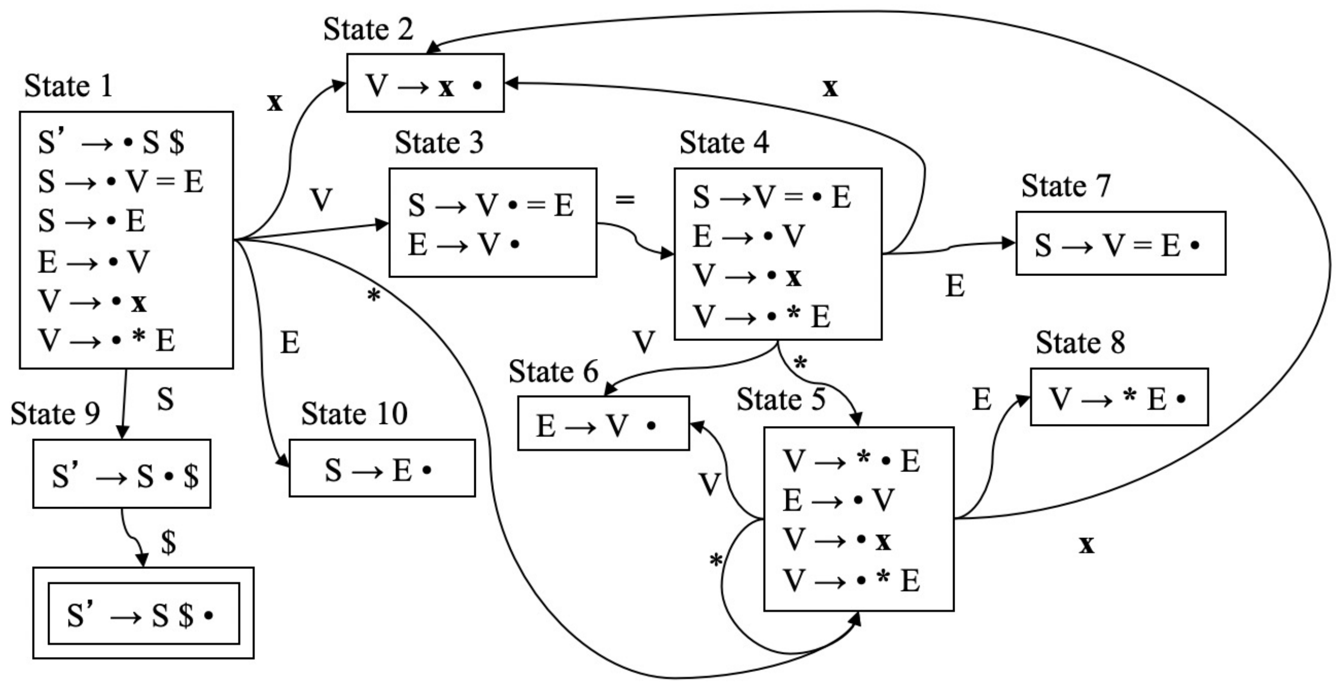
CMPSC 160
Translation of Programming Languages

Lecture 6: LR(0) + SLR + LR(1) Parsing

Example: States of an LR(0) parser

example

0 S' → S $
1 S → V = E
2 S → E
3 E → V
4 V → **x**
5 V → * E



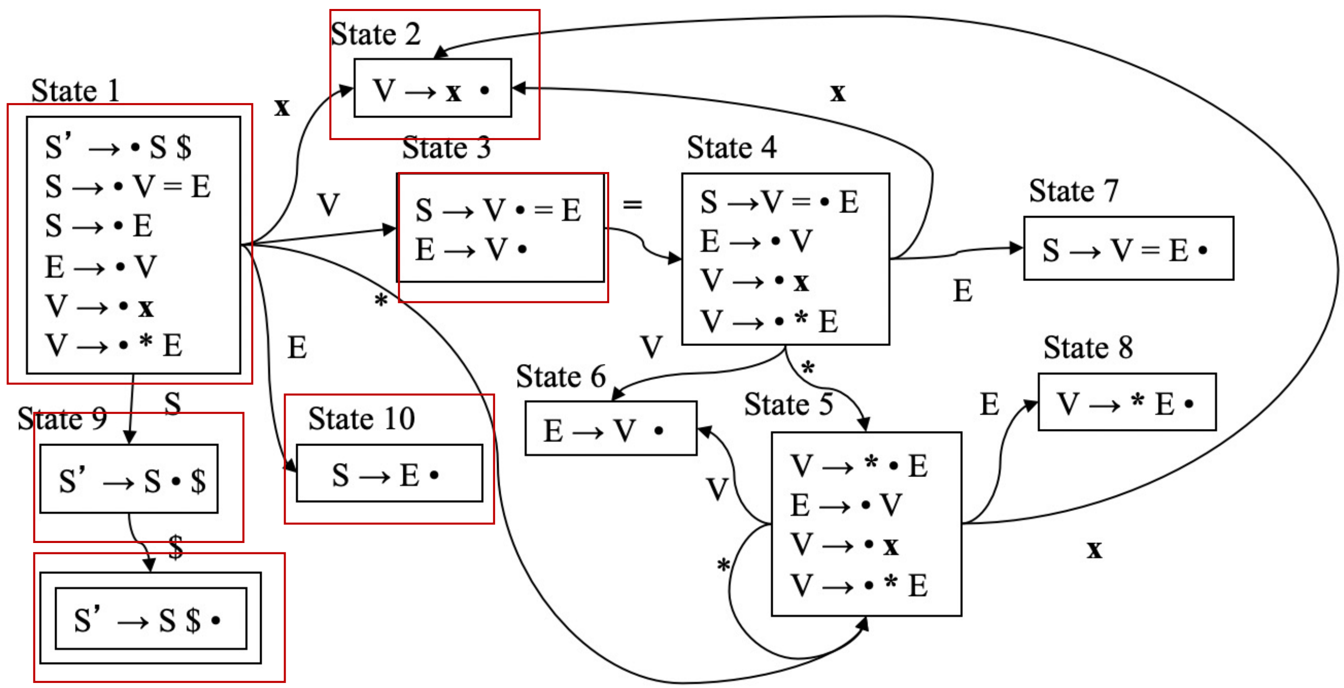
Closure: What other **items** are "equivalent" to the given item?
Given an **item** A → α• Bβ, closure(A → α• Bβ) is the smallest set that contains the item **A → α• Bβ**, and every item in closure(**B → • γ**) for every production B → γ ∈ CFG

Example: States of an LR(0) parser

Input: x$

example

0 S' → S $
1 S → V = E
2 S → E
3 E → V
4 V → **x**
5 V → * E



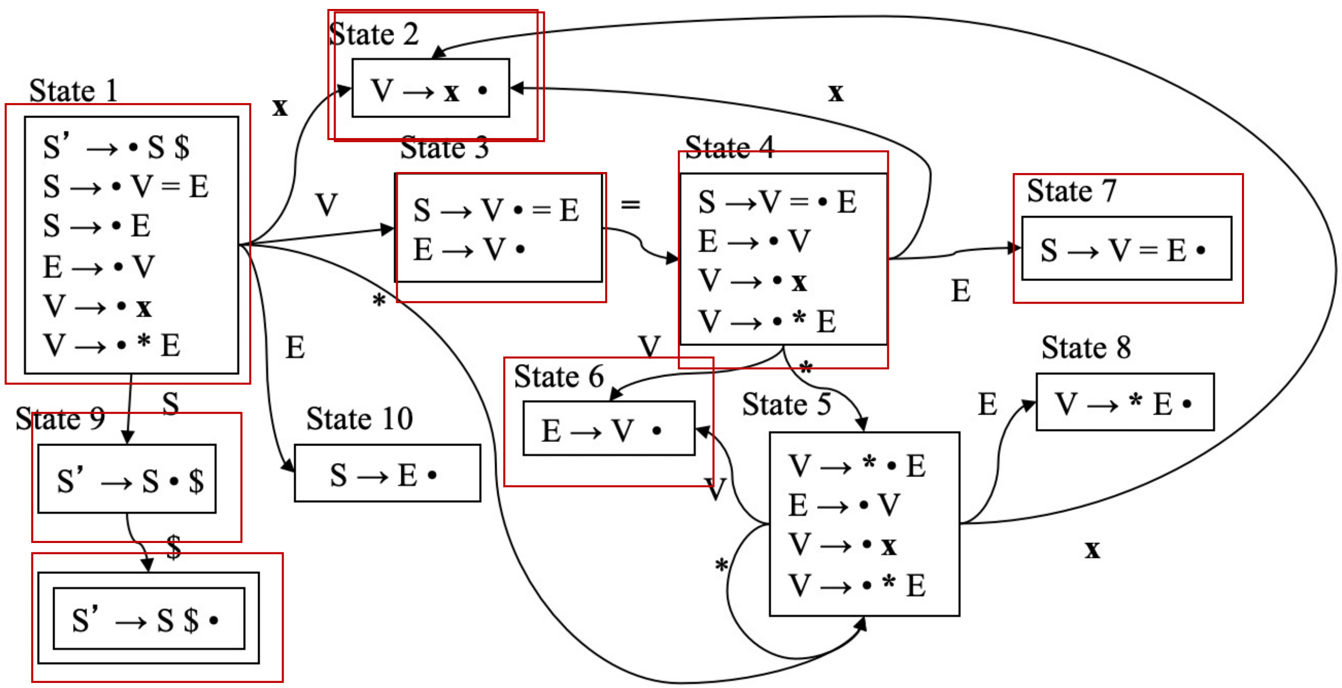
Key: we have a **stack** to store the history of states during the processing.
LR(0) parser = Stack + DFA

Example: States of an LR(0) parser

Input: x = x $

example

0 S' → S $
1 S → V = E
2 S → E
3 E → V
4 V → **x**
5 V → * E



Key: we have a **stack** to store the history of states during the processing.

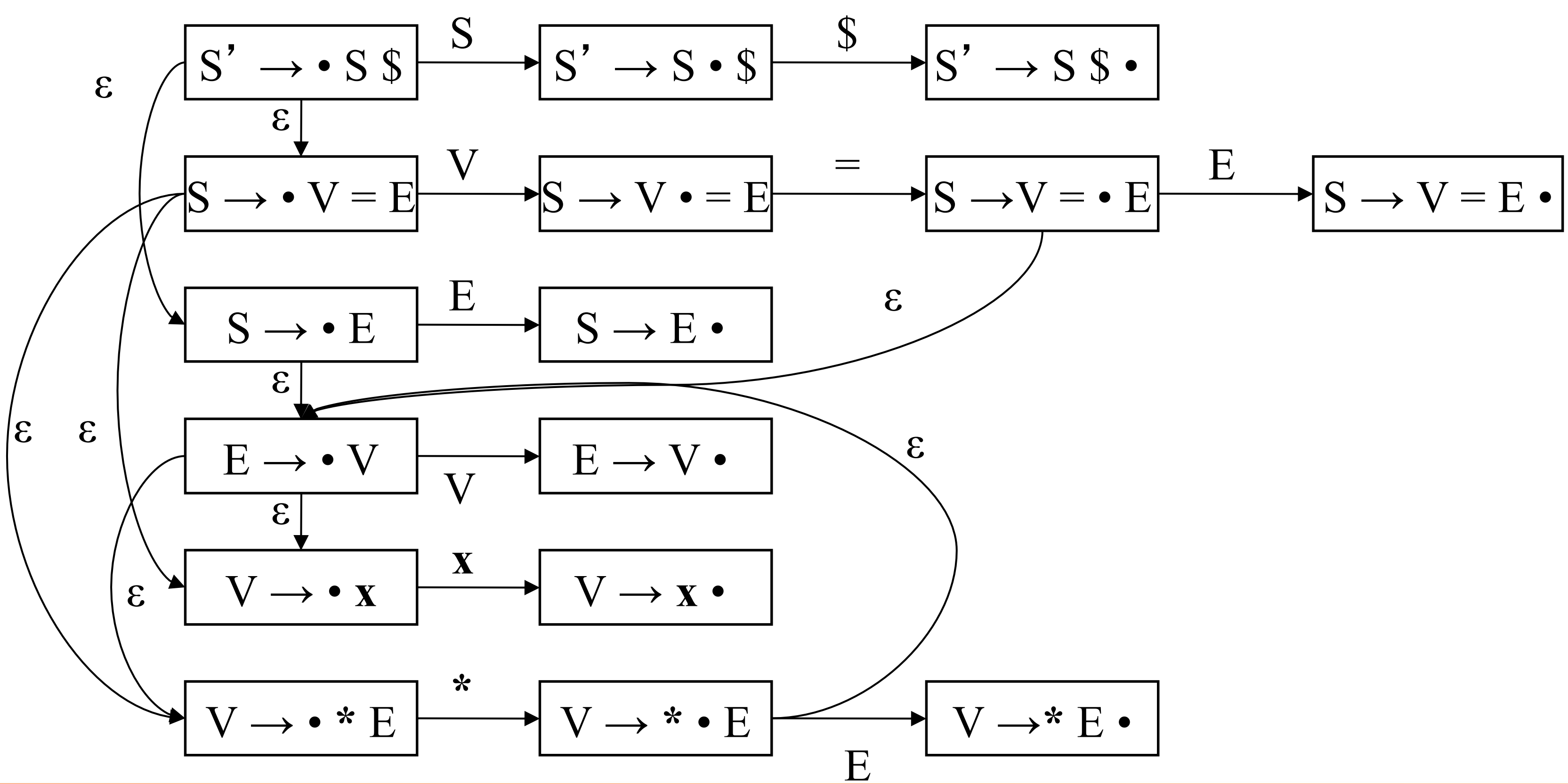
LR(0) parser = Stack + DFA

Things behind the automation: NFA to DFA

For example, to match [S → • V = E] we need to first match V. Let us connect the NFA states together with ε-transitions whenever one state needs to make a “subroutine” call to another state.

example

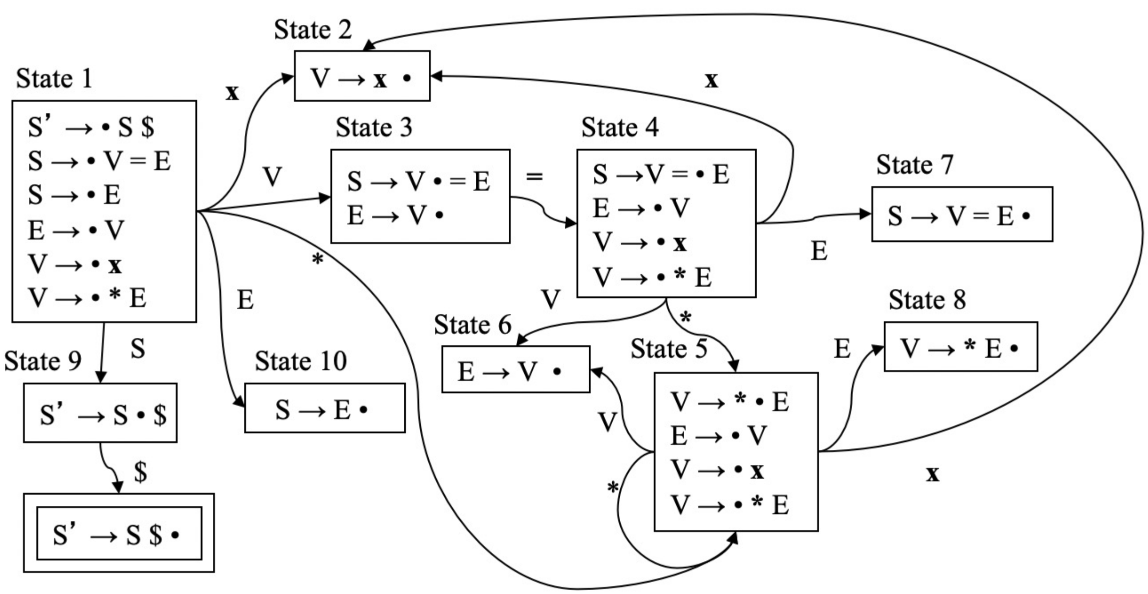
0 S’ → S $
1 S → V = E
2 S → E
3 E → V
4 V → **x**
5 V → * E



Things behind the automation: ACTIO—GOTO Table for LR(0)

example

0 S' → S $
1 S → V = E
2 S → E
3 E → V
4 V → **x**
5 V → * E

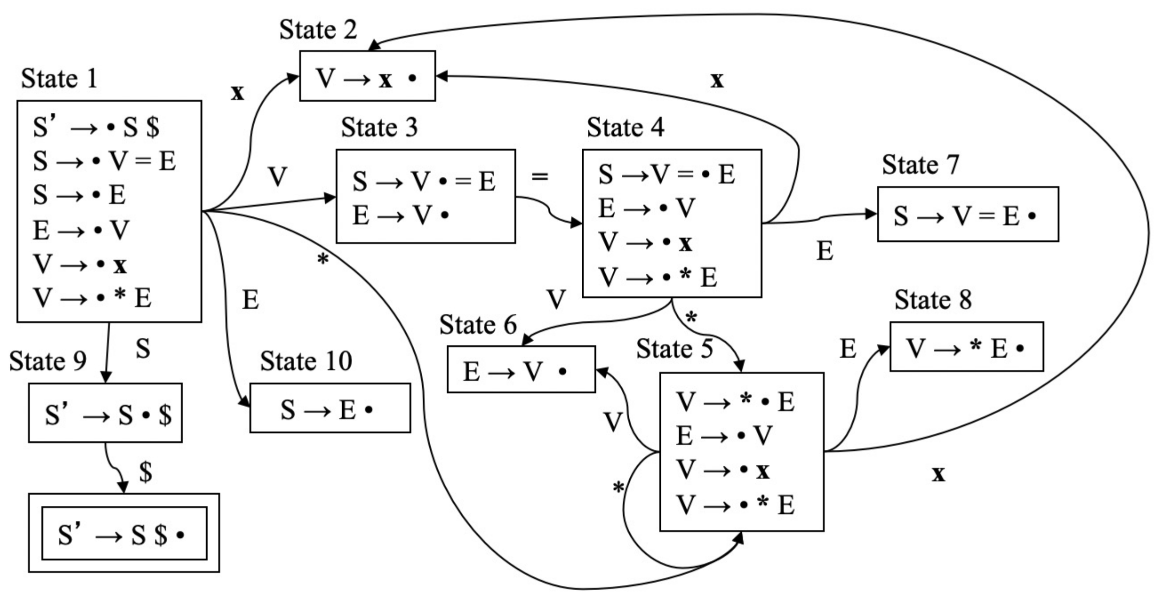


	ACTION				GOTO		
	x	*	=	$	S	E	V
1	S2	S5			9	10	3
2	R4	R4	R4	R4			
3							
4							
5							
6							
7							
8							
9							

Things behind the automation: ACTIO—GOTO Table for LR(0)

example

0 S' → S $
1 S → V = E
2 S → E
3 E → V
4 V → **x**
5 V → * E



	ACTION				GOTO		
	x	*	=	$	S	E	V
1	S2	S5			9	10	3
2	R4	R4	R4	R4			
3	R3	R3	S4/R3	R3			
4	S2	S5				7	6
5	S2	S5				8	6
6	R3	R3	R3	R3			
7	R1	R1	R1	R1			
8		R5		R5			
9				A			

Shift/Reduce conflict will choose to shift

Handle-pruning, Bottom-up Parsers

	ACTION				GOTO		
	x	*	=	$	S	E	V
1	S2	S5			9	10	3
2	R4	R4	R4	R4			
3	R3	R3	S4/R3	R3			
4	S2	S5				7	6
5	S2	S5				8	6
6	R3	R3	R3	R3			
7	R1	R1	R1	R1			
8		R5		R5			
9				A			

example

0 S' → S $
1 S → V = E
2 S → E
3 E → V
4 V → **x**
5 V → * E

Input: x = * x $

Stack:	Input
$ 1	x = * x $
$ 1 x 2	= * x $
$ 1 V 3	= * x $
$ 1 V 3 = 4	* x $
$ 1 V 3 = 4 * 5	x
$ 1 V 3 = 4 * 5 x 2	$
$ 1 V 3 = 4 * 5 V 6	$
$ 1 V 3 = 4 * 5 E 8	$
$ 1 V 3 = 4 V 6	$
$ 1 V 3 = 4 E 7	$
$ 1 S 9	
Accept!	

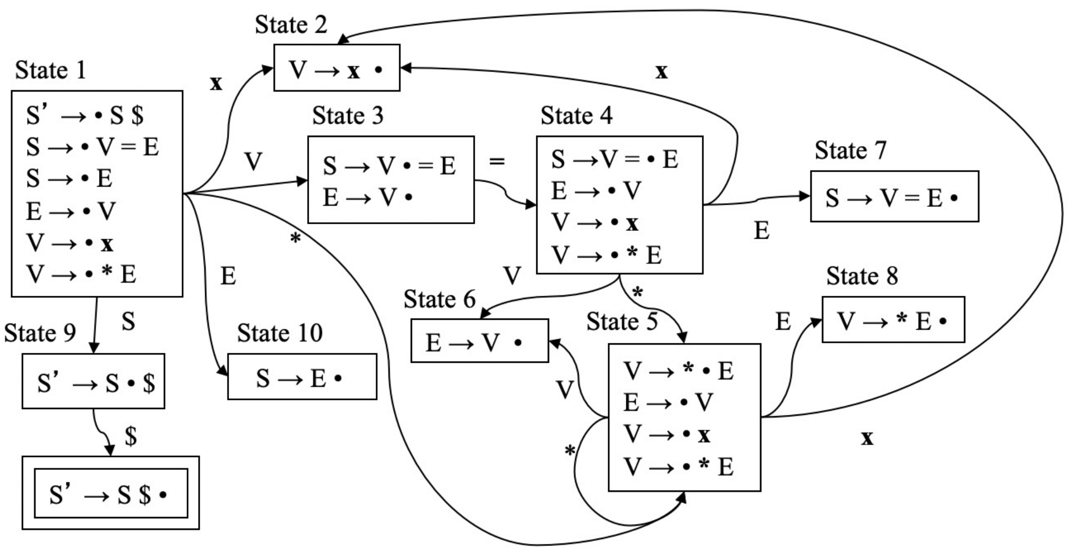
Make, sure that you understand when it is shifting, when it is reducing and where the states come from.

LR(0), SLR(0)

example

0 S' → S $
1 S → V = E
2 S → E
3 E → V
4 V → **x**
5 V → * E

- LR(0), SLR: The same way to build the DFA
- SLR: A smarter way to turn the DFA into the ACTION and GOTO table
  - uses the **non-terminal's FOLLOW set** to help eliminate options and more precisely determine when to reduce.



Follow(V) = {=, $};
Follow(E)={=, $}

	ACTION				GOTO		
	x	*	=	$	S	E	V
1	S2	S5			9	10	3
2	**R4**	**R4**	R4	R4			
3	**R3**	**R3**	S4/R3	R3			
4	S2	S5				7	6
5	S2	S5				8	6
6	**R3**	**R3**	R3	R3			
7				R1			
8		R5		R5			
9				**A**			

Everything in **blue** can be removed

LR(1) Parsing

- A more sophisticated way to use lookahead. More specifically,
  - if I know I am going to try and match the α in $k \rightarrow \bullet \alpha J$, then I know before I start that process that when I am done with α it should be **followed by something from J**.
  - if I know I am going to try and match the α in $k \rightarrow \bullet \alpha T$, then I know before I start that process that when I am done with α it should be **followed by something from T**.
  - Even through both are about reduction for α, they should be treated differently. Just use follow(α) is too rough.

- As such, we for LR(1) we are going to extend the items to a tuple, and keep that information through the process from $k \rightarrow \bullet \alpha J$ to [$\alpha \rightarrow \bullet \beta\gamma\delta$, FIRST(J)], [$\alpha \rightarrow \beta \bullet \gamma\delta$, FIRST(J)], [$\alpha \rightarrow \beta\gamma \bullet \delta$, FIRST(J)], and [$\alpha \rightarrow \beta\gamma\delta \bullet$, FIRST(J)]

LR(1) Items

What's the point of all these look-ahead symbols?

- Has no direct use in [$\alpha \rightarrow \beta\gamma \bullet \delta$, a]
- In [$\alpha \rightarrow \beta\gamma\delta \bullet$, **a**], a look-ahead of **a** implies a reduction by $\alpha \rightarrow \beta\gamma\delta$
- For a state with { [$\alpha \rightarrow \gamma \bullet$, **a**],[$\beta \rightarrow \gamma \bullet \delta$, **b**] }

  lookahead = **a** $\Rightarrow$ ***reduce*** to α

  lookahead $\in$ FIRST(δ) $\Rightarrow$ ***shift***

More items → large number of states in the DFA → larger table

[$\alpha \rightarrow \bullet\beta\gamma\delta$, **a**] and [$\alpha \rightarrow \bullet\beta\gamma\delta$, b] are different items.

Example: Computing I_0 for LR(1)

0	*S'*	$\rightarrow$	*S*
1	*S*	$\rightarrow$	*AA*
2	*A*	$\rightarrow$	*xA*
3		|	*y*

Initial step builds the item [S' $\rightarrow$ • *S* ,$] and takes its *closure( )*

Closure([S' $\rightarrow$ • S ,$])

Item	*From*
[*S'* $\rightarrow$ • *S* , $]	Original item
[*S* $\rightarrow$ • *A A* , $]	first ($) = $
[*A* $\rightarrow$ • x A , x/y]	first(A$) = x/y
[*A* $\rightarrow$ • *y*, x/y]	first (A$) = x/y

So, initial state I_0 is

[*S'* $\rightarrow$ • *S* , $]
[*S* $\rightarrow$ • *A A* , $]
[*A* $\rightarrow$ • x A , x/y]
[*A* $\rightarrow$ • *y*, x/y]

Compute Other States from I_0

0	*S'*	→	*S*
1	*S*	→	*AA*
2	*A*	→	*xA*
3		|	*y*

I_0 is

$[S' \to \bullet\, S\ ,\ \$]$
$[S \to \bullet\, A\, A\ ,\ \$]$
$[A \to \bullet\, x\, A\ ,\ x/y]$
$[A \to \bullet\, y,\ x/y]$

Each state corresponds to a set of LR(1) items

goto(I_0 , A)

- Loop produces

$$[S \to A \bullet A\ ,\ \$]$$

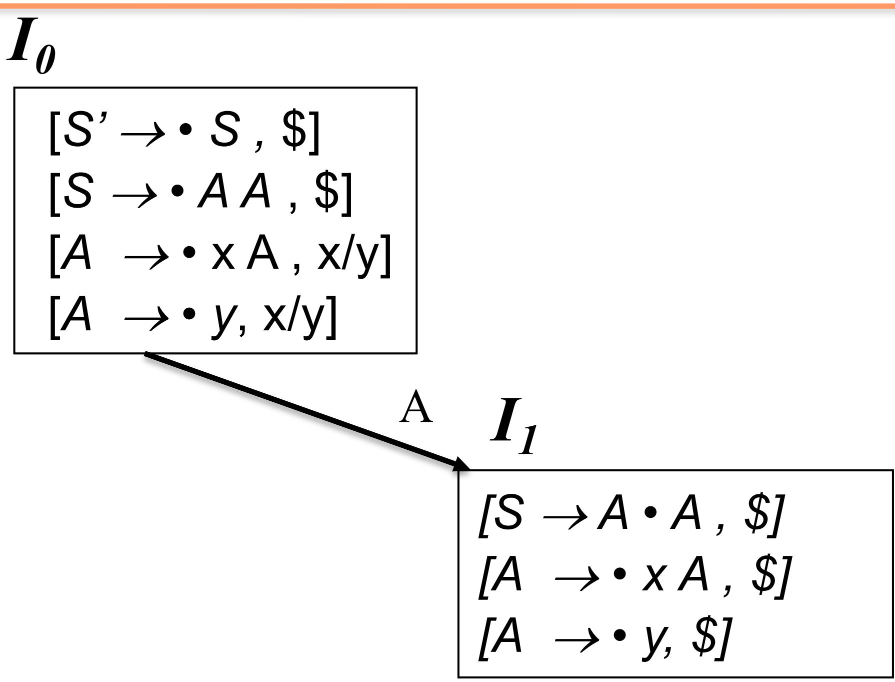
- Apply closure, and it produces s_1

$[S \to A \bullet A\ ,\ \$]$
$[A \to \bullet\, x\, A\ ,\ \$]$
$[A \to \bullet\, y,\ \$]$

Note the difference between $[A \to \bullet\, x\, A\ ,\ \$]$ **and** $[A \to \bullet\, x\, A\ ,\ x/y]$ in I_0

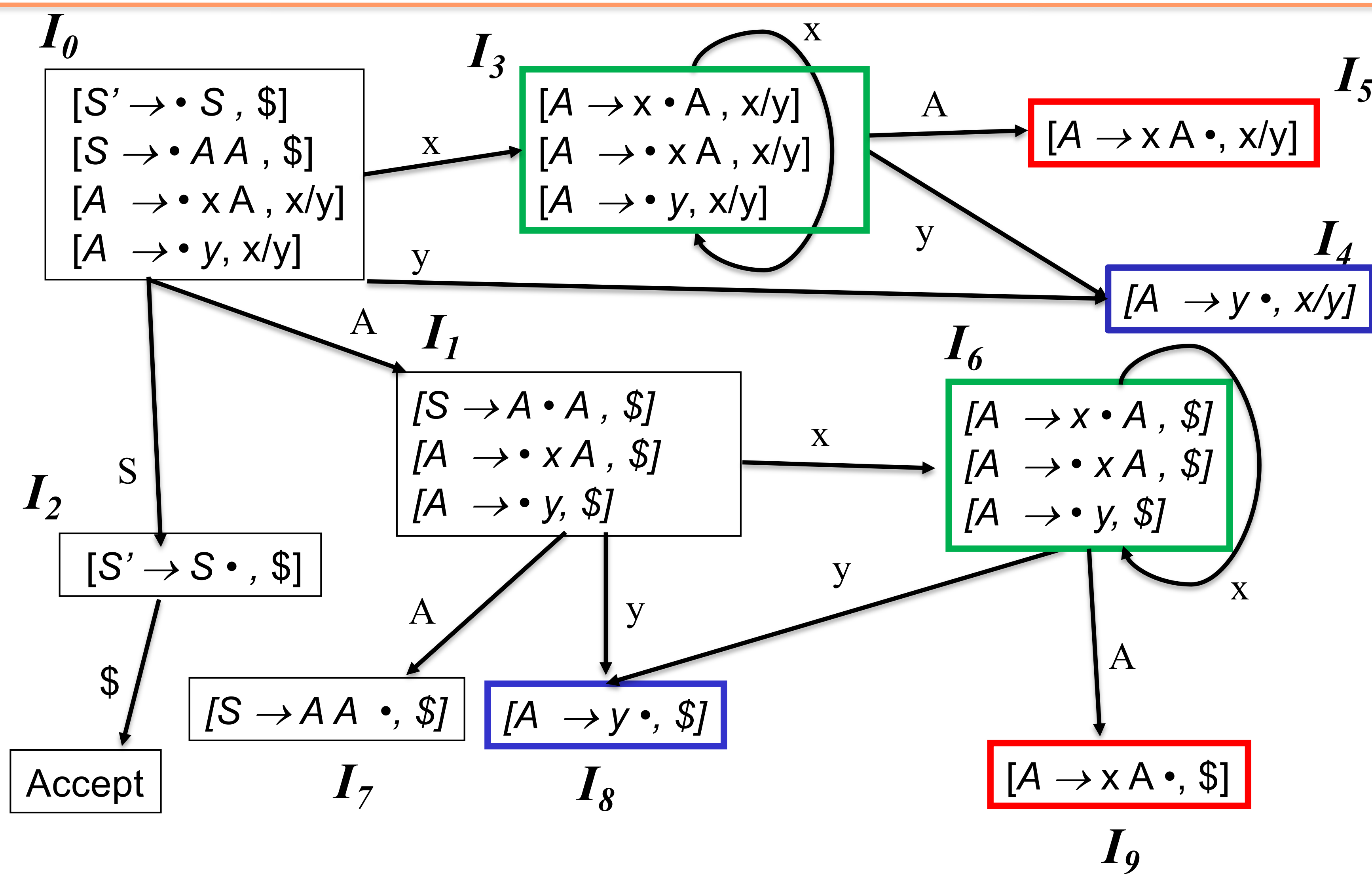
Example Grammar

0	*S'*	$\rightarrow$	*S*
1	*S*	$\rightarrow$	*AA*
2	*A*	$\rightarrow$	*xA*
3		|	*y*



Example Grammar

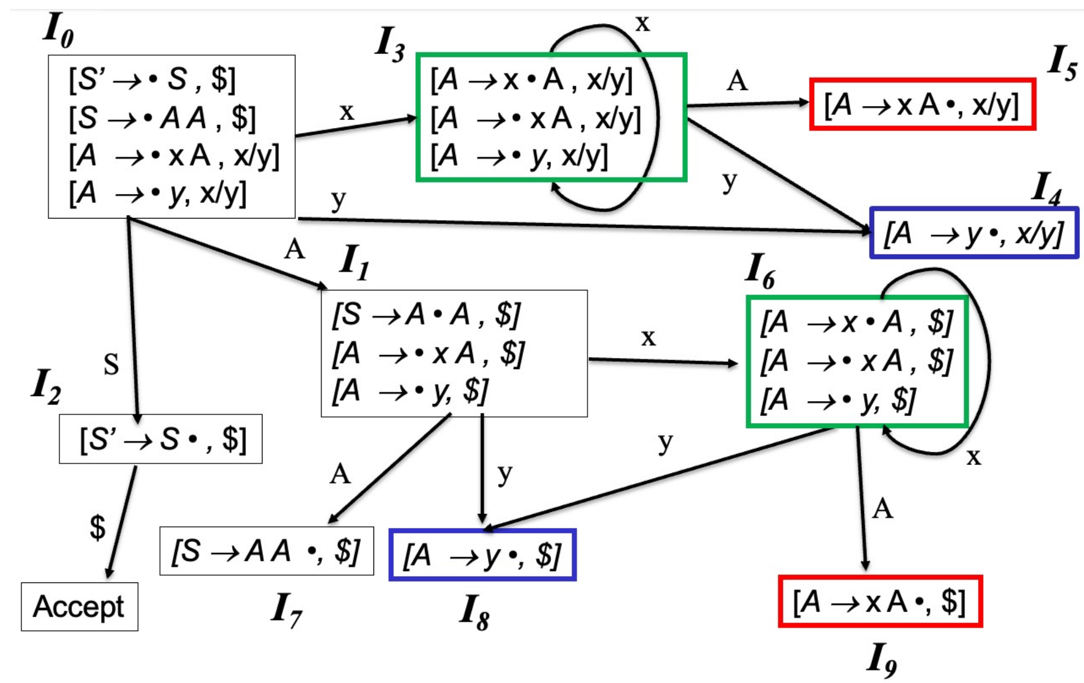
0	S'	$\rightarrow$	S
1	S	$\rightarrow$	AA
2	A	$\rightarrow$	xA
3		|	y



Example (Constructing the LR(1) tables)

The algorithm produces the following table

	ACTION			GOTO	
	x	y	$	*S*	*A*
0	s 3	s4		1	2
1			acc		
2	s6	s8			7
3	s3	s4			5
4	r3	r3			
5	r2	r2			
6	s6	s8			9
7			r1		9
8			r4		
9			r2		



0	*S'*	→	*S $*
1	*S*	→	*AA*
2	*A*	→	*xA*
3		|	*y*

Follow(A) = {$, x, y}

Parsing Example yyy

0	S'	→	S
1	S	→	AA
2	A	→	xA
3		|	y

Stack	Input	Action
0	yyy$	S4
0 y 4	yy$	R3, G2
0 A 2	yy$	S8
0 A 2 y 8	y$	Error

	ACTION			GOTO	
	x	y	$	*S*	*A*
0	s 3	s4		1	2
1			acc		
2	s6	s8			7
3	s3	s4			5
4	r3	r3			
5	r2	r2			
6	s6	s8			9
7			r1		9
8			r4		
9			r2		

Automation: High-level overview of LR(1)

High-level overview

1. Build the handle recognizing DFA (aka *Canonical Collection* of sets of LR(1) items), $C = \{ I_0 , I_1 , \ldots , I_n \}$
   a) Introduce a new start symbol S' which has only one production

      $S' \rightarrow S$

   b) Initial state, I_0 should include
      - $[S' \rightarrow \bullet S, \$]$, along with any equivalent items
      - Derive equivalent items as *closure*(I_0)

   c) Repeatedly compute, for each I_k , and each grammar symbol α, *goto*(I_k , α)
      - If the set is not already in the collection, add it
      - Record all the transitions created by *goto*( )

      This eventually reaches a fixed point

2. Fill in the ACTION and GOTO tables using the DFA

Automation: Overview of Algorithms

Constructing the DFA

I_0 = closure([$S' \rightarrow \bullet\, S$, \$])
C = { I_0 }
while (C is still changing)
  for each $I_i \in C$ and for each $x \in (T \cup NT)$
    I_{new} = goto(I_i, x)
    if $I_{new} \notin C$ then
      $C = C \cup I_{new}$
      record transition $I_i \rightarrow I_{new}$ on x

Computing goto for set of LR(1) items:

Goto(I, x)
  new = Ø
  for each [$\alpha \rightarrow \beta \bullet x\, \delta$, a] $\in I$
    new = new $\cup$ [$\alpha \rightarrow \beta x \bullet \delta$, a]
  return closure(new)

Computing closure of set of LR(1) items:

Closure(I)
  while (I is still changing)
    for each item [$\alpha \rightarrow \beta \bullet \gamma\delta$, a] $\in I$
      for each production $\gamma \rightarrow \tau \in P$
        for each terminal b $\in$ FIRST(δa)
          if [$\gamma \rightarrow \bullet\, \tau$, b] $\notin I$
            then add [$\gamma \rightarrow \bullet\, \tau$, b] to I

- Use the DFA handle recognizing
- Uses Goto to compute the transitions of the DFA
- Uses Closure to compute the states of the DFA

A close look at Closure Computation for LR(1) states

closure(I) adds all the items implied by items already in *I*

- Any item $[\alpha \rightarrow \beta \bullet A\delta , a]$ implies $[A \rightarrow \bullet \ \tau , x]$ for each production with A on the *lhs,* and $x \in$ FIRST(δa)
- Since A is valid, any way to derive A is valid, too
- FIRST(δa) tells us the set of things that could possibly come *after* this particular use of A (and would tell us the production to use)

The algorithm

Fixpoint computation

Closure(*I*)
while (*I* is still changing)
for each item $[\alpha \rightarrow \beta \bullet \gamma\delta , a] \in I$
for each production $\gamma \rightarrow \tau \in P$
for each terminal b $\in$ FIRST(δa)
if $[\gamma \rightarrow \bullet \ \tau , b] \notin I$
then add $[\gamma \rightarrow \bullet \ \tau , b]$ to *I*

- We use Closure to build the states of the handle recognizing DFA
- Closure determines which LR(1) items should be in a state

Constructing the ACTION and GOTO Tables

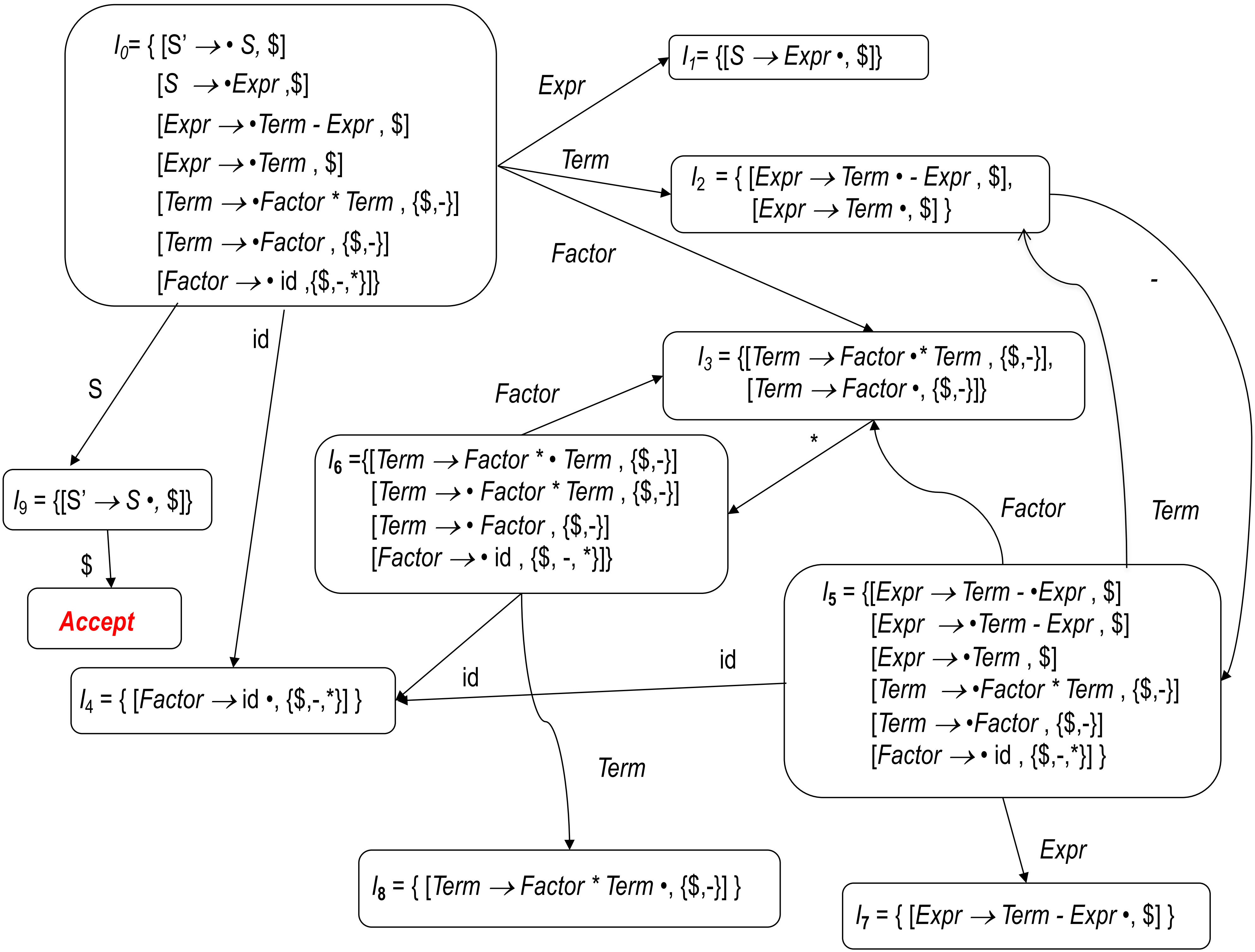
The algorithm

```
for each set of items I_x ∈ C
   for each  item ∈ I_x
      if  item is [α→β •aγ,b] and a ∈ T and goto(I_x,a) = I_k ,
         then ACTION[x,a] ← "shift k"
      else if  item is [S'→S •,$]
         then ACTION[x ,$] ← "accept"
      else if  item is [α→β •,a]
          then ACTION[x,a] ← "reduce α→β"
   for each n ∈ NT
      if  goto(I_x ,n) = I_k
        then GOTO[x,n] ← k
```

Another Example

0	***S' → S***
1	***S → Expr***
2	***Expr → Term - Expr***
3	***Expr → Term***
4	***Term → Factor * Term***
5	***Term → Factor***
6	***Factor →* id**

I_0= { [S' → • *S*, $]
[*S* → •*Expr* ,$]
[*Expr* → •*Term* - *Expr* , $]
[*Expr* → •*Term* , $]
[*Term* → •*Factor* * *Term* , {$,-}]
[*Term* → •*Factor* , {$,-}]
[*Factor* → • id ,{$,-,*}]}



I0= { [S' → • S, $]
[S → •Expr ,$]
[Expr → •Term - Expr , $]
[Expr → •Term , $]
[Term → •Factor * Term , {$,-}]
[Term → •Factor , {$,-}]
[Factor → • id ,{$,-,*}]}
I1= {[S → Expr •, $]}
I2 = { [Expr → Term • - Expr , $],
[Expr → Term •, $] }
I3 = {[Term → Factor •* Term , {$,-}],
[Term → Factor •, {$,-}]}
I6 ={[Term → Factor * • Term , {$,-}]
[Term → • Factor * Term , {$,-}]
[Term → • Factor , {$,-}]
[Factor → • id , {$, -, *}]}
I9 = {[S' → S •, $]}
Accept
I4 = { [Factor → id •, {$,-,*}] }
I5 = {[Expr → Term - •Expr , $]
[Expr → •Term - Expr , $]
[Expr → •Term , $]
[Term → •Factor * Term , {$,-}]
[Term → •Factor , {$,-}]
[Factor → • id , {$,-,*}] }
I8 = { [Term → Factor * Term •, {$,-}] }
I7 = { [Expr → Term - Expr •, $] }
Expr
Term
Factor
S
id
$
Factor
*
-
Factor
Term
id
id
Term
Expr

Example (Constructing the LR(1) tables)

The algorithm produces the following table

	ACTION				GOTO			
	id	-	*	$	*S*	*Expr*	*Term*	*Factor*
0	S4				9	1	2	3
1				R1				
2		S5		R3				
3		R5	S6	R5				
4		R6	R6	R6				
5	S4					7	2	3
6	S4						8	3
7				R2				
8		R4		R4				
9				Acc				

Parsing Example x-z*y

Stack	Input	Action
0	**id** – id * id $	S4
0 id 4	– id * id $	R6, G3
0 F 3	– id * id $	R5, G2
0 T 2	– id * id $	S5
0 T 2 – 5	**id** * id$	S4
0 T 2 – 5 id 4	* id$	R6, G3
0 T 2 – 5 F 3	* id$	S6
0 T 2 – 5 F 3 * 6	**id**$	S4
0 T 2 – 5 F 3 * 6 id 4	**$**	R6,G3
0 T 2 – 5 F 3 * 6 F 3	**$**	R5,G8
0 T 2 – 5 F 3 * 6 T 8	**$**	R4,G2
0 T 2 – 5 T 2	**$**	R3,G7
0 T 2 – 5 E 7	**$**	R2,G1
0 E 1	**$**	R1
0 S 9	**$**	
Accept	**$**	

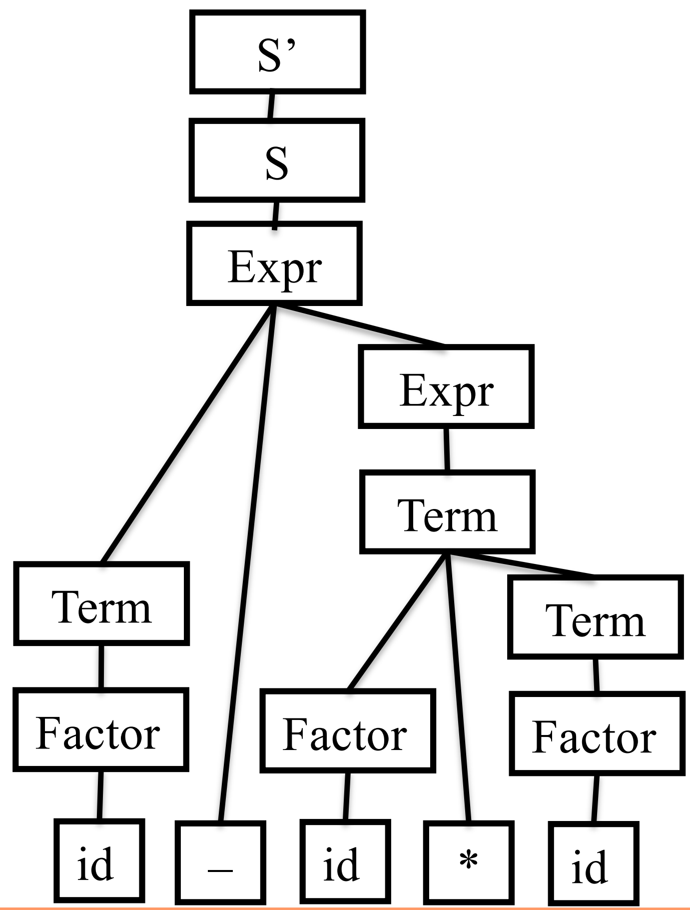
	ACTION				GOTO			
	id	-	*	$	*S*	*Expr*	*Term*	*Factor*
0	s 4				9	1	2	3
1				r 1				
2		s 5		r 3				
3		r 5	s 6	r 5				
4		r 6	r 6	r 6				
5	s 4					7	2	3
6	s 4						8	3
7				r 2				
8		r 4		r 4				
9				Acc				

0 ***S' → S***
1 ***S → Expr***
2 ***Expr → Term - Expr***
3 ***Expr → Term***
4 ***Term → Factor * Term***
5 ***Term → Factor***
6 ***Factor →* id**

Parse tree for x-z*y

0	*S' → S*
1	*S → Expr*
2	*Expr → Term - Expr*
3	*Expr → Term*
4	*Term → Factor * Term*
5	*Term → Factor*
6	*Factor →* id

Stack	Input	Action
0	id – id * id $	S4
0 id 4	– id * id $	**R6**, G3
0 F 3	– id * id $	**R5**, G2
0 T 2	– id * id $	S5
0 T 2 – 5	id * id$	S4
0 T 2 – 5 id 4	* id$	**R6**, G3
0 T 2 – 5 F 3	* id$	S6
0 T 2 – 5 F 3 * 6	id$	S4
0 T 2 – 5 F 3 * 6 id 4	$	**R6**,G3
0 T 2 – 5 F 3 * 6 F 3	$	**R5**,G8
0 T 2 – 5 F 3 * 6 T 8	$	**R4**,G2
0 T 2 – 5 T 2	$	**R3**,G7
0 T 2 – 5 E 7	$	**R2**,G1
0 E 1	$	**R1**
0 S 9	$	
Accept	$	



Parsing Example (x-z-y)

Stack	Input	Action
0	id – id - id $	S4
0 id 4	– id - id $	R6, G3
0 F 3	– id - id $	R5, G2
0 T 2	– id - id $	S5
0 T 2 – 5	id - id$	S4
0 T 2 – 5 id 4	- id$	R6, G3
0 T 2 – 5 F 3	- id$	S5
0 T 2 – 5 F 3 - 5	id$	S4
0 T 2 – 5 F 3 - 5 id 4	$	R6,G3
0 T 2 – 5 F 3 - 5 F 3	$	R5,G2
0 T 2 – 5 F 3 - 5 T 2	$	R3,G7
0 T 2 – 5 F 3 - 5 E 7	$	R2,G7
0 T 2 – 5 E 7	$	R2,G1
0 E 1	$	R1,G9
0 S 9	$	
Accept	$	

	ACTION				GOTO			
	id	-	*	$	*S*	*Expr*	*Term*	*Factor*
0	S4				9	1	2	3
1				R1				
2		S5		R3				
3		R5	S6	R5				
4		R6	R6	R6				
5	S4					7	2	3
6	S4						8	3
7				R2				
8		R4		R4				
9				Acc				

0	*S' → S*
1	*S → Expr*
2	*Expr → Term - Expr*
3	*Expr → Term*
4	*Term → Factor * Term*
5	*Term → Factor*
6	*Factor →* id