SLR(1) PARSER

→ Optimized Version of LR(0) parser.

→ Avoid Shift-Reduce Conflicts.

What is Shift-Reduce Conflict?

→ While Creating parse table, occurance of Shift State and Reduce State in one cell.

like:-

State	(	)	x	$	A
0					
1		S2 / r3			
2					

This is Shift Reduce Conflict.

→ Shift Reduce Conflict means that grammar is not correct.

Example of SLR(1) PARSER

Same as LR(0) except process to create passe table.

EXAMPLE

Given a CFG, check that grammar follows SLR(1) and create parse tree for given input string.

$S \rightarrow AA$

$A \rightarrow aA$

$A \rightarrow b$

Step 1: Adding Augmented ~~Error~~ Production

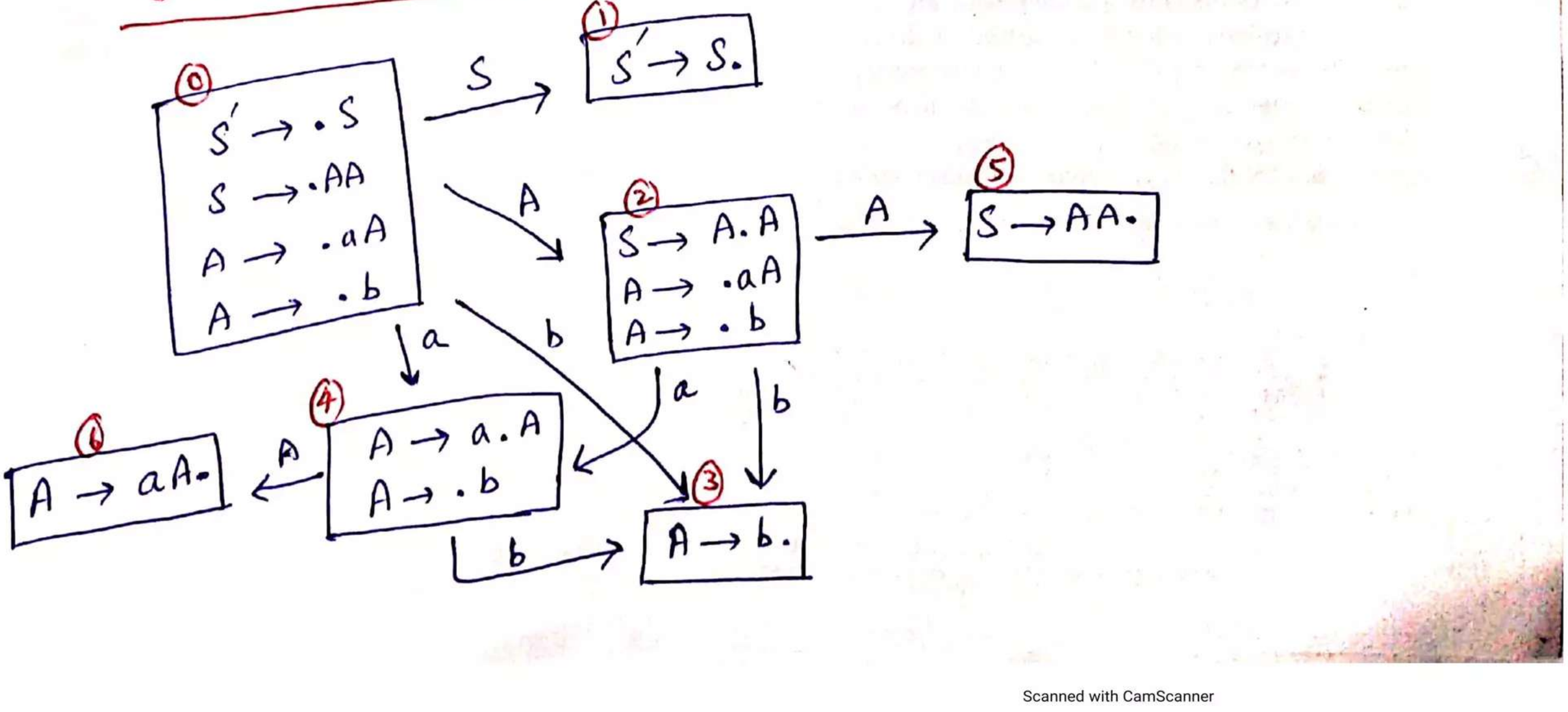
0 $S' \rightarrow S$ (Augmented Production)

1 $S \rightarrow AA$

2 $A \rightarrow aA$

3 $A \rightarrow b$

Step 2: Draw State Diagram



Step3: PARSE TABLE

STATE	TERMINALS (ACTION)			NON TERMINALS	
	a	b	$	S	A
0	S4	S3		1	2
1			Accept		
2	S4	S3			5
3	R3	R3	R3		
4		S3			6
5			R1		
6	R2	R2	R2		

The difference (with LR(0))

- When there is no any outside edge from any State, and that is not ~~terminal~~ augmented production ~~rule~~, either, then we'll not write reduce in every cell. But! we'll find follow of left side of Production and write on those cells only.

* There is no outgoing edge from state 3 which is

$A \rightarrow b.$

So, we'll find FOLLOW(A) = a, b, $

We'll write reduce at these places.

+ There is no outgoing edge from state 5, which is

$S \rightarrow AA.$

So, we'll find FOLLOW(S) = $

We'll write reduce at these places.

+ There is no outgoing edge from state 6 which is

$A \rightarrow aA.$

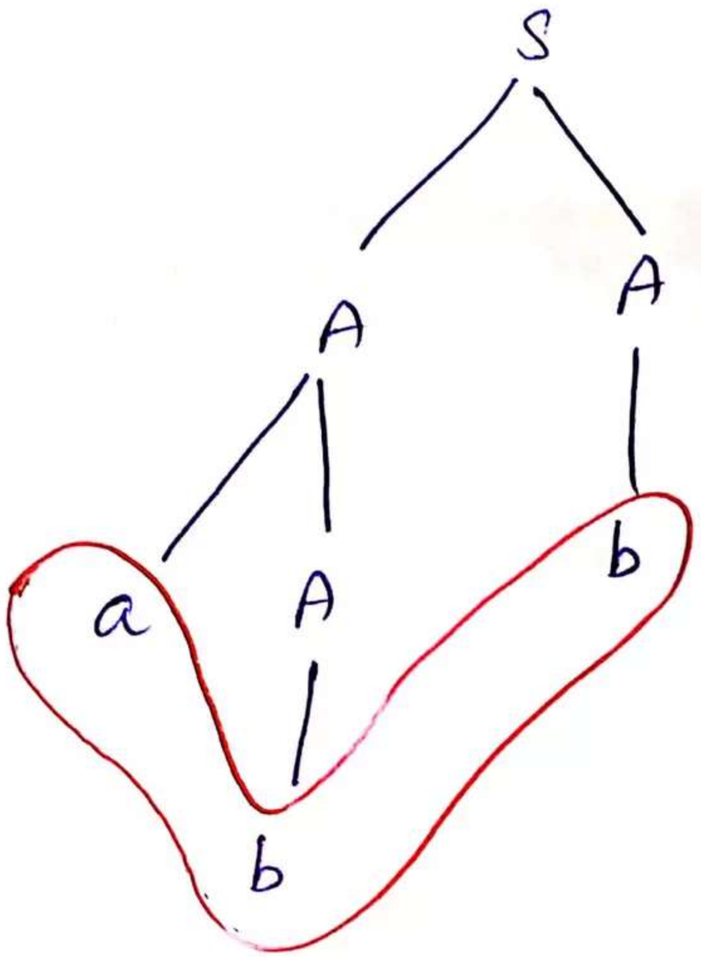
So, we'll find FOLLOW(A) = a, b, $.

We'll write reduce at these places.

Step 4 : Stack Implementation

Stack	Input	Action
0	abb $	S4
0 a 4	bb $	S3
0 a 4 b 3	b $	R3 : A → b
0 a 4 A 6	b $	R2 : A → aA
0 A 2	b $	S3
0 A 2 b 3	$	R3 : A → b
0 A 2 A 5	$	R1 : S → AA
0 S 1	$	Accept.

Step 5: PARSE TREE



SLR PARSER

Steps:

1. create augment grammar
2. generate kernel items
3. find closure
4. compute goto()
5. construct parsing table
6. parse the string

Let us consider grammar:

S->a

S->(L)

L->S

L->L,S

- **Step :1 Create augment grammar**

  S is start symbol in the given grammar

  Augment grammar is S'-> S

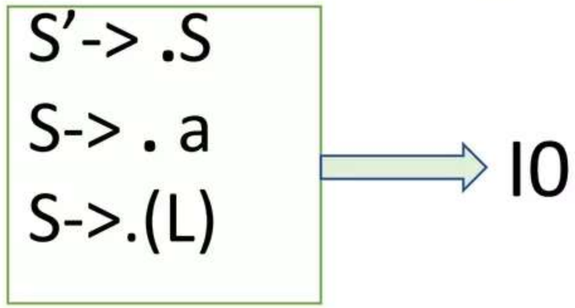
- **Step :2 Generate kernel items**

Introduce dot in RHS of the production``

S'-> •S

- **Step :3 Find closure**

(*Rule:* *A -> α.Xβ i.e if there is nonterminal next to dot then Include X production))*



S'-> .S
S-> . a
S->.(L)

I0

Step :4 compute goto()

Goto (I0,S)

S'-> S .

I1

Goto (I0,a)

S-> a .

I2

Goto (I0,( )

S-> (.L)
L->.S
L>.L,S
S->.a
S->.(L)

I3

Goto (I3,L)

S-> (L .)
L->L . ,S

I4

Goto (I3,S)

L->S .

I5

Goto (I3,a)

S->a .

I2

Goto (I3, ( )

S-> (. L)
L->.S
L>.L,S
S->.a
S->.(L)

I3

Goto (I4,))

S-> (L) .

I6

Goto (I4, ,)

L->L , . S
S-> . a
S->.(L)

I7

Goto (I7,S)

L->L , S .

I8

Goto (I7,a)

S-> a .

I2

Goto (I7,( )

S-> (.L)
L->.S
L>.L,S
S->.a
S->.(L)

I3

label	Production	Ending iteration
R0	S'-> S .	I1
R1	S->a .	I2
R2	S->(L) .	I6
R3	L->S .	I5
R4	L->L,S .	I8

Step :5 construct parsing table

Label the rows of table with no. of iterations

Divide the Column into two parts

- Action part: terminal symbols
- Goto part: Nonterminal symbols

	Terminals					Non terminals	
	a	(	)	,	$	S	L
0							
1							
2							
3							
4							
5							
6							
7							
8							

Step :5 construct parsing table

Label the rows of table with no. of iterations

Divide the Column into two parts

- Action part: terminal symbols
- Goto part: Nonterminal symbols

	ACTION					GOTO	
	a	(	)	,	$	S	L
0	**S2**	**S3**				**1**	
1							
2							
3	**S2**	**S3**				**5**	**4**
4			**S6**	**S7**			
5							
6							
7	**S2**	**S3**				**8**	
8							

from step 4:

Goto (I0, S)	1	**Goto (I3, ( )**	3
Goto (I0, a)	2	**Goto (I4,))**	6
Goto (I0, ( )	3	**Goto (I4, ,)**	7
Goto (I3, L)	4	**Goto (I7, S)**	8
Goto (I3, S)	5	**Goto (I7, a)**	2
Goto (I3, a)	2	**Goto (I7, ( )**	3

Step :5 construct parsing table

Label the rows of table with no. of iterations

Divide the Column into two parts

- Action part: terminal symbols
- Goto part: Nonterminal symbols

	ACTION					GOTO	
	a	(	)	,	$	S	L
0	**S2**	**S3**				**1**	
1					**accept**		
2			**R1**	**R1**	**R1**		
3	**S2**	**S3**				**5**	**4**
4			**S6**	**S7**			
5			**R3**	**R3**			
6			**R2**	**R2**	**R2**		
7	**S2**	**S3**				**8**	
8			**R4**	**R4**			

label	Production	Ending iteration	Follow(LHS)
R0	S'-> S .	I1	Follow(S')={ **$** }
R1	S-> a .	I2	Follow(S)={ **$) ,** }
R2	S-> (L) .	I6	Follow(S)={ **$) ,** }
R3	L-> S .	I5	Follow(L)={ **) ,** }
R4	L-> L , S .	I8	Follow(L)={ **) ,** }

Step :5 construct parsing table

Label the rows of table with no. of iterations

Divide the Column into two parts

- Action part: terminal symbols
- Goto part: Nonterminal symbols

	ACTION					GOTO	
	a	(	)	,	$	S	L
0	**S2**	**S3**				**1**	
1					**accept**		
2			**R1**	**R1**	**R1**		
3	**S2**	**S3**				**5**	**4**
4			**S6**	**S7**			
5			**R3**	**R3**			
6			**R2**	**R2**	**R2**		
7	**S2**	**S3**				**8**	
8			**R4**	**R4**			

Goto (I0, S)	1	**Goto (I3, ( )**	3
Goto (I0, a)	2	**Goto (I4,))**	6
Goto (I0, ( )	3	**Goto (I4, ,)**	7
Goto (I3, L)	4	**Goto (I7, S)**	8
Goto (I3, S)	5	**Goto (I7, a)**	2
Goto (I3, a)	2	**Goto (I7, ( )**	3

label	Production	Ending iteration	Follow(LHS)
R0	S'-> S .	I1	Follow(S')={ **$** }
R1	S-> a .	I2	Follow(S)={ **$) ,** }
R2	S-> (L) .	I6	Follow(S)={ **$) ,** }
R3	L-> S .	I5	Follow(L)={ **) ,** }
R4	L-> L , S .	I8	Follow(L)={ **) ,** }

Step :6 String parsing

Let string w=(a,a) derived from the given grammar

Stack	Input	Action
$0	(a,a)$	S3

	ACTION					GOTO	
	a	(	)	,	$	S	L
0	S2	S3				1	
1					accept		
2			R1	R1	R1		
3	S2	S3				5	4
4			S6	S7			
5			R3	R3			
6			R2	R2	R2		
7	S2	S3				8	
8			R4	R4			

S – Shift action

* shift input symbol to the stack
* shift number to the stack

R – Reduce action

* pop 2 times of RHS symbols
* shift LHS to the stack
* find the number in the parsing table for the last two elements in the stack

label	Production
R0	S'-> S .
R1	S-> a .
R2	S-> (L) .
R3	L-> S .
R4	L-> L , S .

Step :6 String parsing

Let string w=(a,a) derived from the given grammar

Stack	Input	Action
$0	(a,a)$	S3
$0(3	a,a)$	S2

	ACTION					GOTO	
	a	(	)	,	$	S	L
0	S2	S3				1	
1					accept		
2			R1	R1	R1		
3	S2	S3				5	4
4			S6	S7			
5			R3	R3			
6			R2	R2	R2		
7	S2	S3				8	
8			R4	R4			

S – Shift action

* shift input symbol to the stack
* shift number to the stack

R – Reduce action

* pop 2 times of RHS symbols
* shift LHS to the stack
* find the number in the parsing table for the last two elements in the stack

label	Production
R0	S'-> S .
R1	S-> a .
R2	S-> (L) .
R3	L-> S .
R4	L-> L , S .

Step :6 String parsing

Let string w=(a,a) derived from the given grammar

Stack	Input	Action
$0	(a,a)$	S3
$0(3	a,a)$	S2
$0(3~~a2~~	,a)$	R1

	ACTION					GOTO	
	a	(	)	,	$	S	L
0	S2	S3				1	
1					accept		
2			R1	R1	R1		
3	S2	S3				5	4
4			S6	S7			
5			R3	R3			
6			R2	R2	R2		
7	S2	S3				8	
8			R4	R4			

S – Shift action

* shift input symbol to the stack
* shift number to the stack

R – Reduce action

* pop 2 times of RHS symbols
* shift LHS to the stack
* find the number in the parsing table for the last two elements in the stack

label	Production
R0	S'-> S .
R1	S-> a .
R2	S-> (L) .
R3	L-> S .
R4	L-> L , S .

Step :6 String parsing

Let string w=(a,a) derived from the given grammar

Stack	Input	Action
$0	(a,a)$	S3
$0(3	a,a)$	S2
$0(3~~a2~~	,a)$	R1
$0(3~~S5~~	,a)$	R3

	ACTION					GOTO	
	a	(	)	,	$	S	L
0	S2	S3				1	
1					accept		
2			R1	R1	R1		
3	S2	S3				5	4
4			S6	S7			
5			R3	R3			
6			R2	R2	R2		
7	S2	S3				8	
8			R4	R4			

S – Shift action

* shift input symbol to the stack
* shift number to the stack

R – Reduce action

* pop 2 times of RHS symbols
* shift LHS to the stack
* find the number in the parsing table for the last two elements in the stack

label	Production
R0	S'-> S .
R1	S-> a .
R2	S-> (L) .
R3	L-> S .
R4	L-> L , S .

Step :6 String parsing

Let string w=(a,a) derived from the given grammar

Stack	Input	Action
$0	(a,a)$	S3
$0(3	a,a)$	S2
$0(3~~a2~~	,a)$	R1
$0(3~~S5~~	,a)$	R3
$0(3L4	,a)$	S7

	ACTION					GOTO	
	a	(	)	,	$	S	L
0	S2	S3				1	
1					accept		
2			R1	R1	R1		
3	S2	S3				5	4
4			S6	S7			
5			R3	R3			
6			R2	R2	R2		
7	S2	S3				8	
8			R4	R4			

S – Shift action

* shift input symbol to the stack
* shift number to the stack

R – Reduce action

* pop 2 times of RHS symbols
* shift LHS to the stack
* find the number in the parsing table for the last two elements in the stack

label	Production
R0	S'-> S .
R1	S-> a .
R2	S-> (L) .
R3	L-> S .
R4	L-> L , S .

Step :6 String parsing

Let string w=(a,a) derived from the given grammar

Stack	Input	Action
$0	(a,a)$	S3
$0(3	a,a)$	S2
$0(3~~a2~~	,a)$	R1
$0(3~~S5~~	,a)$	R3
$0(3L4	,a)$	S7

	ACTION					GOTO	
	a	(	)	,	$	S	L
0	S2	S3				1	
1					accept		
2			R1	R1	R1		
3	S2	S3				5	4
4			S6	S7			
5			R3	R3			
6			R2	R2	R2		
7	S2	S3				8	
8			R4	R4			

S – Shift action

* shift input symbol to the stack
* shift number to the stack

R – Reduce action

* pop 2 times of RHS symbols
* shift LHS to the stack
* find the number in the parsing table for the last two elements in the stack

label	Production
R0	S'-> S .
R1	S-> a .
R2	S-> (L) .
R3	L-> S .
R4	L-> L , S .

Step :6 String parsing

Let string w=(a,a) derived from the given grammar

Stack	Input	Action
$0	(a,a)$	S3
$0(3	a,a)$	S2
$0(3~~a2~~	,a)$	R1
$0(3~~S5~~	,a)$	R3
$0(3L4	,a)$	S7
$0(3L4 , 7	a)$	S2

	ACTION					GOTO	
	a	(	)	,	$	S	L
0	S2	S3				1	
1					accept		
2			R1	R1	R1		
3	S2	S3				5	4
4			S6	S7			
5			R3	R3			
6			R2	R2	R2		
7	S2	S3				8	
8			R4	R4			

S – Shift action

* shift input symbol to the stack
* shift number to the stack

R – Reduce action

* pop 2 times of RHS symbols
* shift LHS to the stack
* find the number in the parsing table for the last two elements in the stack

label	Production
R0	S'-> S .
R1	S-> a .
R2	S-> (L) .
R3	L-> S .
R4	L-> L , S .

Step :6 String parsing

Let string w=(a,a) derived from the given grammar

Stack	Input	Action
$0	(a,a)$	S3
$0(3	a,a)$	S2
$0(3~~a2~~	,a)$	R1
$0(3~~S5~~	,a)$	R3
$0(3L4	,a)$	S7
$0(3L4 , 7	a)$	S2
$0(3L4 , 7~~a2~~	)$	R1

	ACTION					GOTO	
	a	(	)	,	$	S	L
0	S2	S3				1	
1					accept		
2			R1	R1	R1		
3	S2	S3				5	4
4			S6	S7			
5			R3	R3			
6			R2	R2	R2		
7	S2	S3				8	
8			R4	R4			

S – Shift action

* shift input symbol to the stack
* shift number to the stack

R – Reduce action

* pop 2 times of RHS symbols
* shift LHS to the stack
* find the number in the parsing table for the last two elements in the stack

label	Production
R0	S'-> S .
R1	S-> a .
R2	S-> (L) .
R3	L-> S .
R4	L-> L , S .

Step :6 String parsing

Let string w=(a,a) derived from the given grammar

Stack	Input	Action
$0	(a,a)$	S3
$0(3	a,a)$	S2
$0(3~~a2~~	,a)$	R1
$0(3~~S5~~	,a)$	R3
$0(3L4	,a)$	S7
$0(3L4 , 7	a)$	S2
$0(3L4 , 7~~a2~~	)$	R1
$0(3~~L4 , 7S8~~	)$	R4

	ACTION					GOTO	
	a	(	)	,	$	S	L
0	S2	S3				1	
1					accept		
2			R1	R1	R1		
3	S2	S3				5	4
4			S6	S7			
5			R3	R3			
6			R2	R2	R2		
7	S2	S3				8	
8			R4	R4			

S – Shift action

* shift input symbol to the stack
* shift number to the stack

R – Reduce action

* pop 2 times of RHS symbols
* shift LHS to the stack
* find the number in the parsing table for the last two elements in the stack

label	Production
R0	S'-> S .
R1	S-> a .
R2	S-> (L) .
R3	L-> S .
R4	L-> L , S .

Step :6 String parsing

Let string w=(a,a) derived from the given grammar

Stack	Input	Action
$0	(a,a)$	S3
$0(3	a,a)$	S2
$0(3~~a2~~	,a)$	R1
$0(3~~S5~~	,a)$	R3
$0(3L4	,a)$	S7
$0(3L4 , 7	a)$	S2
$0(3L4 , 7~~a2~~	)$	R1
$0(3~~L4 , 7S8~~	)$	R4
$0(3L4	) $	S6

	ACTION					GOTO	
	a	(	)	,	$	S	L
0	S2	S3				1	
1					accept		
2			R1	R1	R1		
3	S2	S3				5	4
4			S6	S7			
5			R3	R3			
6			R2	R2	R2		
7	S2	S3				8	
8			R4	R4			

S – Shift action

* shift input symbol to the stack
* shift number to the stack

R – Reduce action

* pop 2 times of RHS symbols
* shift LHS to the stack
* find the number in the parsing table for the last two elements in the stack

label	Production
R0	S'-> S .
R1	S-> a .
R2	S-> (L) .
R3	L-> S .
R4	L-> L , S .

Step :6 String parsing

Let string w=(a,a) derived from the given grammar

Stack	Input	Action
$0	(a,a)$	S3
$0(3	a,a)$	S2
$0(3~~a2~~	,a)$	R1
$0(3~~S5~~	,a)$	R3
$0(3L4	,a)$	S7
$0(3L4 , 7	a)$	S2
$0(3L4 , 7~~a2~~	)$	R1
$0(3~~L4 , 7S8~~	)$	R4
$0(3L4	) $	S6
$0~~(3L4)6~~	$	R2

	ACTION					GOTO	
	a	(	)	,	$	S	L
0	S2	S3				1	
1					accept		
2			R1	R1	R1		
3	S2	S3				5	4
4			S6	S7			
5			R3	R3			
6			R2	R2	R2		
7	S2	S3				8	
8			R4	R4			

S – Shift action

* shift input symbol to the stack
* shift number to the stack

R – Reduce action

* pop 2 times of RHS symbols
* shift LHS to the stack
* find the number in the parsing table for the last two elements in the stack

label	Production
R0	S'-> S .
R1	S-> a .
R2	S-> (L) .
R3	L-> S .
R4	L-> L , S .

Step :6 String parsing

Let string w=(a,a) derived from the given grammar

Stack	Input	Action
$0	(a,a)$	S3
$0(3	a,a)$	S2
$0(3~~a2~~	,a)$	R1
$0(3~~S5~~	,a)$	R3
$0(3L4	,a)$	S7
$0(3L4 , 7	a)$	S2
$0(3L4 , 7~~a2~~	)$	R1
$0(3~~L4 , 7S8~~	)$	R4
$0(3L4	) $	S6
$0~~(3L4)6~~	$	R2
$0S1	$	accept
Hence the string parsed successfully !!!!!!		

	ACTION					GOTO	
	a	(	)	,	$	S	L
0	S2	S3				1	
1					accept		
2			R1	R1	R1		
3	S2	S3				5	4
4			S6	S7			
5			R3	R3			
6			R2	R2	R2		
7	S2	S3				8	
8			R4	R4			

S – Shift action

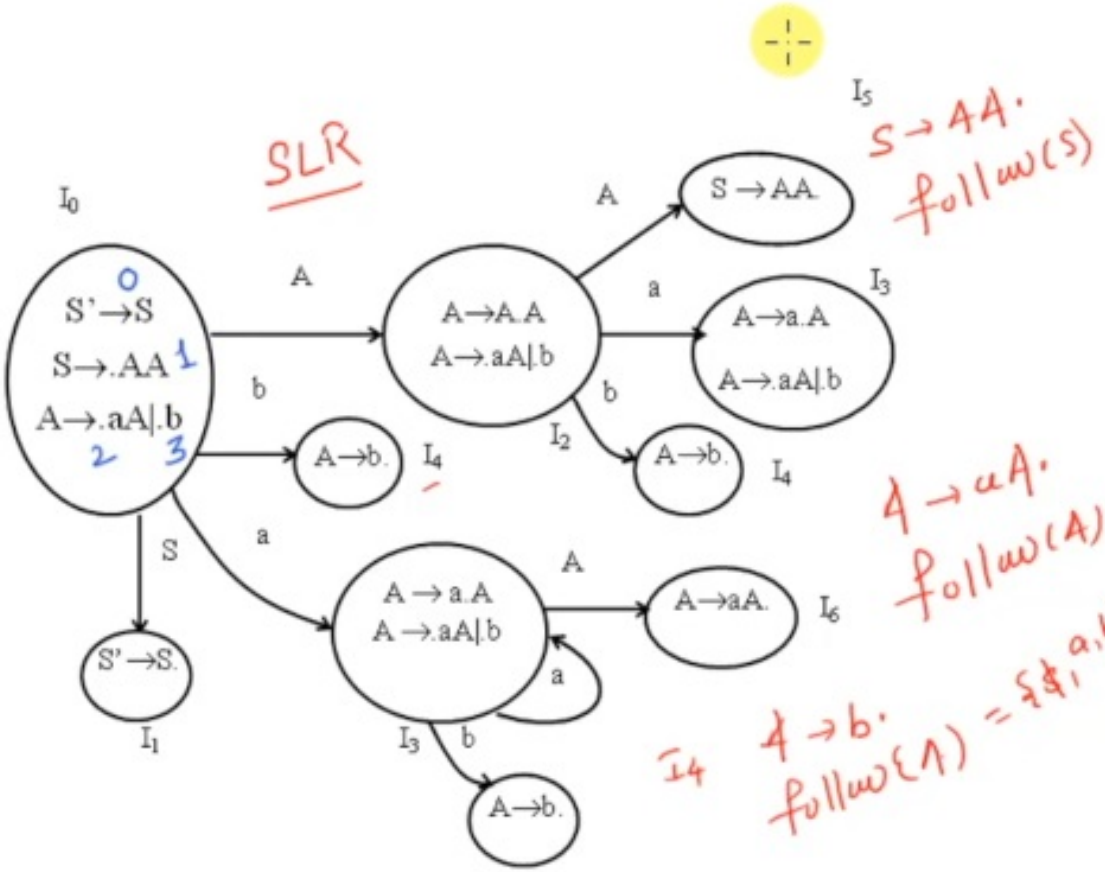
* shift input symbol to the stack
* shift number to the stack

R – Reduce action

* pop 2 times of RHS symbols
* shift LHS to the stack
* find the number in the parsing table for the last two elements in the stack

label	Production
R0	S'-> S .
R1	S-> a .
R2	S-> (L) .
R3	L-> S .
R4	L-> L , S .

Construction of SLR/SLR(1) Parsing table



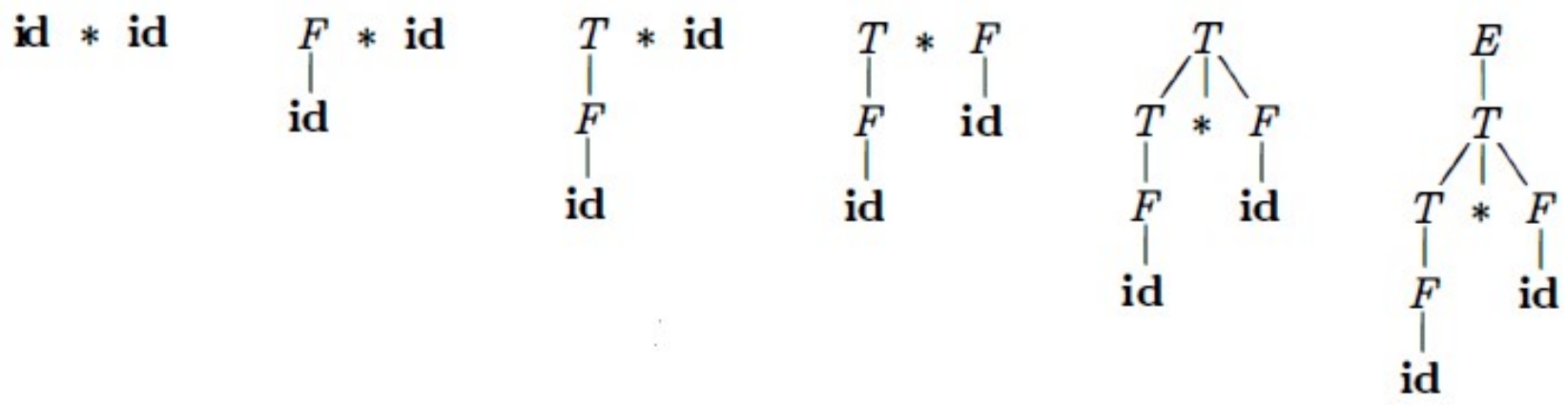
State/Item	Action			Go to	
	a	b	\$	S	A
I_0	I_3	I_4		I_1	I_2
I_1			Acc		
I_2	I_3	I_4			I_5
I_3	I_3	I_4			I_6
I_4	r_3	r_3	r_3		
I_5			r_1		
I_6	r_2	r_2	r_2		

CSE504 Compiler Design
Syntax Analysis (SLR Parser)

YoungMin Kwon

Bottom-Up Parsing

- Attempts to construct a parse tree beginning at the leaves and working up towards the root.



Bottom-up parse for id * id

Reductions

- Bottom-up parsing
  - Reducing a string w to the start symbol
  - At each reduction step, a particular substring matching the RHS of a production is replaced by the LHS.
  - Rightmost derivation is traced out in reverse.

E.g.

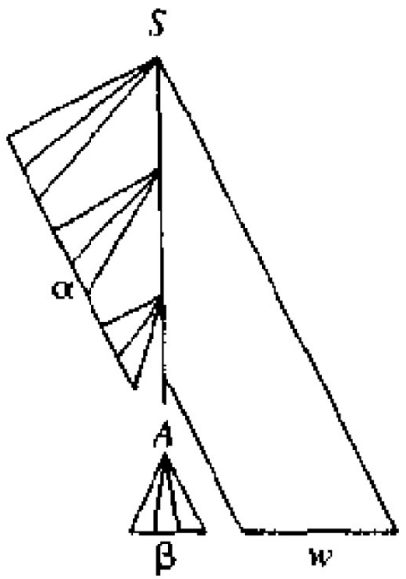
```
S -> aABe
A -> Abc | b
B -> d
```

```
abbcde
aAbcde
aAde
aABe
S
```

`abbcde` can be reduced to `S`

Handle Pruning

- Handle:
  - A handle of a right-sentential form γ is a production A->β and a position of γ where the β may be found and replaced by A to produce the previous step of rightmost derivation.
    - If $S \Rightarrow^* \alpha A w \Rightarrow \alpha \beta w$, then A -> β in the position following α is a handle of α β w.
  - E.g. In the previous example
    - aAbcde => abbcde, handle is A->b at position 2.
    - aAde => aAbcde, handle is A->Abc at position 2.
- Handle pruning:
  - A->β in α β w is a handle.
  - Reducing β to A can be thought as pruning the handle (removing the children of A from the parse tree).
- A Rightmost derivation in reverse can be obtained by handle pruning



Shift-Reduce Parsing

- Shift-Reduce parsing
  - A bottom-up parsing where a stack holds grammar symbols and an input buffer holds the rest of the string to be parsed.
  - While scanning the input from left to right, the parser shifts 0+ input symbols onto the stack
  - If it is ready to reduce the RHS of a production, pop the RHS from the stack and push the LHS to the stack.
  - Handles always appear at the top of the stack
- 4 Actions if Shift-Reduce Parsing
  - Shift: push the next input symbol to the stack
  - Reduce: pop the RHS of a production and push the LHS.
  - Accept: announce the success
  - Error: found an error

Shift-Reduce Parsing

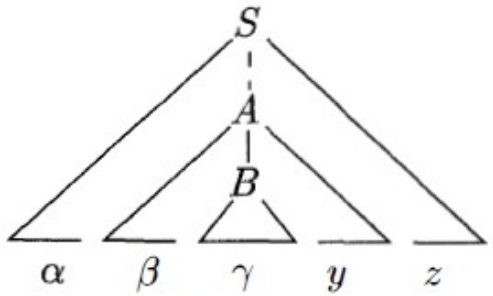
- Why the handle is always on top of the stack?
- Two possible cases of two successive steps of rightmost derivation

  (1) S =>* α A z => α β B y z => α β γ y z

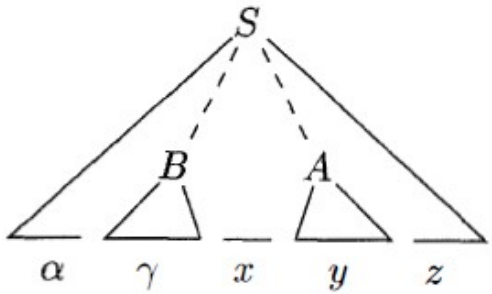
  - A is replaced by β B y (has a nonterminal B), then B is replaced.

  (2) S =>* α B x A z => α B x y z => α γ x y z

  - A is replaced by y (terminals only), then B is replaced.



Case (1)



Case (2)

Shift-Reduce Parsing

- Case 1: S =>* α A z => α β B y z => α β γ y z
  - ($ α β γ | y z $): the parser reached this configuration. γ is the handle and it is reduced to B.
  - ($ α β B | y z $): since B is the rightmost nonterminal in α β B y z, the handle cannot be inside the stack.
  - ($ α β B y | z $): the parser shifted y. β B y is the handle and it gets reduced to A.
- Case 2: S =>* α B x A z => α B x y z => α γ x y z
  - ($ α γ | x y z $): the parser reached this configuration. γ is the handle and it is reduced to B
  - ($ α B x y | z $): after shifting x y, get the next handle y on top of the stack and reduce it to A
  - ($ α B x A | z $): configuration after the reduction.

Shift-Reduce Parsing

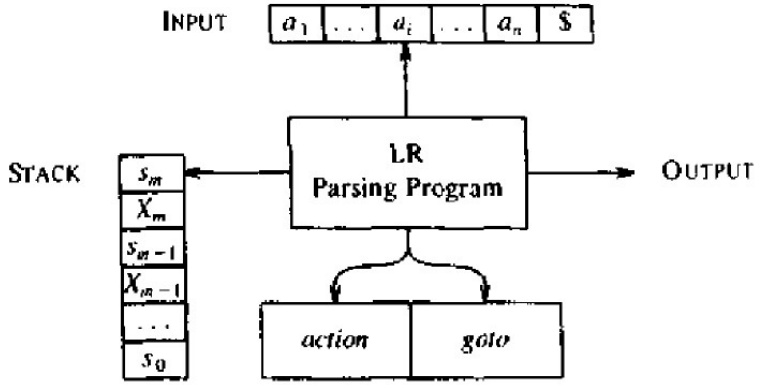
- Viable Prefixes
  - The set of prefixes of right-sentential forms that can appear on the stack of shift-reduce parser.
  - A prefix of a right-sentential form that does not continue past the right end of the rightmost handle.

LR Parsers

- LR(k) Parsing:
  - L: left-to-right scanning of the input.
  - R: constructing the rightmost derivation in reverse.
  - k: number of input symbols of lookahead.
- SLR (Simple LR): easiest to implement, least powerful.
- Canonical LR: most powerful, most expensive.
- LALR (look-ahead LR): intermediate in power and cost. Work with most programming language grammars.

LR Parsing Algorithm

- Configuration
  - (s_1, X_1, s_2, X_2 … s_n | a_1, a_2, …), where s_i is a state, X_i is a symbol, a_i is a token.

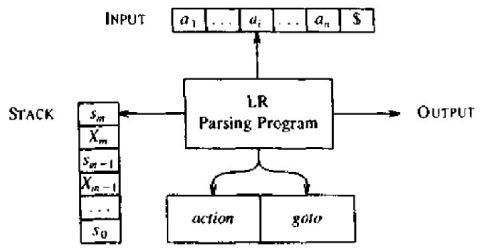


- 4 Actions of LR parser
  - Shift and go to state s
    - (… s_1 | a_1 a_2…) -> (… s_1 a_1 s | a_2 …)
  - Reduce X -> X_1 … X_n
    - (… s_0 X_1 s_1 … X_n s_n | a_1 …) -> (… s_0 X s | a_1 …), where s is the goto target of s_0 for symbol X.
  - Accept: finish with success
  - Error: found an error

LR Parsing Example

Parse id * id + id

(1) $E \rightarrow E + T$
(2) $E \rightarrow T$
(3) $T \rightarrow T * F$
(4) $T \rightarrow F$
(5) $F \rightarrow (E)$
(6) $F \rightarrow \mathbf{id}$



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			

LR Parsing Example

	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

Constructing SLR Parsing Table

- States of an SLR parser represent sets of items.
- LR(0) items of a grammar G is a production of G with a dot at some positions of the RHS.
  - E.g. A -> XYZ: A->.XYZ, A->X.YZ, A->XY.Z, A->XYZ.
    A -> ∈: A->.
  - **An item represents how much of a production we have seen**
    - X->X.YZ means, we've just seen a string derivable from X and expect to see a string derivable from YZ.
- Augmented grammar
  - Add a new start symbol S' and add a production S' -> S
  - To indicate when to stop.

Constructing SLR Parsing Table

- The central idea of SLR parsing is to construct a DFA recognizing the viable prefixes.
  - Imagine an NFA:
    - States are the items
    - Add a transition from A -> α.Xβ to A -> αX.β labeled X.
    - Add a transition from A -> α.Bβ to B->.γ labeled ϵ
  - Construct a DFA using the subset construction algorithm.
- Canonical LR(0) items
  - Give basis for the DFA states
  - CLOSURE and GOTO functions can find the canonical LR(0) items.
- Valid items
  - Item A -> β_1 . β_2 is valid for a viable prefix α β_1 if there is a derivation S' =>* α A w => α β_1 β_2 w

CLOSURE and GOTO functions

- CLOSURE(I)
  - If I is a set of items, CLOSURE(I) is a set of items built by the two rules
    - Add every item in I to CLOSURE(I)
    - If A -> α.Bβγ is in CLOSURE(I) and B->γ is a production, add B->.γ to CLOSURE(I). Apply this rule until no more new items are added to CLOSURE(I).
  - A -> α.Bβ in CLOSURE(I) means, we might next see a substring derivable from Bβ. Hence we add B->.γ to CLOSURE(I).
- GOTO(I,X)
  - GOTO(I,X) is the closure of the set of all items A -> αX.β such that A -> α.Xβ is in I.
  - The closures of items are the states of DFA and GOTO(I,X) specifies the transition from the state I under input X.

CLOSURE and GOTO functions

- Given the augmented grammar

```
E' -> E
E  -> E + T | T
T  -> T * F | F
F  -> (E) | id
```

- CLOSURE({ E'->.E }) is

```
{ E'->.E, E->.E+T, E->.T, T->.T*F, T->.F,
  F->.(E), F->.id }
```

- GOTO({ E'->E., E->E.+T }, +) is

```
{ E->E+.T, T->.T*F, T->.F, F->.(E),
  F->.id }
```

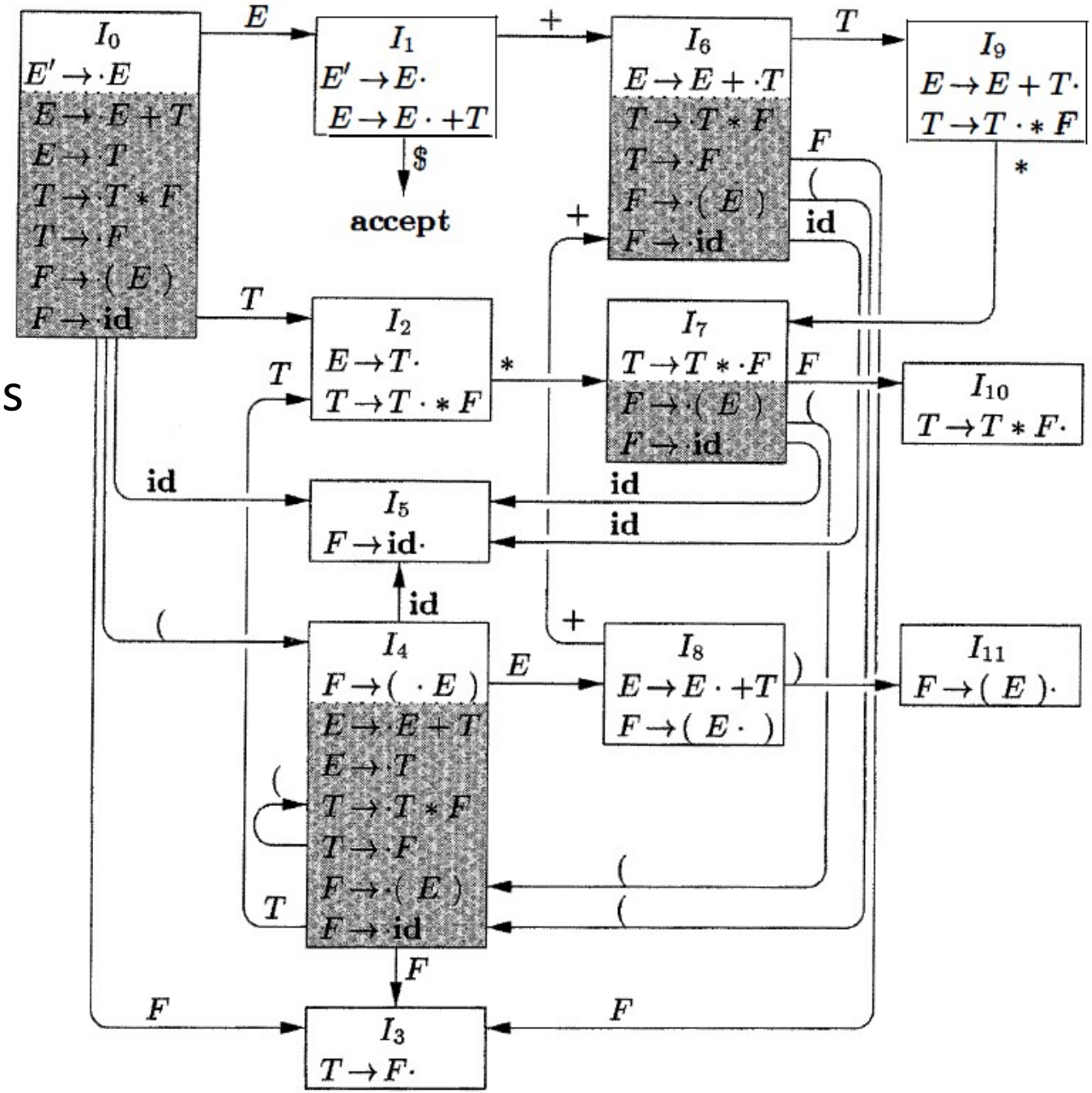
Canonical LR(0) items

```
SetOfItems CLOSURE(I) {
        J = I;
        repeat
                for ( each item A → α·Bβ 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 on one round;
        return J;
}

void items(G') {
        C = CLOSURE({[S' → ·S]});
        repeat
                for ( each set of items I in C )
                        for ( each grammar symbol X )
                                if ( GOTO(I, X) is not empty and not in C )
                                        add GOTO(I, X) to C;
        until no new sets of items are added to C on a round;
}
```

DFA for viable prefixes

$$
\begin{array}{lcl}
E' & \rightarrow & E \\
E & \rightarrow & E + T \mid T \\
T & \rightarrow & T * F \mid F \\
F & \rightarrow & (E) \mid \mathbf{id}
\end{array}
$$



Constructing SLR Parsing Tables

1. Construct $C = \{I_0, I_1, \ldots, I_n\}$, the collection of sets of LR(0) items for G'.

2. State i is constructed from I_i. The parsing actions for state i are determined as follows:

   (a) If $[A \to \alpha \cdot a\beta]$ is in I_i and $\text{GOTO}(I_i, a) = I_j$, then set $\text{ACTION}[i, a]$ to "shift j." Here a must be a terminal.

   (b) If $[A \to \alpha \cdot]$ is in I_i, then set $\text{ACTION}[i, a]$ to "reduce $A \to \alpha$" for all a in $\text{FOLLOW}(A)$; here A may not be S'.

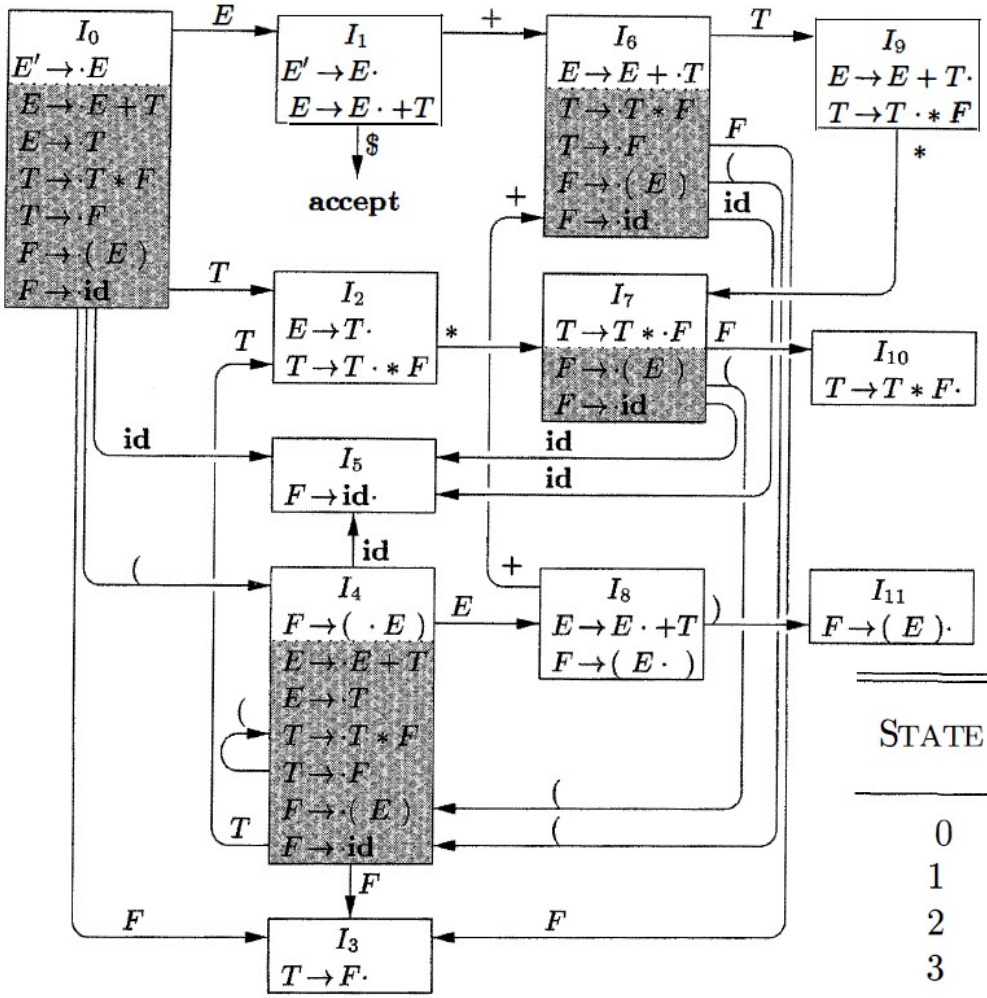
   (c) If $[S' \to S \cdot]$ is in I_i, then set $\text{ACTION}[i, \$]$ to "accept."

   If any conflicting actions result from the above rules, we say the grammar is not SLR(1). The algorithm fails to produce a parser in this case.

3. The goto transitions for state i are constructed for all nonterminals A using the rule: If $\text{GOTO}(I_i, A) = I_j$, then $\text{GOTO}[i, A] = j$.

4. All entries not defined by rules (2) and (3) are made "error."

5. The initial state of the parser is the one constructed from the set of items containing $[S' \to \cdot S]$.



$E' \to E$

(1) $E \to E + T$

(2) $E \to T$

(3) $T \to T * F$

(4) $T \to F$

(5) $F \to (E)$

(6) $F \to \mathbf{id}$

```
FIRST(E'): (, id
FIRST(E) : (, id
FIRST(T) : (, id
FIRST(F) : (, id
FOLLOW(E'): $
FOLLOW(E) : +, ), $
FOLLOW(T) : +, *, ), $
FOLLOW(F) : +, *, ), $
```

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			

Constructing SLR Parsing Tables

- Quiz: build an SLR Parsing Table for the grammar below.

```
E -> E + id
E -> id
```

Items

```
I0: E'->.E, E->.E+id, E->.id
I1: E'->E., E->E.+id
I2: E->id.
I3: E->E+.id
I4: E->E+id.
```

FIRST/FOLLOW

```
FIRST(E') = FIRST(E) = {id}
FOLLOW(E') = {$}
FOLLOW(E)= {+,$}
```

	+	id	$	E
0		s2		1
1	s3		acc	
2	r2			
3		s4		
4	r1		r1	