SLR (1) Parsers, LR (0) items

Bottom up Parsers

- Simple Shift-reduce parsers has lot of Shift/Reduce conflicts
- Operator precedence parsers is for a small class of grammars
- Go for LR parsers

LR Parsers

- LR(1) parsers recognize the languages in which one symbol of look-ahead is sufficient to decide whether to shift or reduce
  - L : for left-to-right scan of the input
  - R : for reverse rightmost derivation
  - 1: for one symbol of look-ahead

LR Parsers

- Read input, one token at a time
- Use stack to keep track of current state
  - The state at the top of the stack summarizes the information below.
  - The stack contains information about what has been parsed so far.

LR Parsers

- Use parsing table to determine action based on current state and look-ahead symbol.
- Parsing table construction takes into account the shift, reduce, accept or error action

LR Parsers

- SLR
  - Simple LR parsing
  - Easy to implement, but not powerful
  - Uses LR(0) items
- Canonical LR
  - Larger parser but powerful
  - Uses LR(1) items

- LALR
  - Condensed version of canonical LR
  - May introduce conflicts
  - Uses LR(1) items

SLR Parsers - Handle

- As a SLR parser processes the input, it must identify all possible handles.
- For example, consider the usual expression grammar and the input string

  a + b.

SLR Parsers

- If the parser has processed ‘a’ and reduced it to E. Then, the current state can be represented by E • +E where • means
  - E has already been parsed and
  - +E is a potential suffix, which, if determined, yields to a successful parse.

$E \rightarrow \bullet E + E$

$\underline{E} \bullet \underline{+E}$

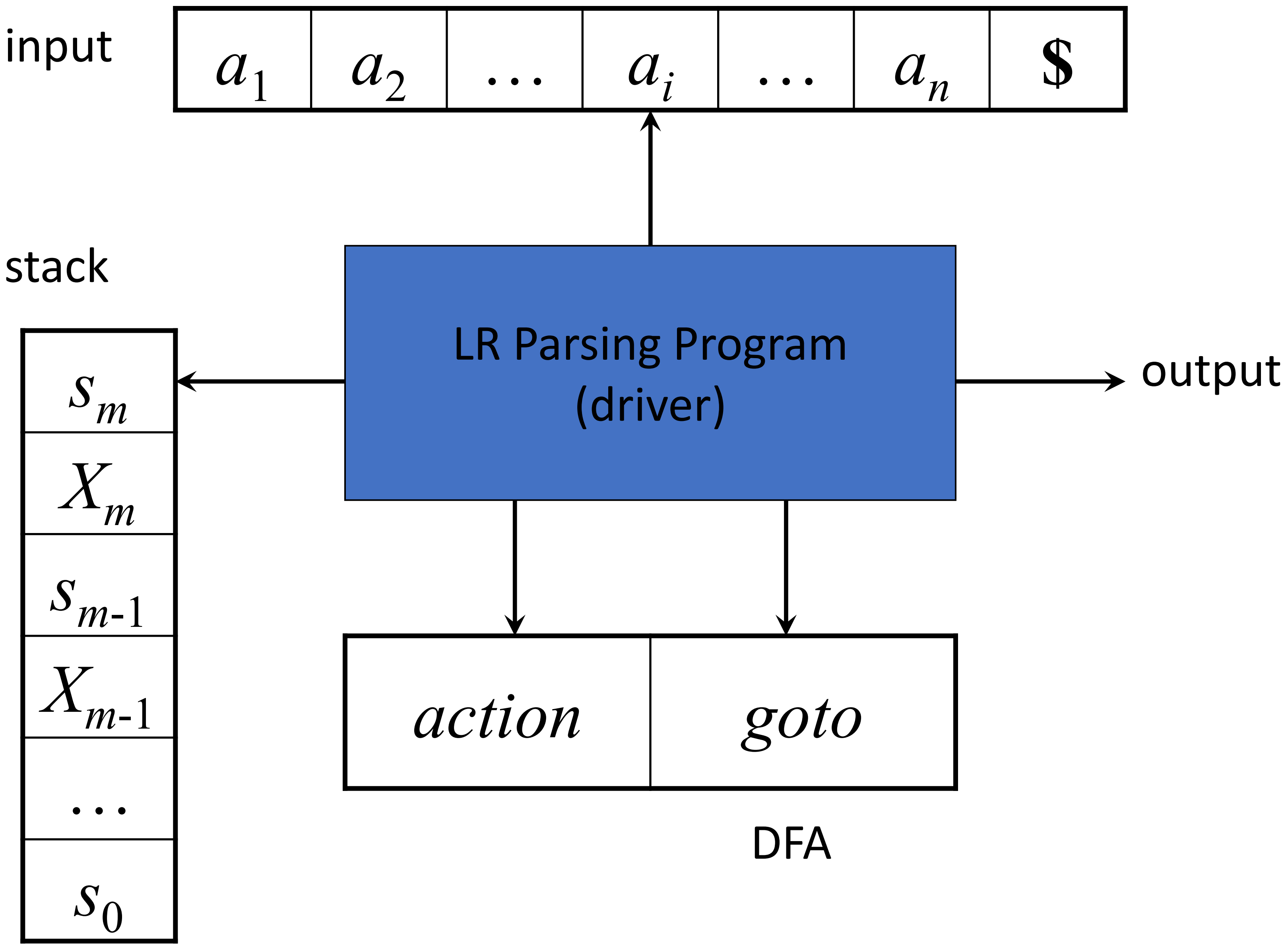
$E + E \bullet$

SLR parsers

- Our ultimate aim is to finally reach state E+E•, which corresponds to an actual handle yielding to the reduction $E \rightarrow E+E$

SLR Parsers

- LR parsing works by building an automata where each state represents what has been parsed so far and what we intend to parse after looking at the current input symbol. This is indicated by productions having a “.” These productions are referred to as items.
- Items that has the “.” at the end leads to the reduction by that production


input
a_1
a_2
…
a_i
…
a_n
$
stack
LR Parsing Program
(driver)
output
s_m
X_m
s_{m-1}
X_{m-1}
…
s_0
action
goto
DFA

SLR (1) Parser

- Form the augmented grammar
- Construction of LR(0) items
- Construct the follow() for all the non-terminals which requires construction of first() for all the terminals and non-terminals

$E' \to .E$

$E \to .E+T / T$

$T \to T*F \mid F$

$F \to (E) \mid id$

$\boxed{E' \to E.}$

SLR(1) parser

- Using this and the follow( ) of the grammar, construct the parsing table
- Using the parsing table, a stack and an input parse the input

LR (0) items ⇒ Itemsets

- An *LR(0) item* of a grammar *G* is a production of *G* with a • at some position of the right-hand side
- Thus, a production

  $A \rightarrow X\ Y\ Z$

  has four items:

  $[A \rightarrow \bullet\ X\ Y\ Z]$ ≠ ~~A → X Y • Z~~

  $[A \rightarrow X \bullet Y\ Z]$

  $[A \rightarrow X\ Y \bullet Z]$

  $[A \rightarrow X\ Y\ Z\ \bullet]$

- that production $A \rightarrow \varepsilon$ has one item $[A \rightarrow \bullet]$

$A \rightarrow X \bullet \boxed{Y} Z$

$\Downarrow$

$A \rightarrow X Y \bullet Z$

$\Downarrow$

LR (0) items

- The grammar is augmented with a new start symbol S' and production $S' \to S$
- Initially, set $C = closure(\{[S' \to \bullet S]\})$
- For each set of items $I \in C$ and each grammar symbol $X \in (N \cup T)$ such that $goto(I,X) \notin C$ and $goto(I,X) \neq \emptyset$,
  - add the set of items $goto(I,X)$ to C
- Repeat until no more sets can be added to C

Closure (I)

- Start with *closure*(I) = I
- If $[A \rightarrow \alpha \bullet B\beta] \in$ *closure*(I) then for each production $B \rightarrow \gamma$ in the grammar, add the item $[B \rightarrow \bullet\gamma]$ to I if not already in I
- Repeat 2 until no new items can be added

$B \rightarrow cD \mid dD$

$C \rightarrow \gamma$

$A \rightarrow a \cdot B b$

$B \rightarrow \cdot cD$

$B \rightarrow \cdot dD$

$C \rightarrow \cdot \gamma$

Goto (I, X)

- For each item $[A \rightarrow \alpha \bullet X\beta] \in I$, add the set of items *closure*($\{[A \rightarrow \alpha X \bullet \beta]\}$) to *goto*($I$,$X$) if not already there
- Repeat until no more items can be added to *goto*(I,X)
- Intuitively, *goto*(I,X) is the set of items that are valid for the viable prefix γX when I is the set of items that are valid for γ

Augmented Grammar

E' → E

1. E → E + T
2. E → T
3. T → T * F
4. T → F
5. F → (E)
6. F → id

Augmented Grammar

E' → E

1. E → E + T
2. E → T
3. T → T * F
4. T → F
5. F → (E)
6. F → id

I_0: E' → .E
E → .E+T
E → .T
T → .T*F
T → .F
F → .(E)
F → .id

I_1: Goto(I_0, E)
E' → E.
E → E.+T

I_2: Goto(I_0, T)
Goto(I_4, T)
E → T.
T → T.*F

I_3: Goto(I_0, F)
T → F.
Goto(I_4, F)
Goto(I_6, F)

I_5: Goto(I_0, id)
F → id.
Goto(I_4, id)
Goto(I_6, id)
Goto(I_7, id)

I_4: Goto(I_0, ( )
Goto(I_6, ( )
F → (.E)
Goto(I_4, ( )
Goto(I_7, ( )
E → .E+T
E → .T
T → .T*F
T → .F
F → .(E)
F → .id

I_6: Goto(I_1, +)
E → E+.T
Goto(I_8, +)
T → .T*F
T → .F
F → .(E)
F → .id

I_0

- E' → .E
- E → . E +T
- E → .T
- T → .T * F
- T → .F
- F → . (E)
- F → . id

I_7 : Goto(I_2, *)

T → T * . F

F → . (E)

F → . id

Goto(I_9, *)

I_8 : Goto(I_4, E)

F → (E.)

E → E . + T

I_9 : Goto(I_6, T)

E → E + T.

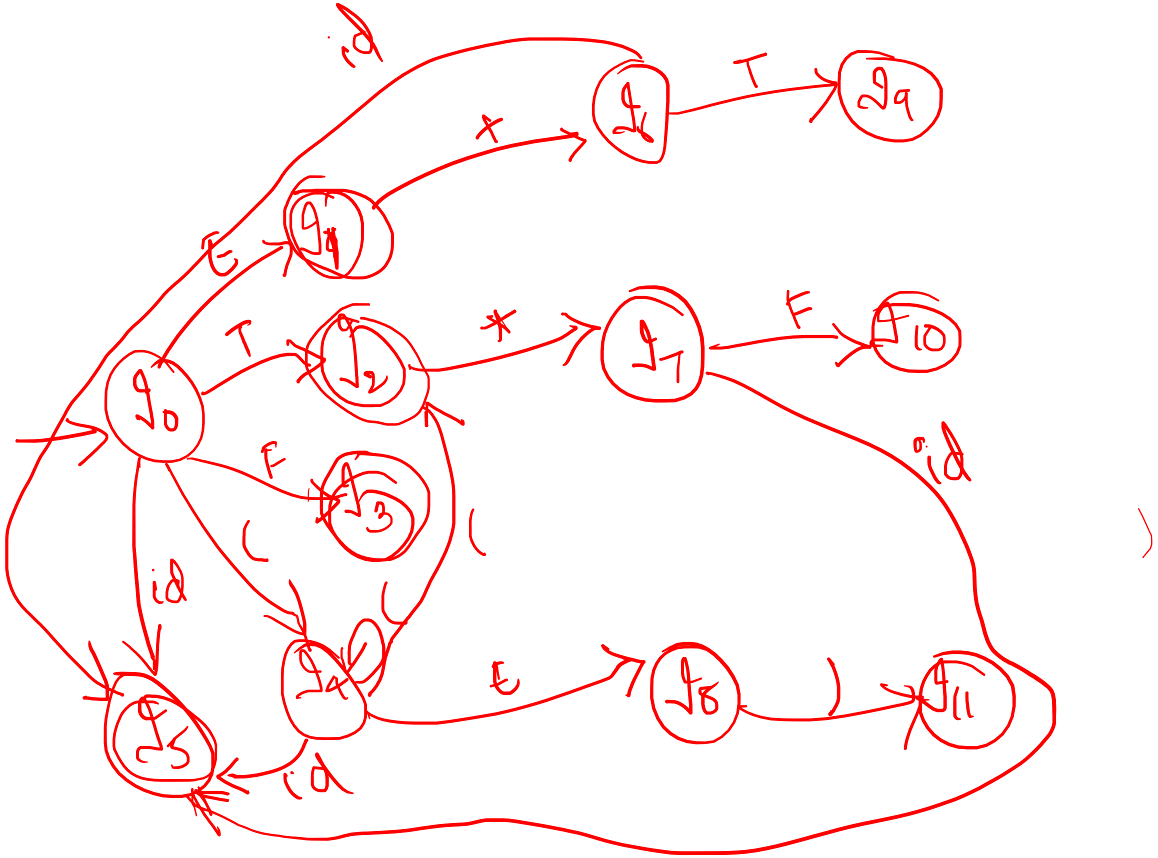
T → T. * F

I_{10} : Goto(I_7, F)

T → T * F.

I_{11} : Goto(I_8,))

F → (E).



q_0
q_1
q_2
q_3
q_4
q_5
q_6
q_7
q_8
q_9
q_{10}
q_{11}
E
T
F
id
(
+
*
)

I_0

- $E' \rightarrow .E$
- $E \rightarrow . E + T$
- $E \rightarrow .T$
- $T \rightarrow .T * F$
- $T \rightarrow .F$
- $F \rightarrow . (E)$
- $F \rightarrow . id$

Items

- I_1 = Goto (I_0 , E)
- E’ → E.
- E → E . + T

- I_2 = Goto (I_0 , T), Goto (I_3 , T),
  E → T.
  T → T.*F

I_3 = Goto (I_0 , ( ), Goto (I_3, (), Goto (I_6, () Goto (I_{10},()

- F → (.E)
- E → .E + T
- E → .T
- T → .T * F
- T → .F
- F → .(E)
- F → .id

- I_4 = Goto (I_0 , F), Goto (I_3 , F),

  Goto (I_6 , F)

- T → F.

- I_5= Goto (I_0 , id) Goto (I_3 , id) Goto (I_6, id)

  Goto (I_{10}, id)

  F → id .

Items

I_6= Goto (I_1 , +), Goto (I_7 , +),

E → E + . T

T→ .T * F

T → .F

F→ .(E)

F → .id

Items

I_7= Goto (I_3 , E)

F → (E.)

E → E . +T

I_8 = Goto(I_6 , T)

E → E + T.

T→ T. * F

I_9 : Goto (I_7,))

F $\rightarrow$ (E).

I_{10}: Goto (I_8, *), Goto (I_2, *)

T$\rightarrow$ T * . F

F $\rightarrow$.(E)

F $\rightarrow$ .id

I_{11} : Goto (I_{10}, F)

T $\rightarrow$ T * F.

SLR Parsing Table

- Input: Augmented Grammar G'
- Output: SLR parsing table with functions, shift, reduce and accept
- Parsing table is between items and Terminals and non-terminals
- The non-terminals correspond to the goto() of the items set
- The terminals have the parsing table corresponding to the action – shift / reduce/accept

SLR Parsing Table

- Augment the grammar with $S' \rightarrow S$
- Construct the set $C=\{I_0,I_1,\ldots,I_n\}$ of *LR(0) items*
- If $[A \rightarrow \alpha \bullet a\beta] \in I_i$ and $goto(I_i,a)=I_j$ then set $action[i,a]$=shift *j, where a is a terminal*
- If $[A \rightarrow \alpha \bullet] \in I_i$ then set $action[i,a]$=reduce $\mathrm{A} \rightarrow \alpha$ for all $a \in$ FOLLOW(A) where $A \neq S'$)

SLR parsing table

- If $[S' \rightarrow S\bullet]$ is in I_i then set *action*[i,**$**]=accept
- If *goto*(I_i,A)=I_j then set *goto*[i,A]=j
- Repeat for all the items until no more entries added
- The initial state i is the I_i holding item $[S' \rightarrow \bullet S]$
- All other entries are error

Grammar

- E' → E
1 • E → E +T
2 • E → T
3 • T → T * F
4 • T → F
5 • F → (E)
6 • F → id

I_0

- E’ → .E
- E → . E +T
- E → .T
- T → .T * F
- T → .F
- F → . (E)
- F → . id

- I_1 = Goto (I_0 , E)
  E’ → E.
  E → E . + T
- I_2 = Goto (I_0 , T), Goto (I_3 , T),
  E → T.
  T → T.*F
- I_4 = Goto (I_0 , F), Goto (I_3 , F),
  Goto (I_6 , F)
  T → F.
- I_5= Goto (I_0 , id) Goto (I_3 , id) Goto (I_6, id)
  Goto (I_{10}, id)
  F → id .

- I_3 = Goto (I_0 , ( ), Goto (I_3, (), Goto (I_6, ()
  Goto (I_{10},()
- F → (.E)
- E → .E + T
- E → .T
- T → .T * F
- T → .F
- F → .(E)
- F → .id

- I_6= Goto (I_1 , +), Goto (I_7 , +),

E → E + . T

T→ .T * F

T → .F

F→ .(E)

F → .id

- I_7= Goto (I_3 , E)

F → (E.)

E → E . +T

- I_8 = Goto(I_6 , T)

E → E + T.

T→ T. * F

- I_9 : Goto (I_7,))

F → (E).

- I_{10}: Goto (I_8, *), Goto (I_2, *)

T→ T * . F

F → .(E)

F → .id

- I_{11} : Goto (I_{10}, F)

T → T * F.

Follow

- FOLLOW(A) =

  if A is the start symbol S **then**

  add **$** to FOLLOW($A$)

  for all ($B \to \alpha\ A\ \beta$) $\in P$ **do**

  add FIRST(β)$\backslash\{\varepsilon\}$ to FOLLOW(A)

  for all ($B \to \alpha\ A\ \beta$) $\in P$ and $\varepsilon \in$ FIRST(β)

  do

  add FOLLOW(B) to FOLLOW(A)

  for all ($B \to \alpha\ A$) $\in P$

  do

  add FOLLOW(B) to FOLLOW(A)

Follow

- Follow (E) = {$, +,) }
- Follow (T) = {$, +, *,) }
- Follow (F) = {$, +, *,) }

- si means shift state i
- rj means reduce by production numbered j
- Blank means error

Shift and Accept

State	Action						Goto		
	id	+	*	(	)	$	E	T	F
0	s5			s3			1	2	4
1		s6				accept			
2			s10						
3	s5			s3			7	2	4
4									
5									
6	s5			s3				8	4

Shift and Accept

State	Action						Goto		
	id	+	*	(	)	$	E	T	F
7		s6			s9				
8			s10						
9									
10	s5			s3					11
11									

Shift, Accept, Reduce

State	Action						Goto		
	id	+	*	(	)	$	E	T	F
0	s5			s3			1	2	4
1		s6				accept			
2		r2	s10		r2	r2			
3	s5			s3			7	2	4
4		r4	r4		r4	r4			
5		r6	r6		r6	r6			
6	s5			s3				8	4

Shift, Accept and Reduce

State	Action						Goto		
	id	+	*	(	)	$	E	T	F
7		s6			s9				
8		r1	s10		r1	r1			
9		r5	r5		r5	r5			
10	s5			s3					11
11		r3	r3		r3	r3			

SLR Parsing

$$\underbrace{(s_0\, X_1\, s_1\, X_2\, s_2 \ldots X_m\, s_m,}_{\text{stack}} \qquad \underbrace{a_i\, a_{i+1} \ldots a_n\, \mathbf{\$})}_{\text{input}}$$

Parsing action

- If *action*[s_m,a_i] = shift *s*, then push a_i, push *s*, and advance input:
  $(s_0\ X_1\ s_1\ X_2\ s_2 \ldots X_m\ s_m\ a_i\ s, \quad a_{i+1} \ldots a_n\ \mathbf{\$})$
- If *action*[s_m,a_i] = reduce $A \rightarrow \beta$ and *goto*[s_{m-r},A] = *s* with $r=|\beta|$ then pop 2*r* symbols, push *A*, and push *s*:
  $(s_0\ X_1\ s_1\ X_2\ s_2 \ldots X_{m-r}\ s_{m-r}\ A\ s, \quad a_i\ a_{i+1} \ldots a_n\ \mathbf{\$})$

$E \rightarrow \boxed{E + T}$

2×3

$s_0 \ldots s_{m-r} \ldots s_m,$ $\quad a_i\ a \ldots$

$s_0 \ldots s_{m-r}\ A\ s$ $\quad a_i$

- If $action[s_m,a_i]$ = accept, then stop
- If $action[s_m,a_i]$ = error, then attempt recovery

Parsing algorithm

- Set input to point to the first symbol of w$
- Repeat
  - Let s be the state on the top of the stack
  - Let a be the symbol pointed to by ip
  - If action [s, a] = shift s’ then
    - Push a then s’ on top of the stack
    - Move input to the next input symbol

Parsing algorithm

- Else if action [s, a] = reduce A → β then
  - Pop 2 * | β | symbols off the stack
- Let s’ be the state now on the top of the stack
  - Push A then goto [s’, A] on top of the stack
  - Output the production A → β
- Else if action[s, a] = accept then return;
- Else error()

Parsing action

Stack	Input	Action
0	id * id + id $	[0, id] → s5 , shift
0 id 5	* id + id $	[5, *], r6, pop 2 symbols, Goto[0, F] = 4 F → id
0 F 4	* id + id $	[4, *], r4, pop 2 symbols, Goto[0, T] = 2 T → F
0 T 2	* id + id $	[2, *], → s10, shift
0 T 2 * 10	id + id $	[10, id] → s5, shift
0 T 2 * 10 id 5	+ id $	[5, +], r6, pop 2 symbols, Goto[10, F] = 11 F → id

Parsing action

Stack	Input	Action
0 T 2 * 10 F 11	+ id $	[11, +] = r3, pop 6 symbols and goto[0, T] = 2 T → T * F
0 T 2	+ id $	[2, +] → r2, pop 2 symbols and goto [0 , E] = 1 E → T
0 E 1	+ id $	[1, +] → s6, shift
0 E 1 + 6	id $	[6, id] = s5, shift
0 E 1 + 6 id 5	$	[5, $] = r6, pop 2 symbols, goto [6, F] = 4 F → id
0 E 1 + 6 F 4	$	[4 , $] = r4, pop 2 symbols, goto [6, T] = 8 T → F

Parsing action

Stack	Input	Action
$\underbrace{0\ E\ 1 + 6\ T\ 8}$	$	[8, $] = r1, pop 6 symbols from the stack and goto [0, E] = 1 E → E+T
0 E 1	$	[1, $] = accept, hence successful parsing

Problems with SLR grammar

- Every SLR grammar is unambiguous, but **not** every unambiguous grammar is SLR
- Consider for example the unambiguous grammar

Example

id = *id

*id = *id

- S → L = R
- S→ R
- L → * R
- L → id
- R → L

Items set

- I_0:
  $S' \rightarrow \bullet S$

1. $S \rightarrow \bullet L\mathbf{=}R$
2. $S \rightarrow \bullet R$
3. $L \rightarrow \bullet \mathbf{*}R$
4. $L \rightarrow \bullet \mathbf{id}$
5. $R \rightarrow \bullet L$

- I_1: (I_0 , S)
  $S' \rightarrow S\bullet$
- I_2: (I_0 , L)
  $S \rightarrow L\bullet\mathbf{=}R$
  $R \rightarrow L\bullet$
- I_3: (I_0 ,R)
  $S \rightarrow R\bullet$

Items set

- I_4: $(I_0 , *)$ $(I_4 , *)$ $(I_6 , *)$
  $L \rightarrow \mathbf{*}\bullet R$
  $R \rightarrow \bullet L$
  $L \rightarrow \bullet\mathbf{*}R$
  $L \rightarrow \bullet\mathbf{id}$
- I_5: (I_0 , id) (I_4 , id) (I_6 , id)
  $L \rightarrow \mathbf{id}\bullet$
- I_9: (I_6 , R)
  $S \rightarrow L\mathbf{=}R\bullet$
- I_6: $(I_2 , =)$
  $S \rightarrow L\mathbf{=}\bullet R$
  $R \rightarrow \bullet L$
  $L \rightarrow \bullet\mathbf{*}R$
  $L \rightarrow \bullet\mathbf{id}$
- I_7: (I_4 , R)
  $L \rightarrow \mathbf{*}R\bullet$
- I_8: (I_4 , L) (I_6 , L)
  $R \rightarrow L\bullet$

- Follow (S) = { $}
- Follow (L) = { =, $ }
- Follow (R) = { $, =}

State	Action				Goto		
	id	=	*	$	S	L	R
0	s5		s4		1	2	3
1				accept			
2		S6 /r5		r5			
3				r2			
4	s5		s4			8	7
5		r4		r4			
6	s5		s4			8	9

Shift/ Reduce

Shift

0 L 2 = 6 * id $

0 L 2 = 6 * 4 id $

0 L 2 = 6 * 4 id 5 $

0 L 2 = 6 * 4 L 8 $

L → id

R → L

State	Action				Goto		
	id	=	*	$	S	L	R
7		r3		r3			
8		r5		r5			
9				r1			

0 L 2 = 6 * 4 R 7 $ R3

0 L 2 = 6 L 8 $ R → L

0 L 2 = 6 R 9 $ R1

0 S 1 $

$ 0 id = *id $ s5

0 id5 = *id $ r4 L → id

0 L2 = *id $ r5 R → L

0 R3 = *id$

Reduce → Error

Conflict

- Shift / reduce conflict arises
- Because the grammar is not SLR(1)
- Follow information alone is not sufficient
- Hence, powerful parser is required

Summary

- Learnt to parse the SLR(1) grammar using the SLR(1) parsing algorithm
- Some grammar results in Shift / Reduce conflict