing

- But is it really possible to get to state 3 through a B – no, the only viable prefix involves an A!
- So the reduce is a bad choice
- Limit introduced by SLR parsing in using the FOLLOW set to decide reductions
- Idea: augment LR items with 1 character lookahead [$S \rightarrow .\ AxB$, b] making an LR(1) item

Canonical LR Parsing

- States similar to SLR, but use LR(1) rather than LR(0) items
- When reduction is possible, use reduction of an item [S $\rightarrow \alpha$. , x] only when next token is x (lookahead items used only for reductions)
- Advantage: avoids some conflicts introduced by SLR parsing tables
- Disadvantage: table is often MUCH larger as items are differentiated by which character currently used for lookahead
- Building LR(1) tables – similar to SLR, only need change closure and goto functions

Look Ahead LR (LALR) Parsing

- Disadvantage of large tables can be mitigated by merging states
- States can be merged when there is no fundamental difference
  - E.g., similar states with no reductions possible with different lookahead characters