Bottom-Up Parsing

Part II

Canonical Collection of LR(0) items

```
CC_LR(0)_I items(G':augmented_grammar){
    C = {CLOSURE({S' → •S})} ;
    repeat{
        foreach(I ∈ C)
            foreach(grammar symbol X)
                if(GOTO(I,X)≠∅ && GOTO(I,X) ∉ C)
                                C = C U {GOTO(I,X)};
    }until(no new sets of items are added to C)
    return C;
}
```

LR(0) automaton

G' : augmented grammar

LR(0) automaton for G'

$\langle \mathbf{Q}, \mathbf{q_0}, \mathbf{GOTO}: Q \times (T_{G'} \cup N_{G'}) \rightarrow Q, \mathbf{F} \rangle$
where:
$\mathbf{Q} = \mathbf{F} =$ items(G'),
$\mathbf{q_0} =$ CLOSURE({S' → •S})

Construction of the LR(0) automaton for the expression grammar:

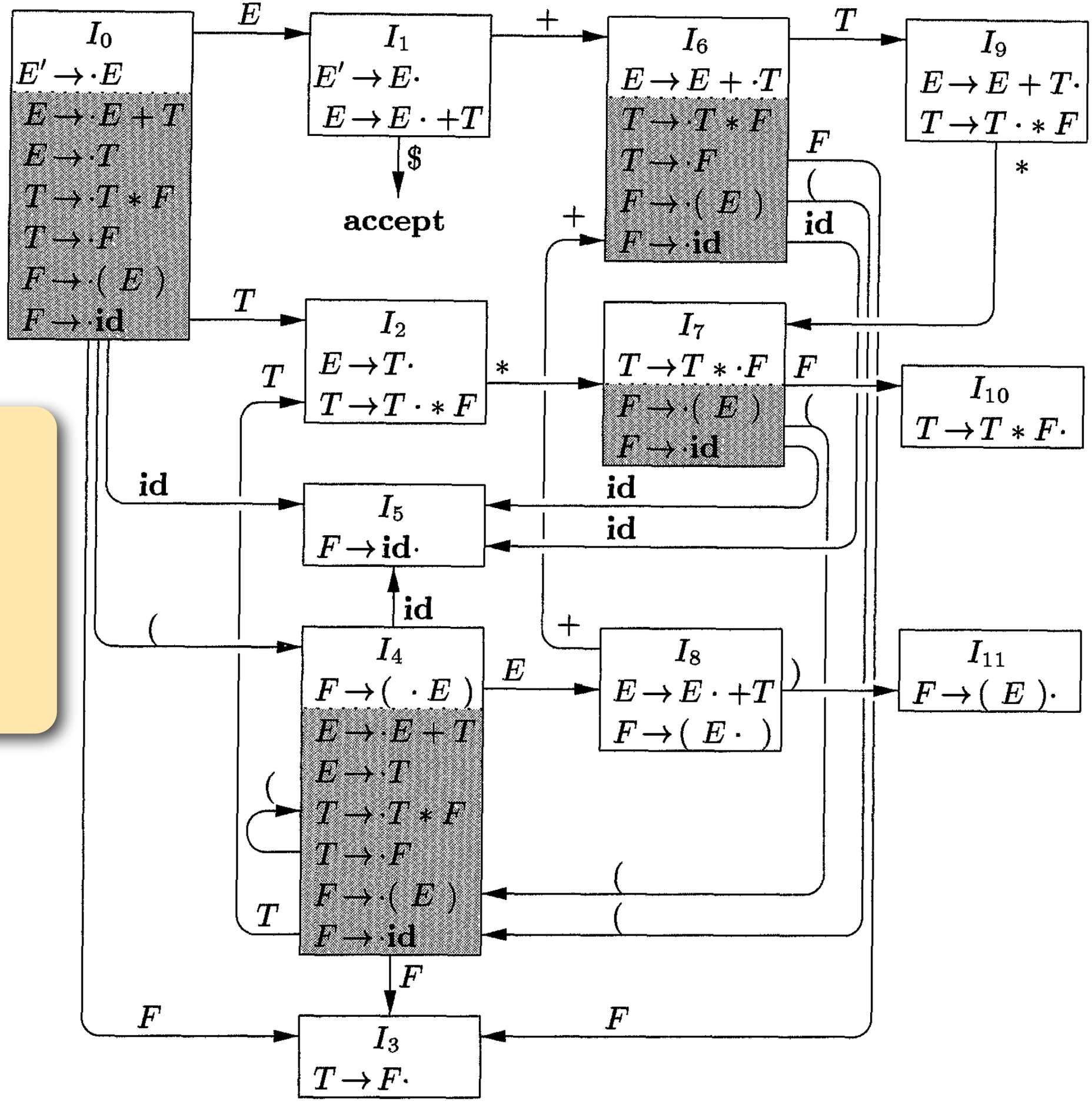
E' → E
E → E + T | T
T → T* F | F
F → (E) | **id**

E’ → E

E → E + T | T

T → T* F | F

F → (E) | **id**

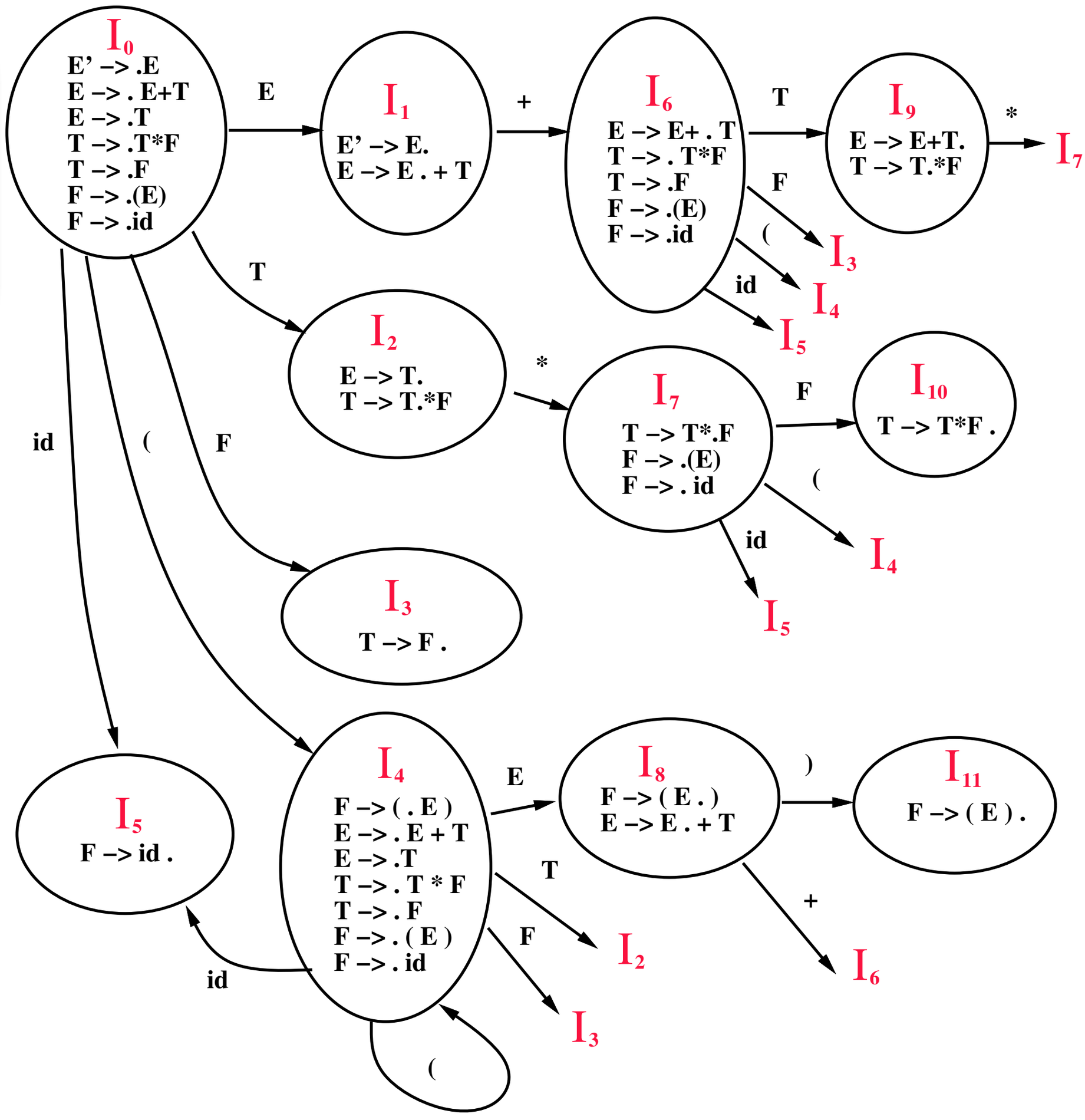


E’ → E

E → E + T | T

T → T* F | F

F → (E) | **id**

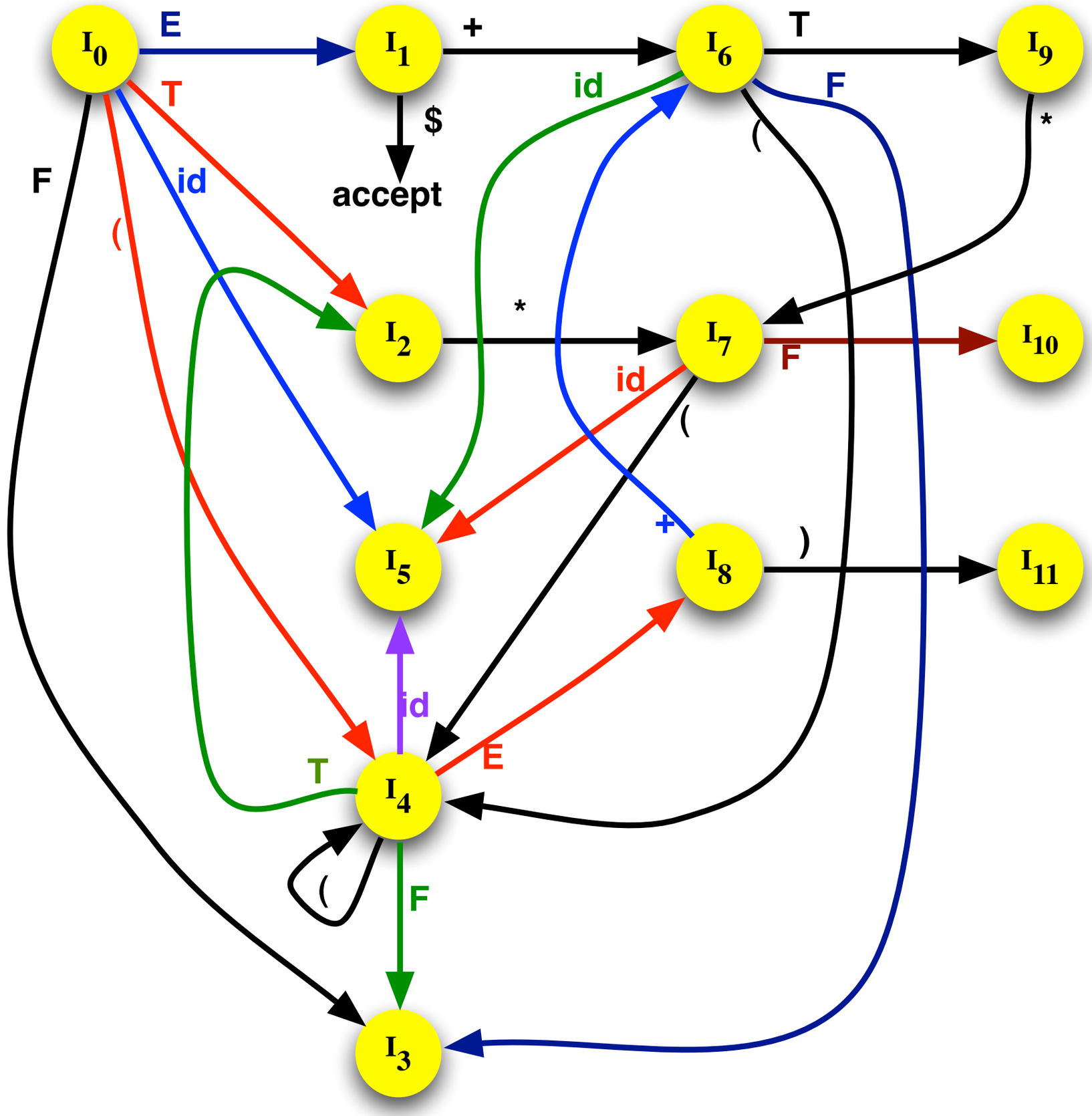


E’ → E

E → E + T | T

T → T* F | F

F → (E) | **id**



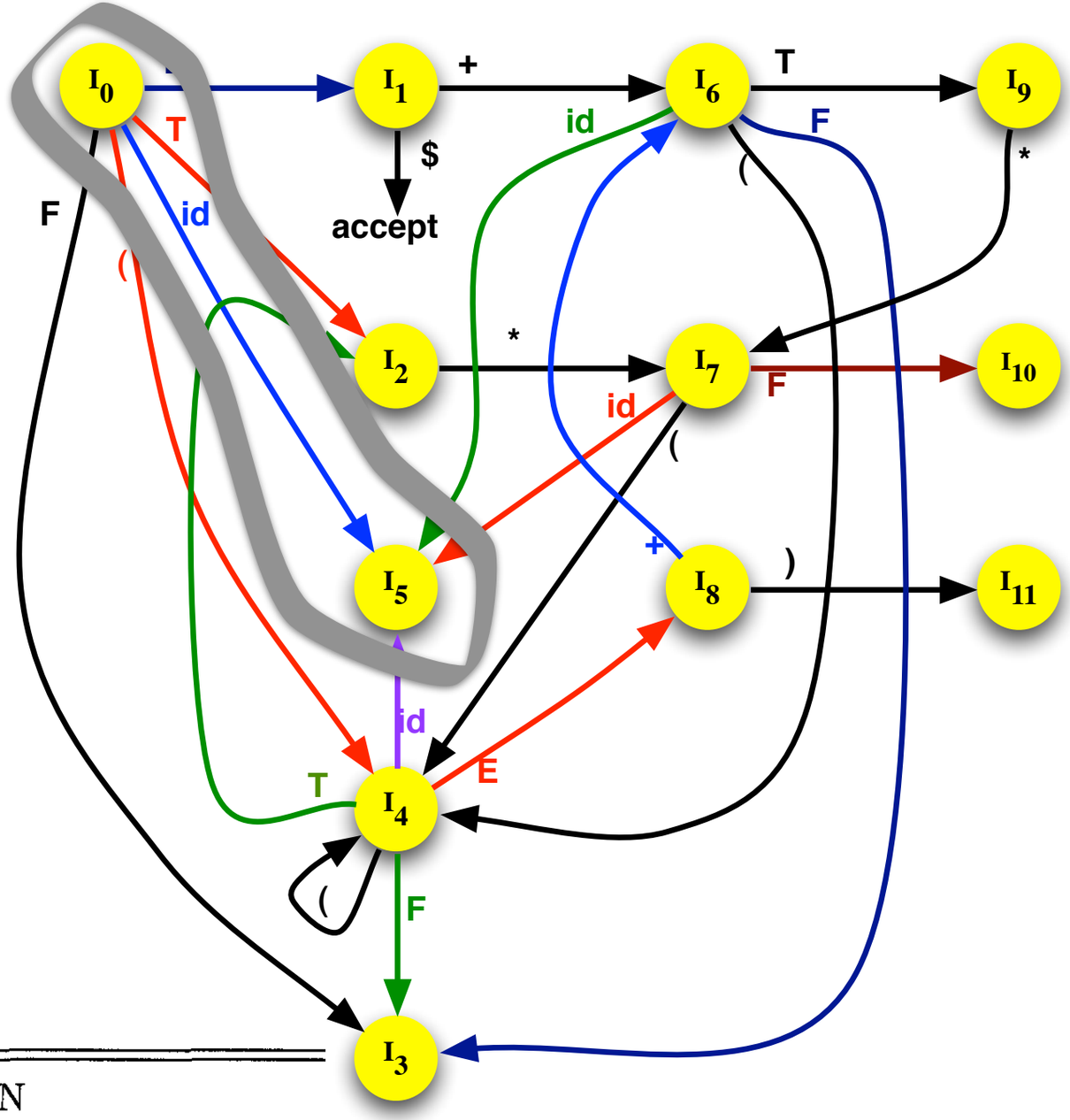
Shift-reduce parsing using LR(0) automaton

```
Push(I0);
repeat{
   //Begin to scan the input from left to right
   Ii = top() and next input symbol a;
   if (Ij = GOTO(Ii,a)) then shift a and Ij;
                        // Push(a)
                        // push(Ij).
   else if (A→β• ∈ Ii) then{
            perform "reduce by A→β";
            go to the state Ij = GOTO(I,A)
               and
            push A and Ij  into the  stack
                //where I is the state on the top_of__stack
                //after removing β and the corresponding states;
   }
    _exit(Reject if none of the above can be done);
    _exit(Report "conflicts" if more than one can be done);
} until EOF is seen
```

E’ → E
E → E + T | T
T → T* F | F
F → (E) | **id**

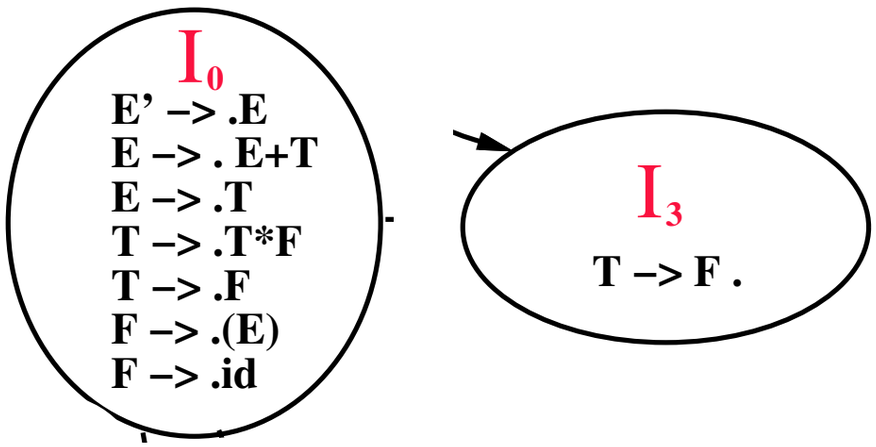
I_0
E’ –> .E
E –> . E+T
E –> .T
T –> .T*F
T –> .F
F –> .(E)
F –> .id

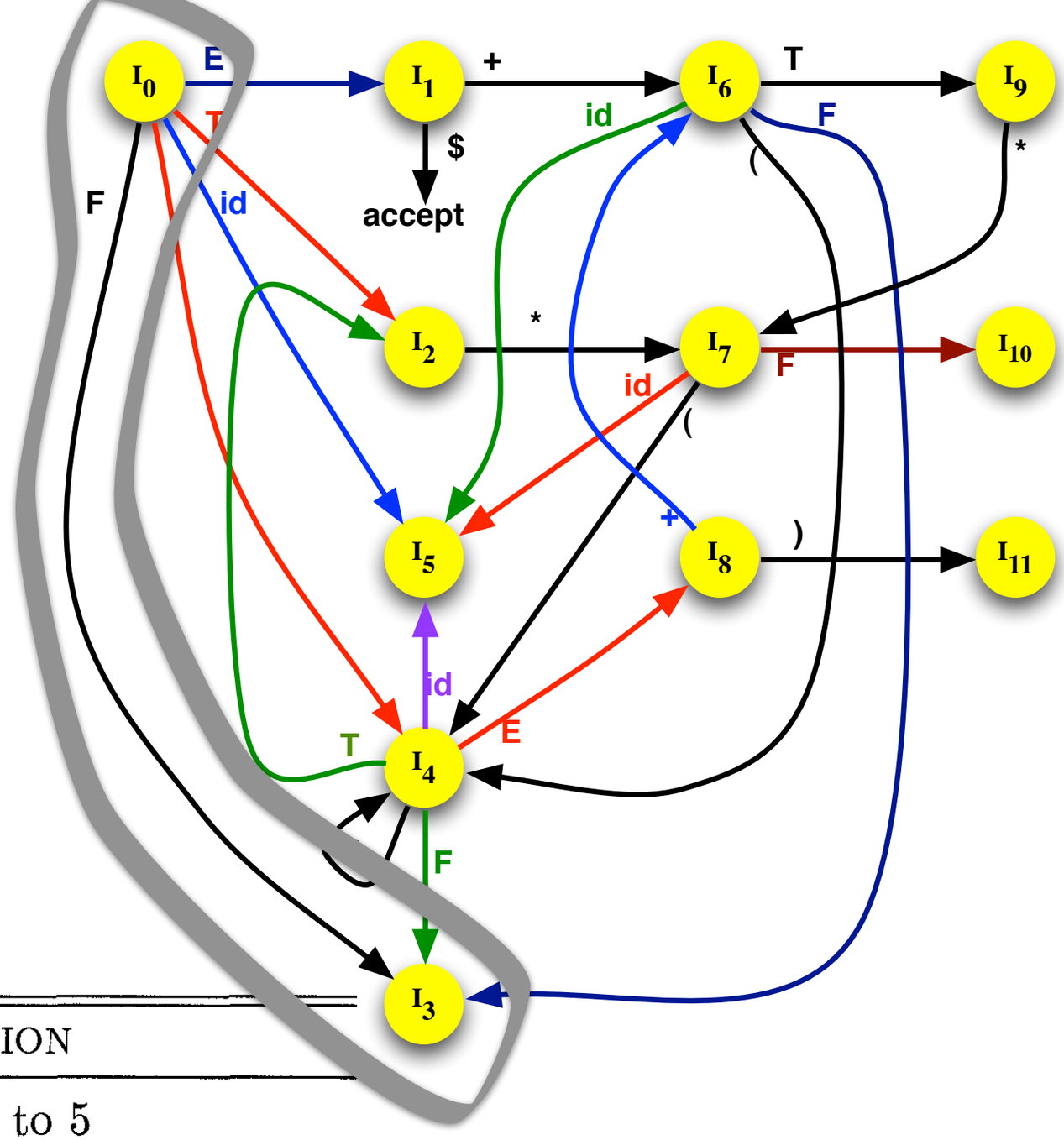
I_5
F –> id .



Line	Stack	Symbols	Input	Action
(1)	0	$	**id** ∗ **id** $	shift to 5
(2)	0 5	$ **id**	∗ **id** $	reduce by $F \rightarrow$ **id**

E’ → E
E → E + T | T
T → T* F | F
F → (E) | **id**



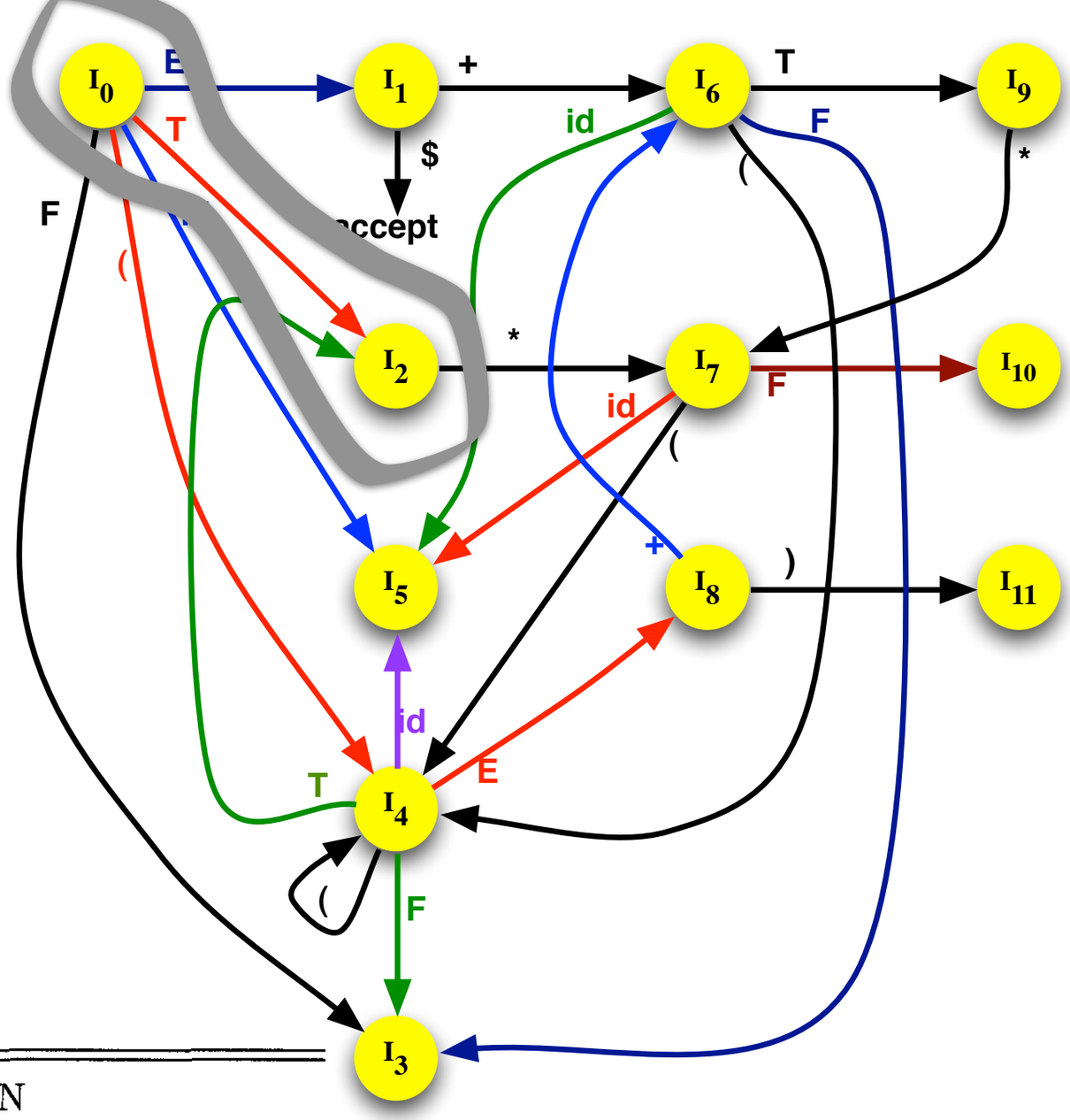


Line	Stack	Symbols	Input	Action
(1)	0	$	**id** ∗ **id** $	shift to 5
(2)	0 5	$ **id**	∗ **id** $	reduce by $F \rightarrow$ **id**
(3)	0 3	$ F	∗ **id** $	reduce by $T \rightarrow F$

E’ → E
E → E + T | T
T → T* F | F
F → (E) | **id**

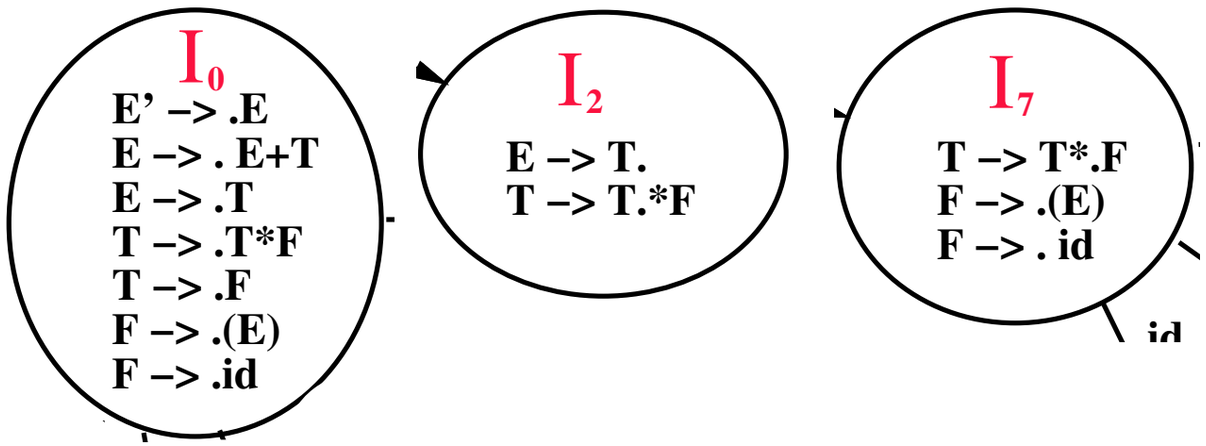
I_0
E’ –> .E
E –> . E+T
E –> .T
T –> .T*F
T –> .F
F –> .(E)
F –> .id

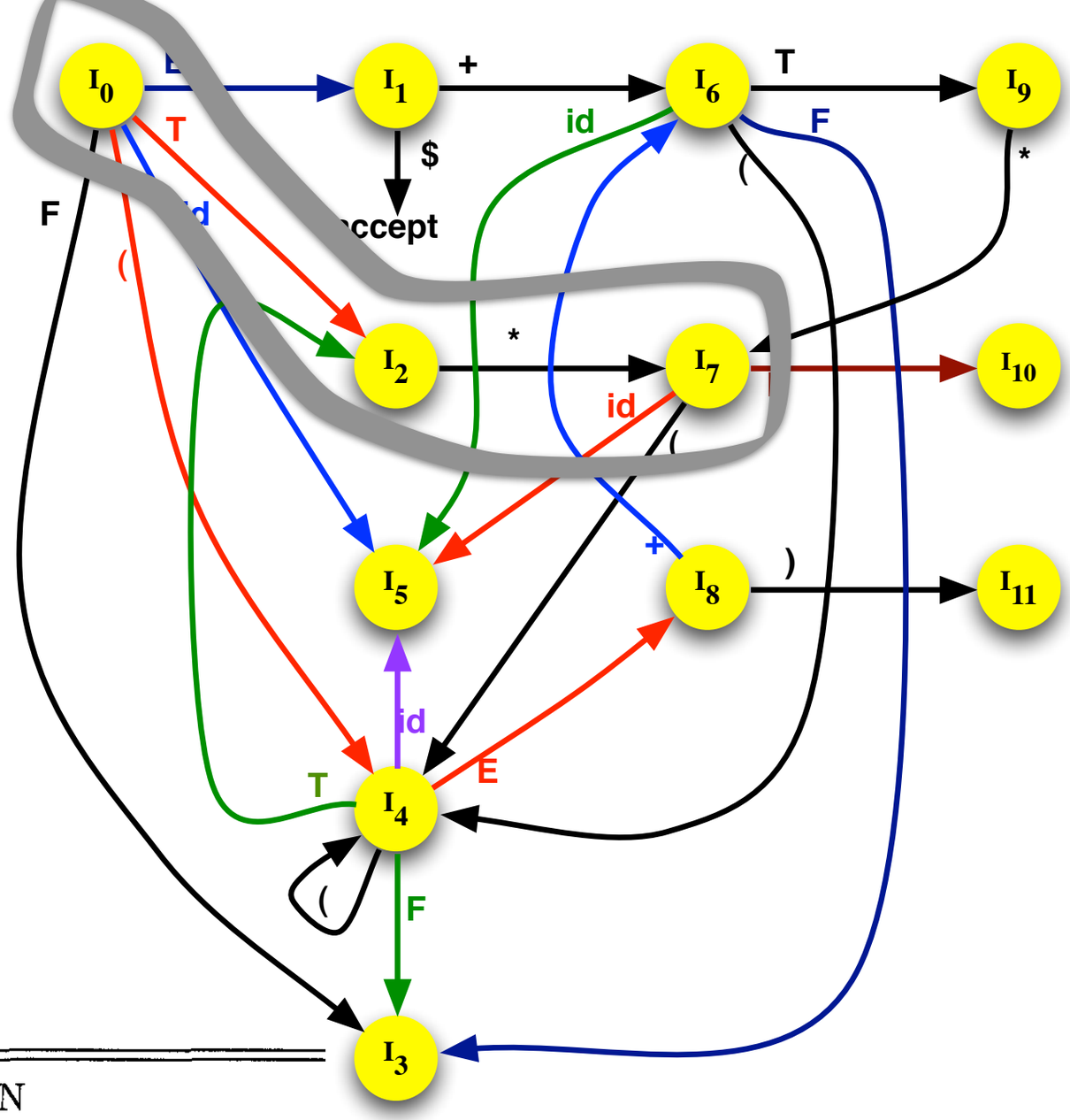
I_2
E –> T.
T –> T.*F



Line	Stack	Symbols	Input	Action
(1)	0	$	**id** ∗ **id** $	shift to 5
(2)	0 5	$ **id**	∗ **id** $	reduce by $F \rightarrow$ **id**
(3)	0 3	$ F	∗ **id** $	reduce by $T \rightarrow F$
(4)	0 2	$ T	∗ **id** $	shift to 7

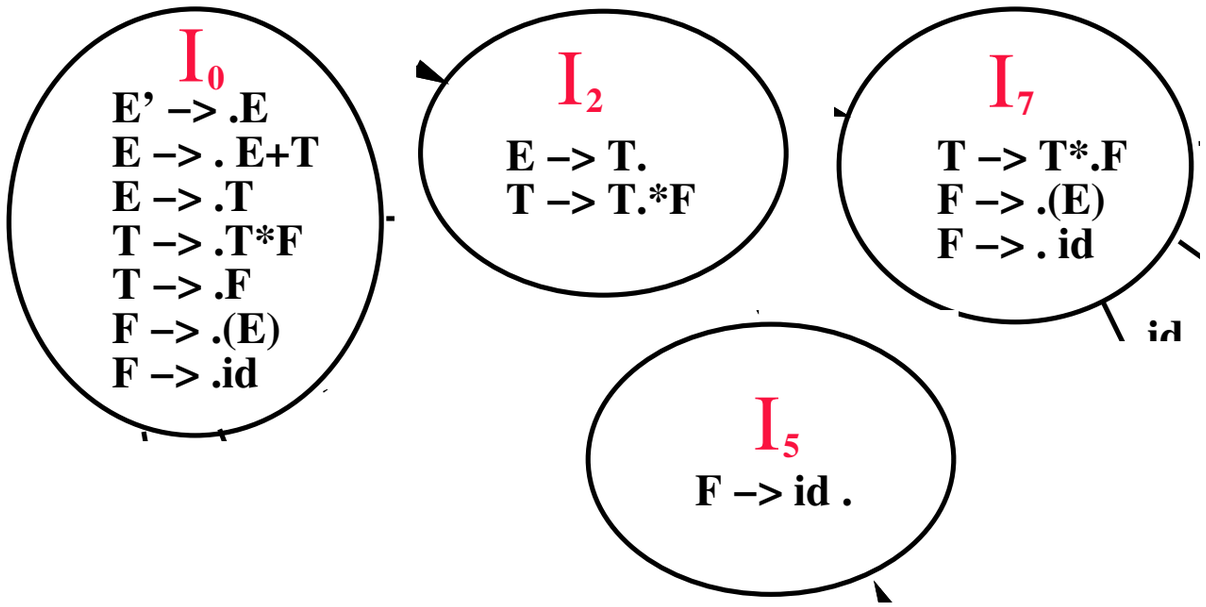
$$
\begin{aligned}
E' &\rightarrow E \\
E &\rightarrow E + T \mid T \\
T &\rightarrow T^* F \mid F \\
F &\rightarrow (E) \mid \mathbf{id}
\end{aligned}
$$

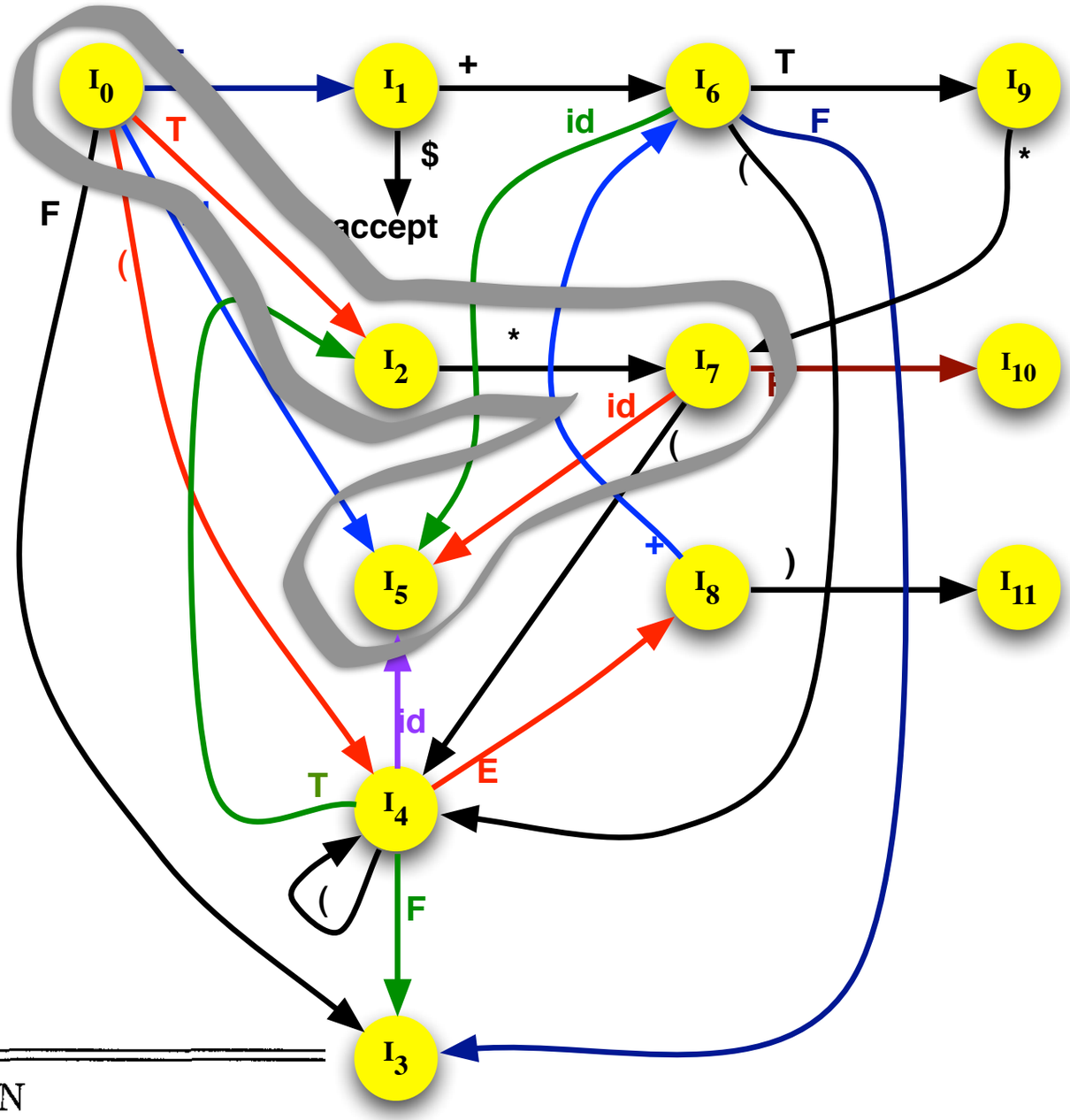




Line	Stack	Symbols	Input	Action
(1)	0	\$	**id** $*$ **id** \$	shift to 5
(2)	0 5	\$ **id**	$*$ **id** \$	reduce by $F \rightarrow$ **id**
(3)	0 3	\$ F	$*$ **id** \$	reduce by $T \rightarrow F$
(4)	0 2	\$ T	$*$ **id** \$	shift to 7
(5)	0 2 7	\$ $T*$	**id** \$	shift to 5

E’ → E
E → E + T | T
T → T* F | F
F → (E) | **id**





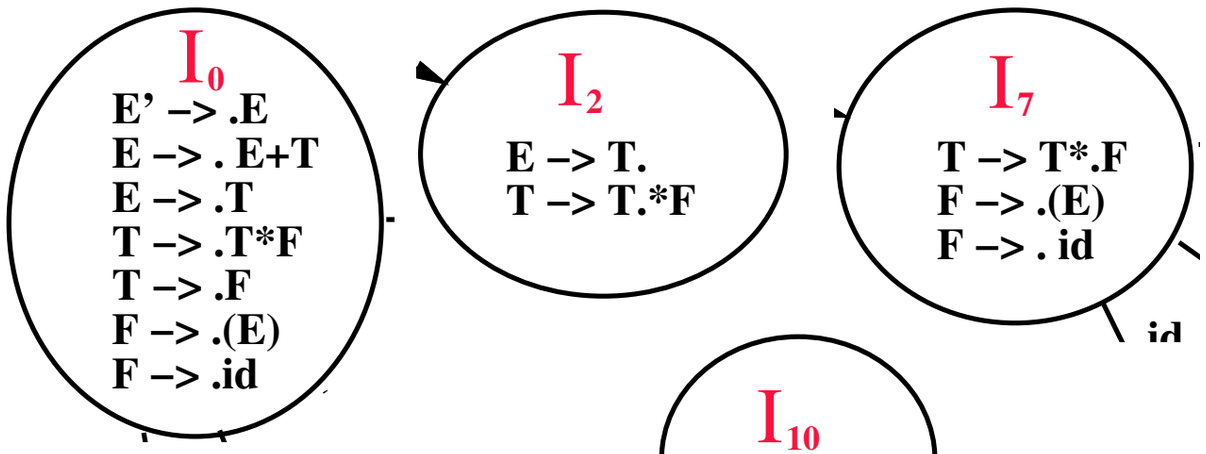
Line	Stack	Symbols	Input	Action
(1)	0	$	**id** ∗ **id** $	shift to 5
(2)	0 5	$ **id**	∗ **id** $	reduce by $F \rightarrow$ **id**
(3)	0 3	$ F	∗ **id** $	reduce by $T \rightarrow F$
(4)	0 2	$ T	∗ **id** $	shift to 7
(5)	0 2 7	$ T ∗	**id** $	shift to 5
(6)	0 2 7 5	$ T ∗ **id**	$	reduce by $F \rightarrow$ **id**

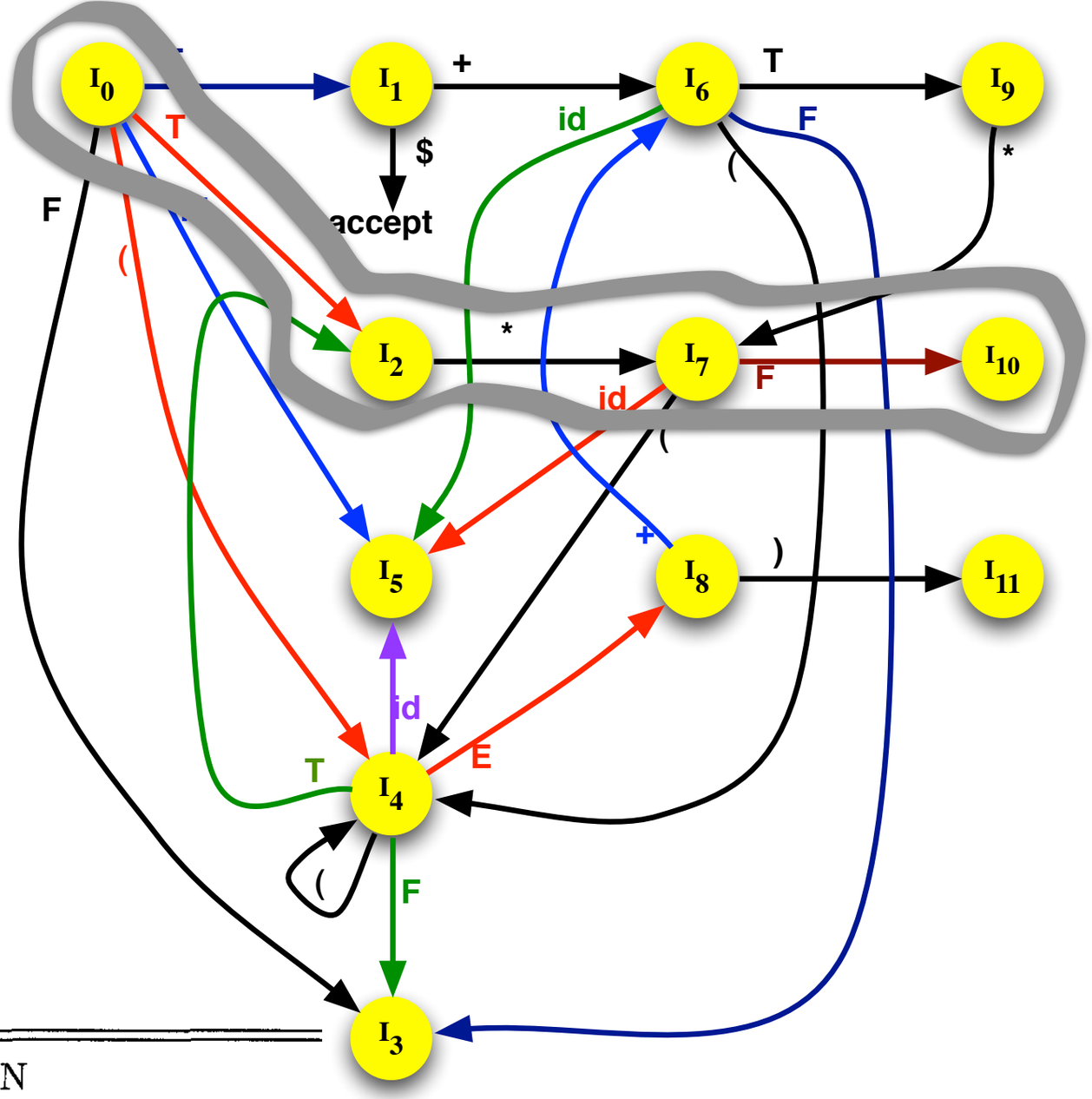
$E' \rightarrow E$

$E \rightarrow E + T \mid T$

$T \rightarrow T^* F \mid F$

$F \rightarrow (E) \mid \mathbf{id}$





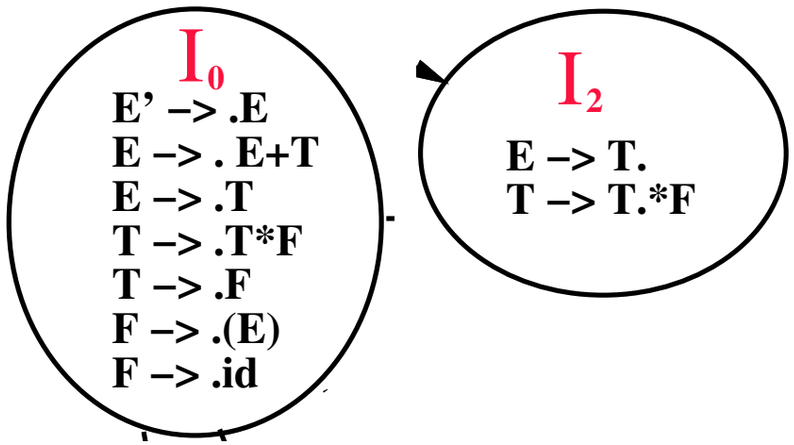
Line	Stack	Symbols	Input	Action
(1)	0	\$	**id** $*$ **id** \$	shift to 5
(2)	0 5	\$ **id**	$*$ **id** \$	reduce by $F \rightarrow$ **id**
(3)	0 3	\$ F	$*$ **id** \$	reduce by $T \rightarrow F$
(4)	0 2	\$ T	$*$ **id** \$	shift to 7
(5)	0 2 7	\$ $T*$	**id** \$	shift to 5
(6)	0 2 7 5	\$ $T*$ **id**	\$	reduce by $F \rightarrow$ **id**
(7)	0 2 7 10	\$ $T*F$	\$	reduce by $T \rightarrow T*F$

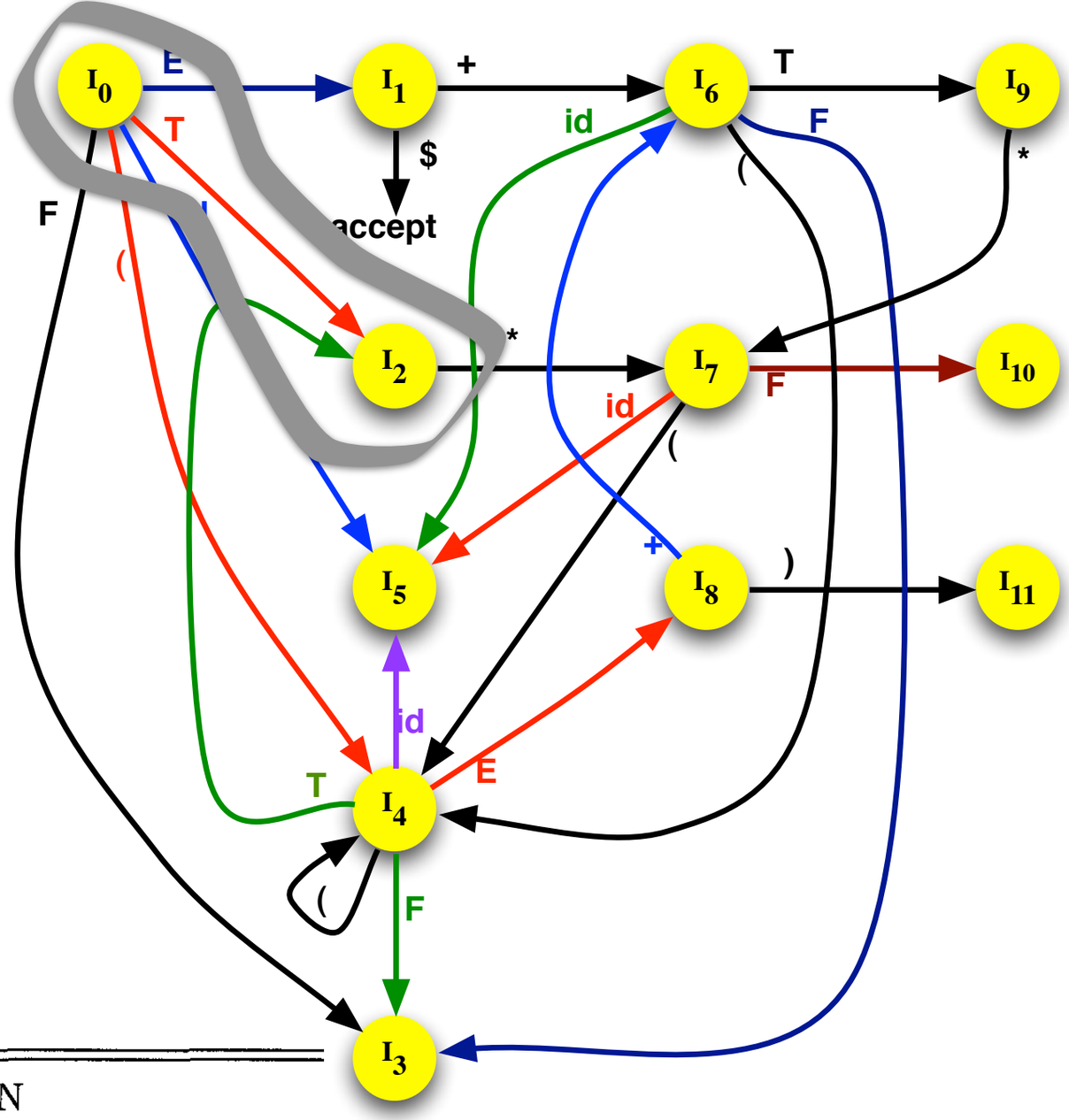
$E' \rightarrow E$

$E \rightarrow E + T \mid T$

$T \rightarrow T^* F \mid F$

$F \rightarrow (E) \mid \mathbf{id}$



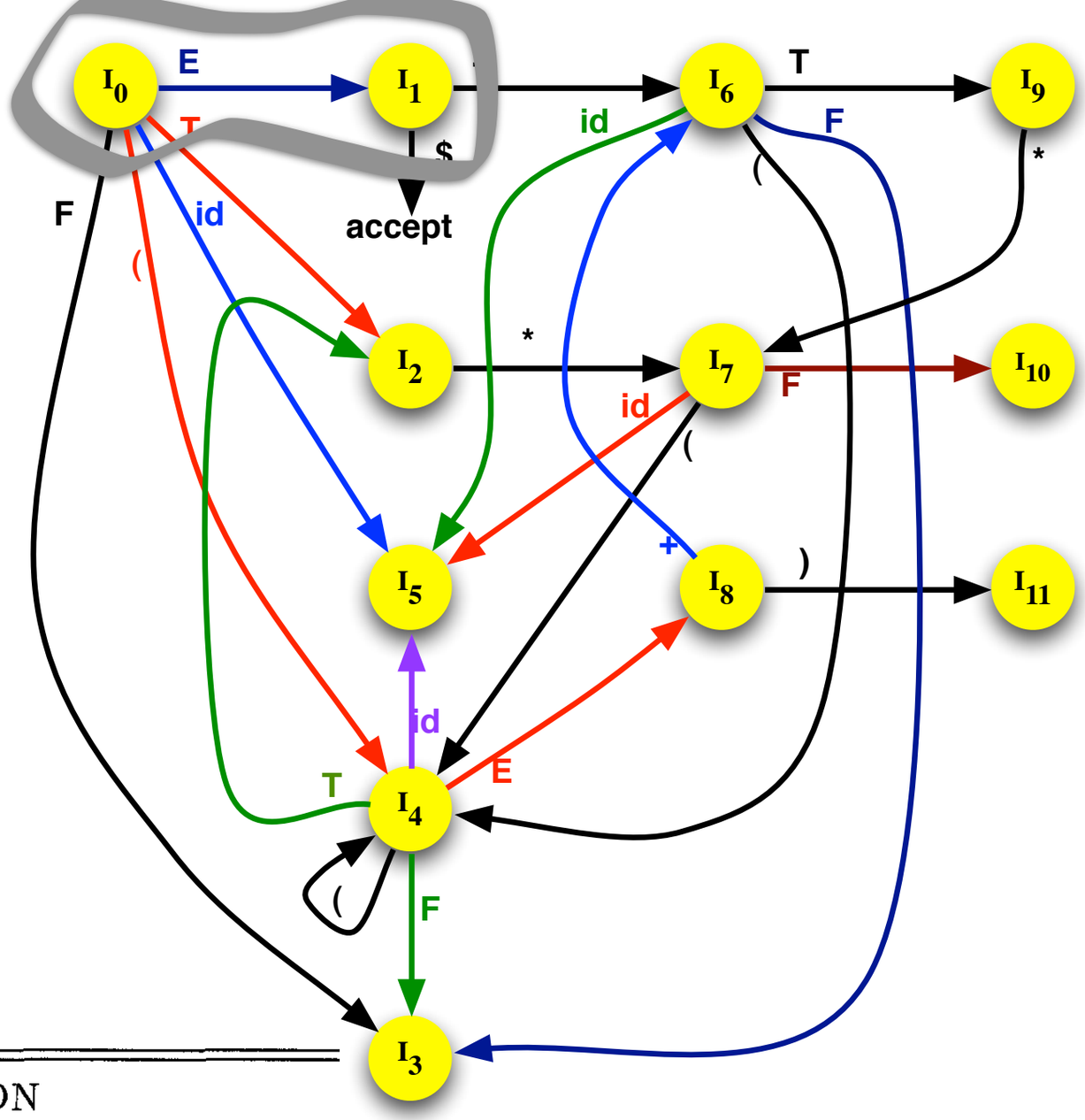


Line	Stack	Symbols	Input	Action
(1)	0	\$	**id** $*$ **id** \$	shift to 5
(2)	0 5	\$ **id**	$*$ **id** \$	reduce by $F \rightarrow$ **id**
(3)	0 3	\$ F	$*$ **id** \$	reduce by $T \rightarrow F$
(4)	0 2	\$ T	$*$ **id** \$	shift to 7
(5)	0 2 7	\$ $T*$	**id** \$	shift to 5
(6)	0 2 7 5	\$ $T*$ **id**	\$	reduce by $F \rightarrow$ **id**
(7)	0 2 7 10	\$ $T*F$	\$	reduce by $T \rightarrow T*F$
(8)	0 2	\$ T	\$	reduce by $E \rightarrow T$

E’ → E
E → E + T | T
T → T* F | F
F → (E) | **id**

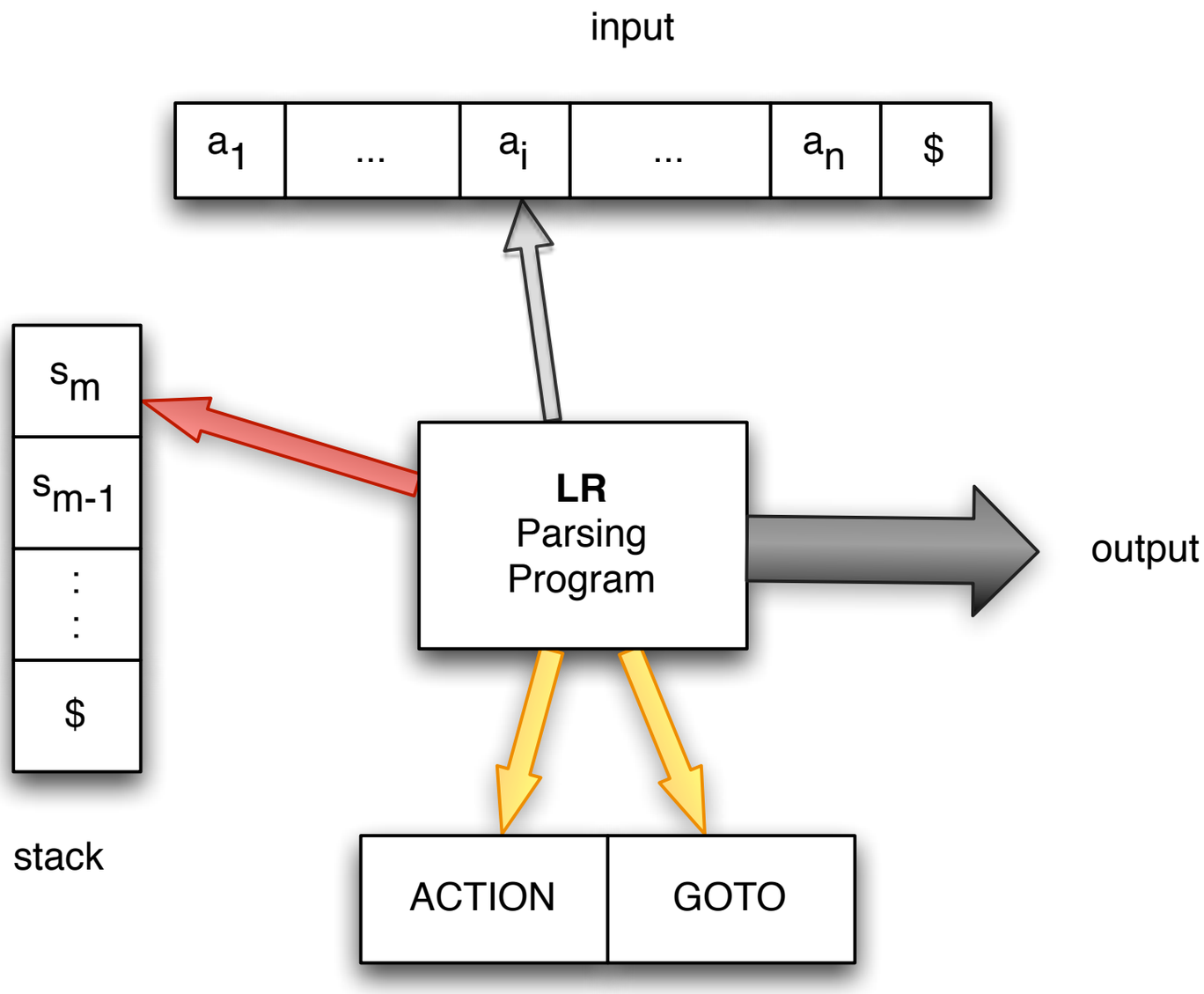
I_0
E’ –> .E
E –> . E+T
E –> .T
T –> .T*F
T –> .F
F –> .(E)
F –> .id

I_1
E’ –> E.
E –> E . + T



Line	Stack	Symbols	Input	Action
(1)	0	$	**id** ∗ **id** $	shift to 5
(2)	0 5	$ **id**	∗ **id** $	reduce by $F \rightarrow$ **id**
(3)	0 3	$ F	∗ **id** $	reduce by $T \rightarrow F$
(4)	0 2	$ T	∗ **id** $	shift to 7
(5)	0 2 7	$ T ∗	**id** $	shift to 5
(6)	0 2 7 5	$ T ∗ **id**	$	reduce by $F \rightarrow$ **id**
(7)	0 2 7 10	$ T ∗ F	$	reduce by $T \rightarrow T * F$
(8)	0 2	$ T	$	reduce by $E \rightarrow T$
(9)	0 1	$ E	$	accept

MODEL OF AN LR PARSER



$I_0,\ldots,I_n$ enumeration of **items(G')**

I_0= CLOSURE({S' → •S})

Q = {0,...,n} **s** ⟶ **I_s**

GOTO(s,X)=s' ⇔ GOTO(I_s,X)= $I_{s'}$

Proposition: if GOTO(s',X)=s and GOTO(s",Y)=s then X=Y

∀ s ∈ Q we associate a unique symbol $X_s \in T_{G'} \cup N_{G'}$ to s≠0
if GOTO(s',X)=s then X_s = X.
Moreover, X_0 represents the symbol $ (bottom of the stack)

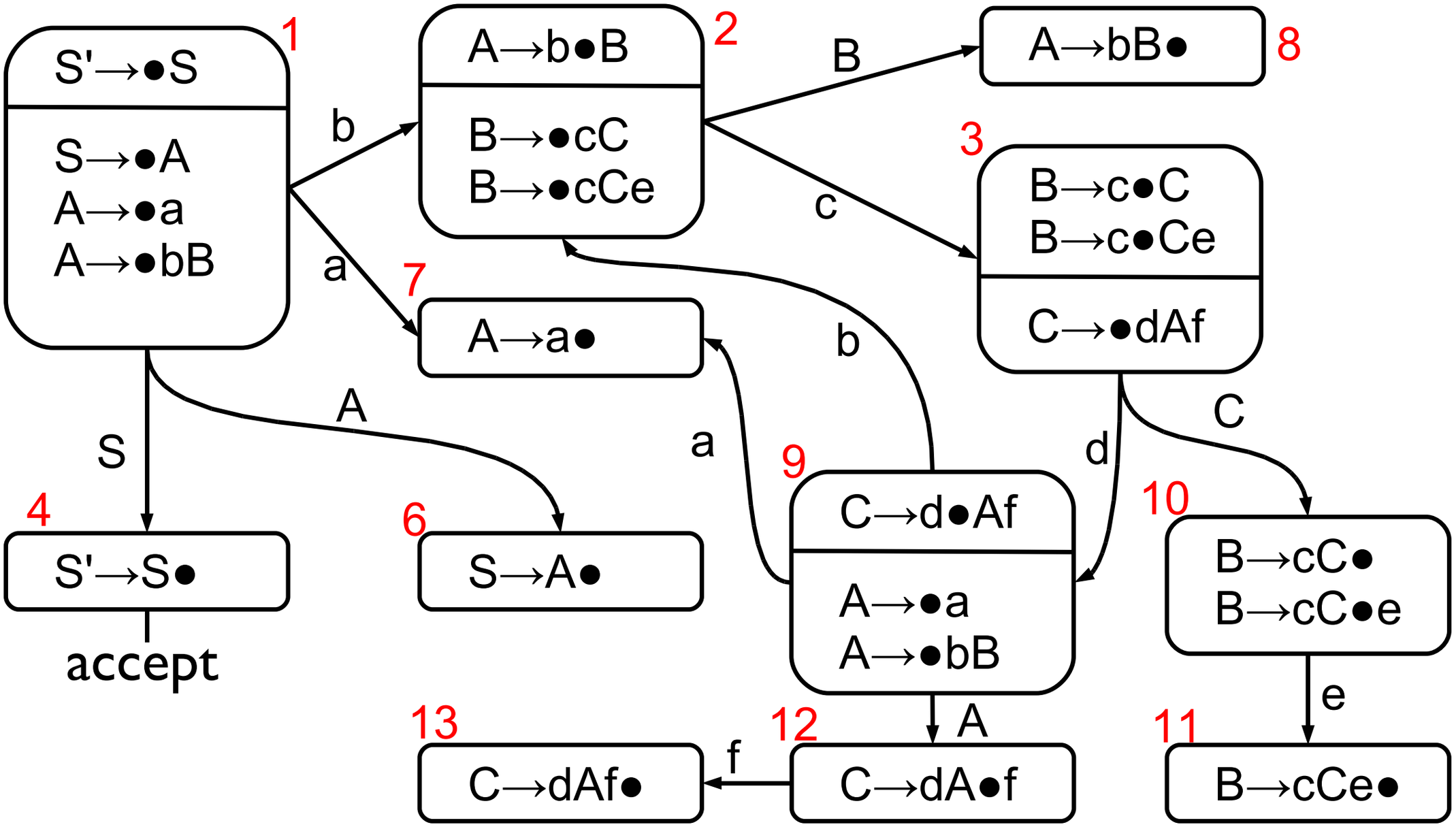
⟨**Q**, **0**, **GOTO**: Q × ($T_{G'} \cup N_{G'}$) → Q, **F**⟩

Fix for G' a finite bijective enumeration
p: {1,...,n} → G'–productions,
s.t. p(1)= S→S'

Construct the LR(0) automaton for the grammar

S′→S
S→A
A→bB
A→a
B→cC
B→cCe
C → dAf

S′→S
S→A
A→bB
A→a
B→cC
B→cCe
C → dAf



S′→S
S→A
A→bB
A→a
B→cC
B→cCe
C → dAf

Computing FOLLOW

- Place $ into FOLLOW(S)
- Repeat until nothing changes:
  - if $A \to \alpha B \beta$ then add FIRST(β)\{ε} to FOLLOW(B)
  - if $A \to \alpha B$ then add FOLLOW(A) to FOLLOW(B)
  - if $A \to \alpha B \beta$ and ε is in FIRST(β) then add FOLLOW(A) to FOLLOW(B)

- Follow $(S) =$
- Follow $(A) =$
- Follow $(B) =$
- Follow $(C) =$

S′→S
S→A
A→bB
A→a
B→cC
B→cCe
C → dAf

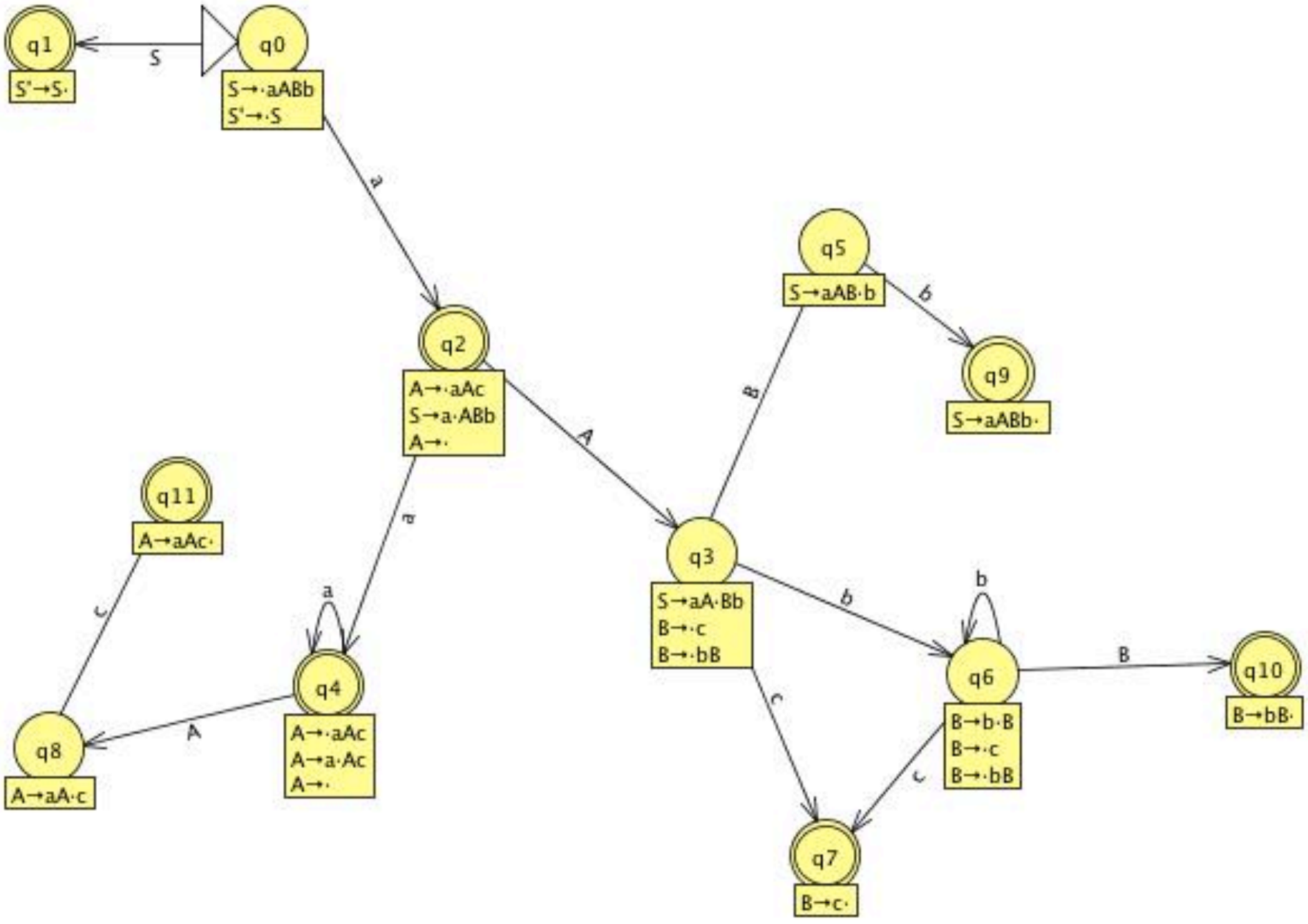
Computing FOLLOW

- Place $ into FOLLOW(S)
- Repeat until nothing changes:
  - if $A \to \alpha B \beta$ then add FIRST(β)\{ε} to FOLLOW(B)
  - if $A \to \alpha B$ then add FOLLOW(A) to FOLLOW(B)
  - if $A \to \alpha B \beta$ and ε is in FIRST(β) then add FOLLOW(A) to FOLLOW(B)

- Follow $(S) = \{\$\}$
- Follow $(A) = \{\texttt{f}, \$\}$
- Follow $(B) = \{\texttt{f}, \$\}$
- Follow $(C) = \{\texttt{e}, \texttt{f}, \$\}$

S'	→	S
S	→	aABb
A	→	aAc
B	→	bB
B	→	c
A	→	λ

S'	→	S
S	→	aABb
A	→	aAc
B	→	bB
B	→	c
A	→	λ



PARSING TABLE

terminals and **$**

non terminals

State	ACTION			GOTO		
		a			A	
i		A				
j					G	

A= ACTION[i,a]
G = GOTO[j,A]

ACTION[i,a]= s j (shift symbol j (namely X_j))
ACTION[i,a]= r k (reduce production p(k))
ACTION[i,a]= accept
ACTION[i,a]= error

LR-PARSER CONFIGURATIONS

configuration

stack | remaining input

$(0\ s_1 \ldots s_m , a_i\ a_{i+1} \ldots\ a_n\ \$)$

top

right-sentential form

$X_0\ X_{s_1} \ldots X_{s_m}\ \ a_i\ a_{i+1} \ldots\ \ a_n$

BEHAVIOR OF THE LR PARSER

($s_0\ s_1 \ldots s_m$, $a_i\ a_{i+1} \ldots\ a_n$ \$)

ACTION[s_m,a_i] = shift s

($s_0\ s_1 \ldots s_m$ **s**, $a_{i+1} \ldots\ a_n$ \$)

symbol X_s

($s_0\ s_1 \ldots s_m$, $a_i\ a_{i+1} \ldots\ a_n$ \$)

ACTION[s_m,a_i] = reduce k

($s_0\ s_1 \ldots s_{m-r}$ **s**, $a_i\ a_{i+1} \ldots\ a_n$ \$)

$p(k)=A \rightarrow \beta$

$|\beta| = r$

s=GOTO[s_{m-r}, A]

symbol X_s

BEHAVIOR OF THE LR PARSER

($s_0 s_1 \ldots s_m$, $a_i a_{i+1} \ldots a_n$ \$)

ACTION[s_m,a_i] = accept

end of parsing

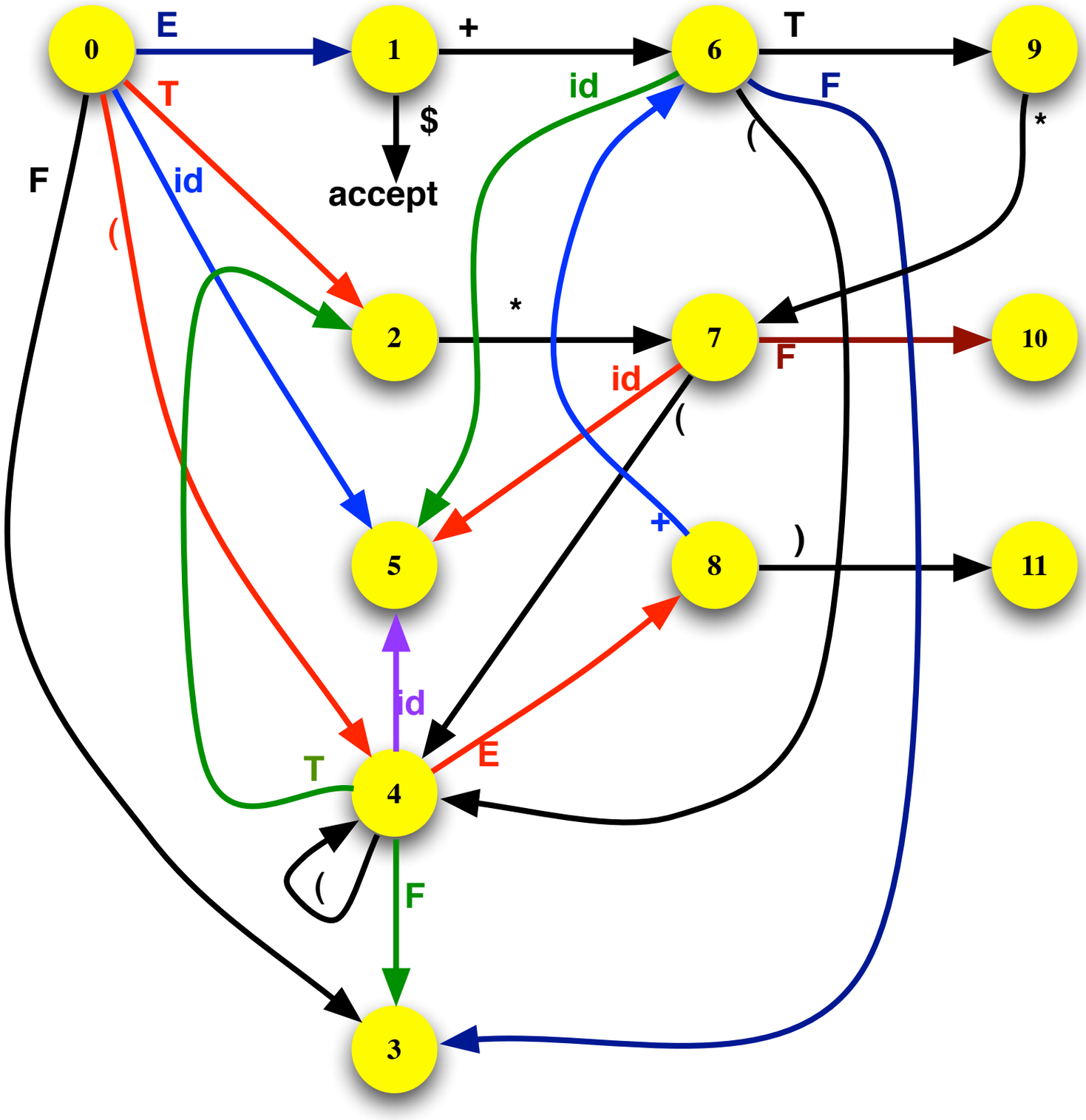
($s_0 s_1 \ldots s_m$, $a_i a_{i+1} \ldots a_n$ \$)

ACTION[s_m,a_i] = error

error recovery routine

LR-parsing algorithm

```
let a be the first symbol of w$;
while(1) { /* repeat forever */
        let s be the state on top of the stack;
        if ( ACTION[s, a] = shift t ) {
                push t onto the stack;
                let a be the next input symbol;
        } else if ( ACTION[s, a] = reduce A → β ) {
                pop |β| symbols off the stack;
                let state t now be on top of the stack;
                push GOTO[t, A] onto the stack;
                output the production A → β;
        } else if ( ACTION[s, a] = accept ) break; /* parsing is done */
        else call error-recovery routine;
}
```



$X_1 = E$

$X_2 = T$

$X_3 = F$

$X_4 = ($

$X_5 = id$

$X_6 = +$

$X_7 = *$

$X_8 = E$

$X_9 = T$

$X_{10} = F$

$X_{11} =)$

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

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			

$X_1 = E$
$X_2 = T$
$X_3 = F$
$X_4 = ($
$X_5 = id$
$X_6 = +$
$X_7 = *$
$X_8 = E$
$X_9 = T$
$X_{10} = F$
$X_{11} =)$

id * id + id

	STACK	SYMBOLS	INPUT	ACTION
(1)	0		**id** $*$ **id** $+$ **id** \$	shift
(2)	0 5	**id**	$*$ **id** $+$ **id** \$	reduce by $F \to$ **id**
(3)	0 3	F	$*$ **id** $+$ **id** \$	reduce by $T \to F$
(4)	0 2	T	$*$ **id** $+$ **id** \$	shift
(5)	0 2 7	$T*$	**id** $+$ **id** \$	shift
(6)	0 2 7 5	$T*$ **id**	$+$ **id** \$	reduce by $F \to$ **id**
(7)	0 2 7 10	$T*F$	$+$ **id** \$	reduce by $T \to T*F$
(8)	0 2	T	$+$ **id** \$	reduce by $E \to T$
(9)	0 1	E	$+$ **id** \$	shift
(10)	0 1 6	$E+$	**id** \$	shift
(11)	0 1 6 5	$E+$ **id**	\$	reduce by $F \to$ **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

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			

$X_1 = E$
$X_2 = T$
$X_3 = F$
$X_4 = ($
$X_5 = id$
$X_6 = +$
$X_7 = *$
$X_8 = E$
$X_9 = T$
$X_{10} = F$
$X_{11} =)$

(1) $E \to E+T$
(2) $E \to T$
(3) $T \to T*F$
(4) $T \to F$
(5) $F \to (E)$
(6) $F \to$ **id**

1.G′ : augmented grammar, with enumeration **p** of productions;
2.construct the LR(0) automaton (canonical item set C={I_0,...,I_n},
  I_0=CLOSURE[S′→ •S] GOTO function);
3.**foreach** state i:
  (a) **if** (A→ α•aβ ∈ I_i && GOTO[I_i,a]=I_j)
      add shift j **to** ACTION[i,a];
  (b) **if** (A→ α• ∈ I_i && A ≠ S′)
      foreach (a ∈ **FOLLOW**(A))
          add reduce $\mathbf{p}^{-1}$(A → α) **to** ACTION[i,a];
  (c) **if** (S′→ S• ∈ I_i)
      add accept **to** ACTION[i,$]
4. **foreach** (state i,k && symbol A)
     if (GOTO[I_j,A]=I_k)
         add k **to** GOTO[s,A]
5. **if** (ACTION[i,a] is undefined) **add** error **to** ACTION[i,a]
6. **if** (GOTO[i,A] is undefined) **add** error **to** GOTO[i,A]

The parsing table consisting of the ACTION and GOTO functions determined by Algorithm 4.46 is called the *SLR(1) table for* G. An LR parser using the SLR(1) table for G is called the SLR(1) parser for G, and a grammar having an SLR(1) parsing table is said to be *SLR(1)*. We usually omit the "(1)" after the "SLR," since we shall not deal here with parsers having more than one symbol of lookahead.

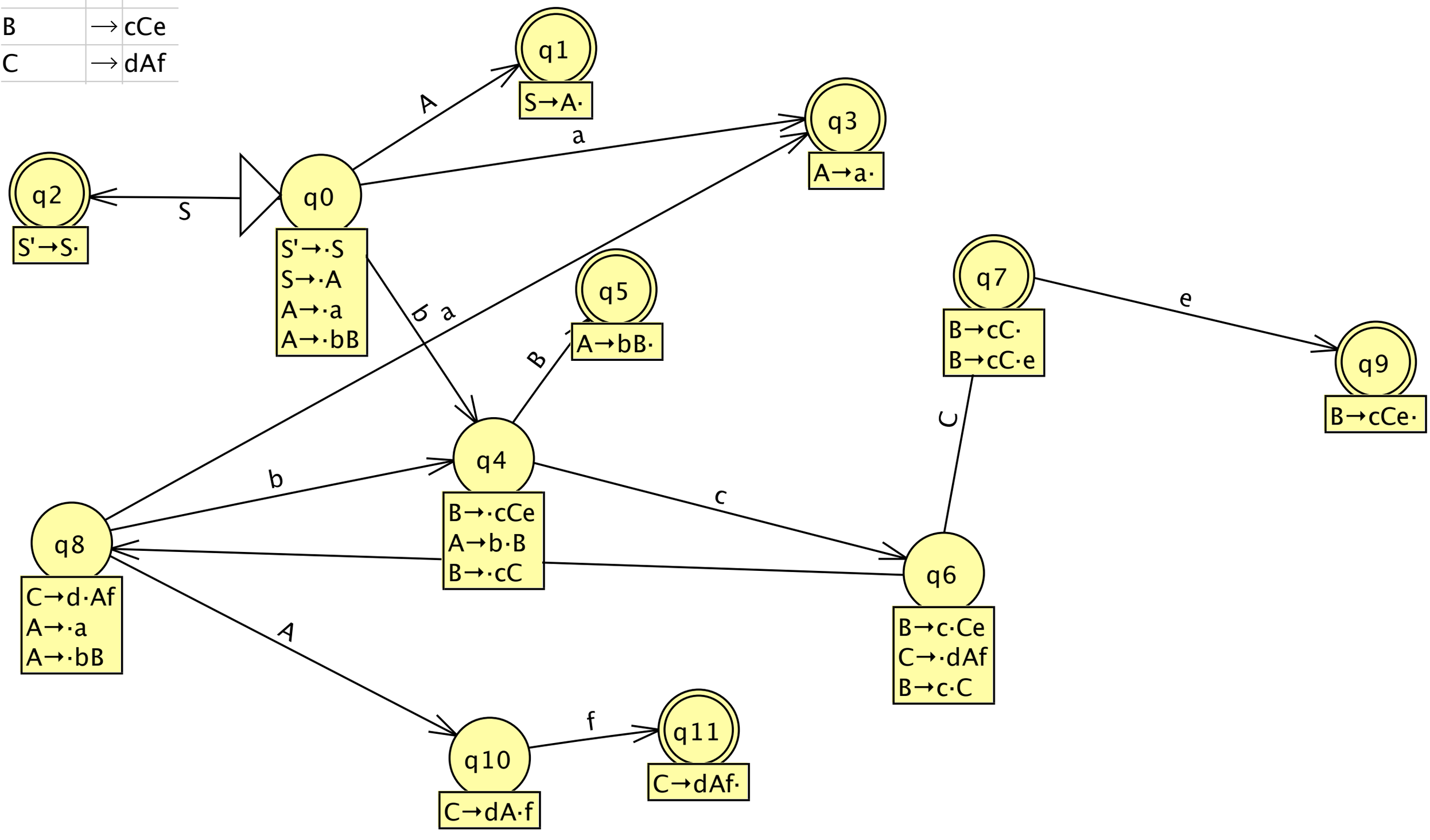
LR-parsing algorithm

```
let a be the first symbol of w$;
while(1) { /* repeat forever */
        let s be the state on top of the stack;
        if ( ACTION[s, a] = shift t ) {
                push t onto the stack;
                let a be the next input symbol;
        } else if ( ACTION[s, a] = reduce A → β ) {
                pop |β| symbols off the stack;
                let state t now be on top of the stack;
                push GOTO[t, A] onto the stack;
                output the production A → β;
        } else if ( ACTION[s, a] = accept ) break; /* parsing is done */
        else call error-recovery routine;
}
```

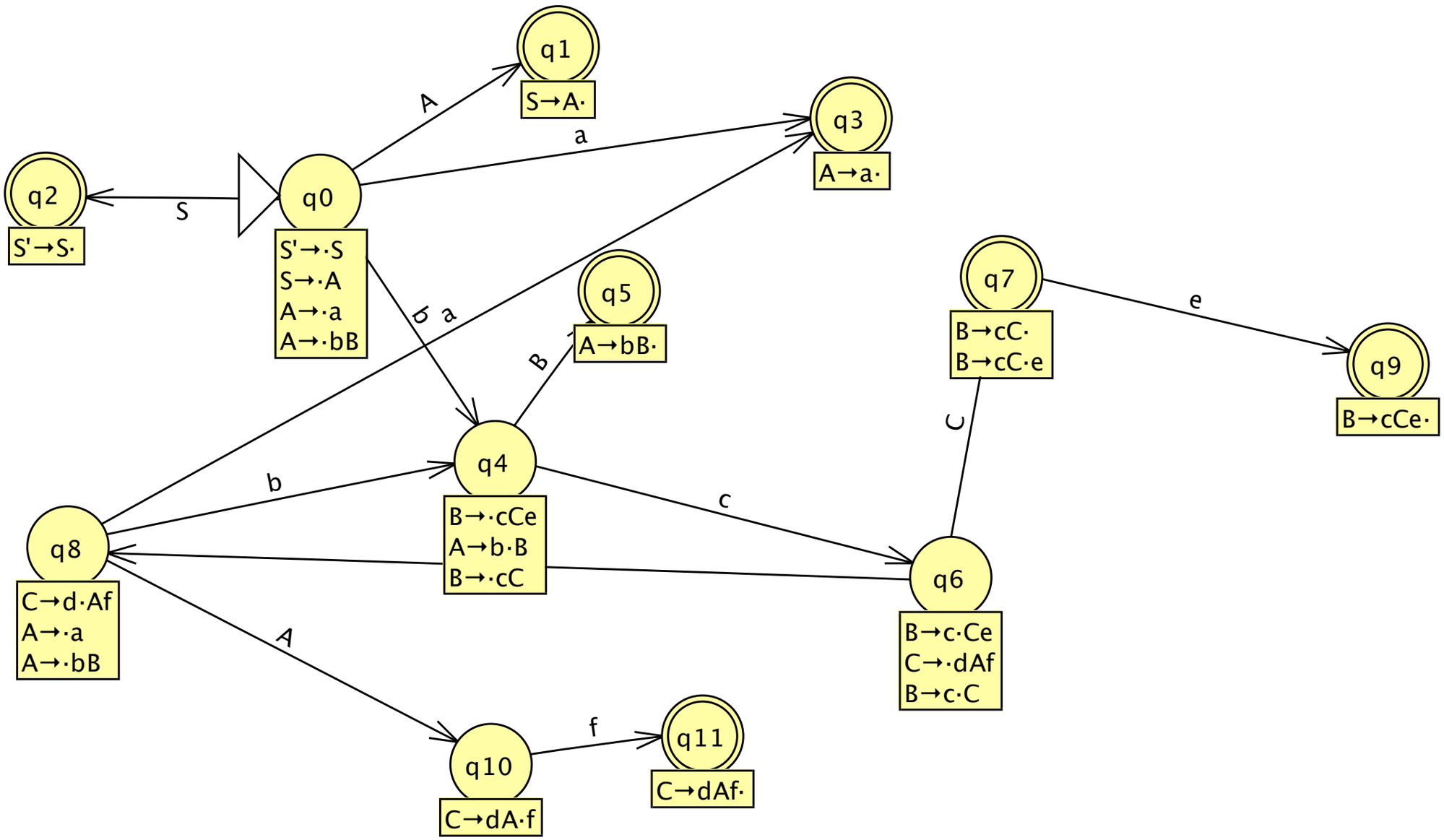
S'	→	S
S	→	A
A	→	bB
A	→	a
B	→	cC
B	→	cCe
C	→	dAf

Exercise: Construct the PARSING TABLE

S'	→	S
S	→	A
A	→	bB
A	→	a
B	→	cC
B	→	cCe
C	→	dAf



S'	→	S
S	→	A
A	→	bB
A	→	a
B	→	cC
B	→	cCe
C	→	dAf



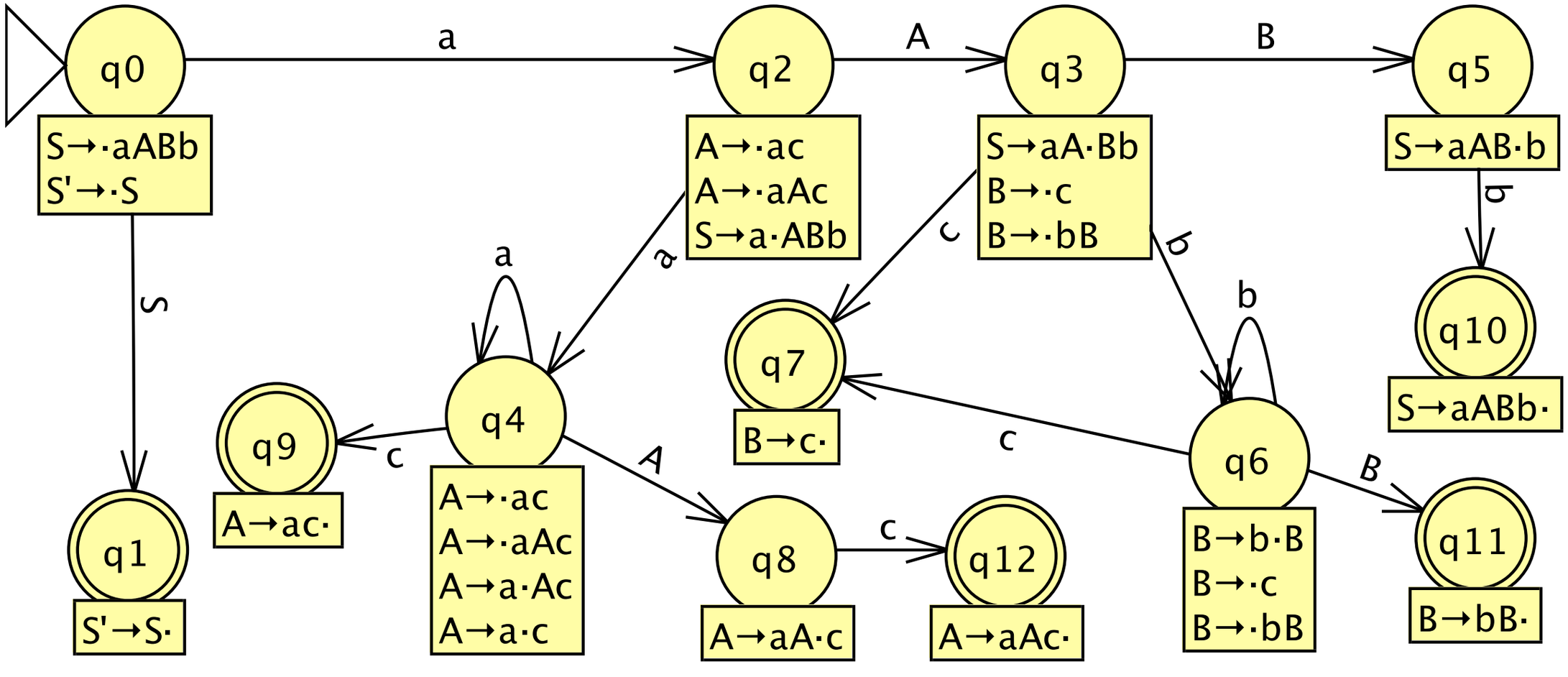
	FIRST	FOLLOW
A	{ b, a }	{ f, $ }
B	{ c }	{ f, $ }
C	{ d }	{ f, e, $ }
S	{ b, a }	{ $ }

Constructing an SLR-parsing Table

1. G′ : augmented grammar, with enumeration **p** of productions;
2. construct the LR(0) automaton (canonical item set $C=\{I_0,...,I_n\}$, I_0=CLOSURE[S′→ •S] GOTO function);
3. **foreach** state i:
   (a) **if** (A→ α•aβ ∈ I_i && GOTO[I_i,a]=I_j)
        add shift j **to** ACTION[i,a];
   (b) **if** (A→ α• ∈ I_i && A ≠ S′)
        foreach (a ∈ **FOLLOW**(A))
            add reduce $\mathbf{p^{-1}}$(A → α) **to** ACTION[i,a];
   (c) **if** (S′→ S• ∈ I_i)
        add accept **to** ACTION[i,$]
4. **foreach** (state i,k && symbol A)
      if (GOTO[I_j,A]=I_k)
           add k **to** GOTO[s,A]
5. **if** (ACTION[i,a] is undefined) **add** error **to** ACTION[i,a]
6. **if** (GOTO[i,A] is undefined) **add** error **to** GOTO[i,A]

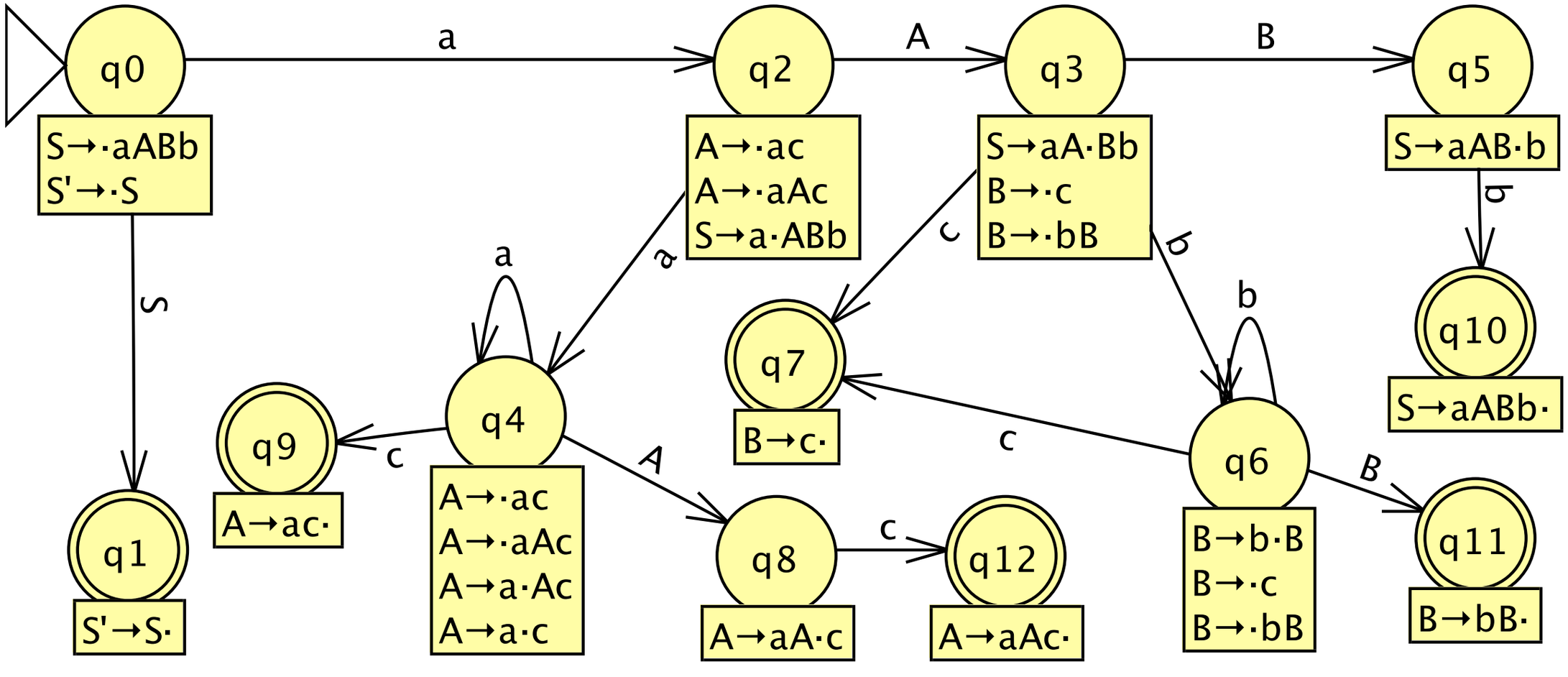
S'	$\longrightarrow$	S
S	$\longrightarrow$	aABb
A	$\longrightarrow$	aAc
A	$\longrightarrow$	ac
B	$\longrightarrow$	bB
B	$\longrightarrow$	c

S'	→	S
S	→	aABb
A	→	aAc
A	→	ac
B	→	bB
B	→	c



S'	→	S
S	→	aABb
A	→	aAc
A	→	ac
B	→	bB
B	→	c

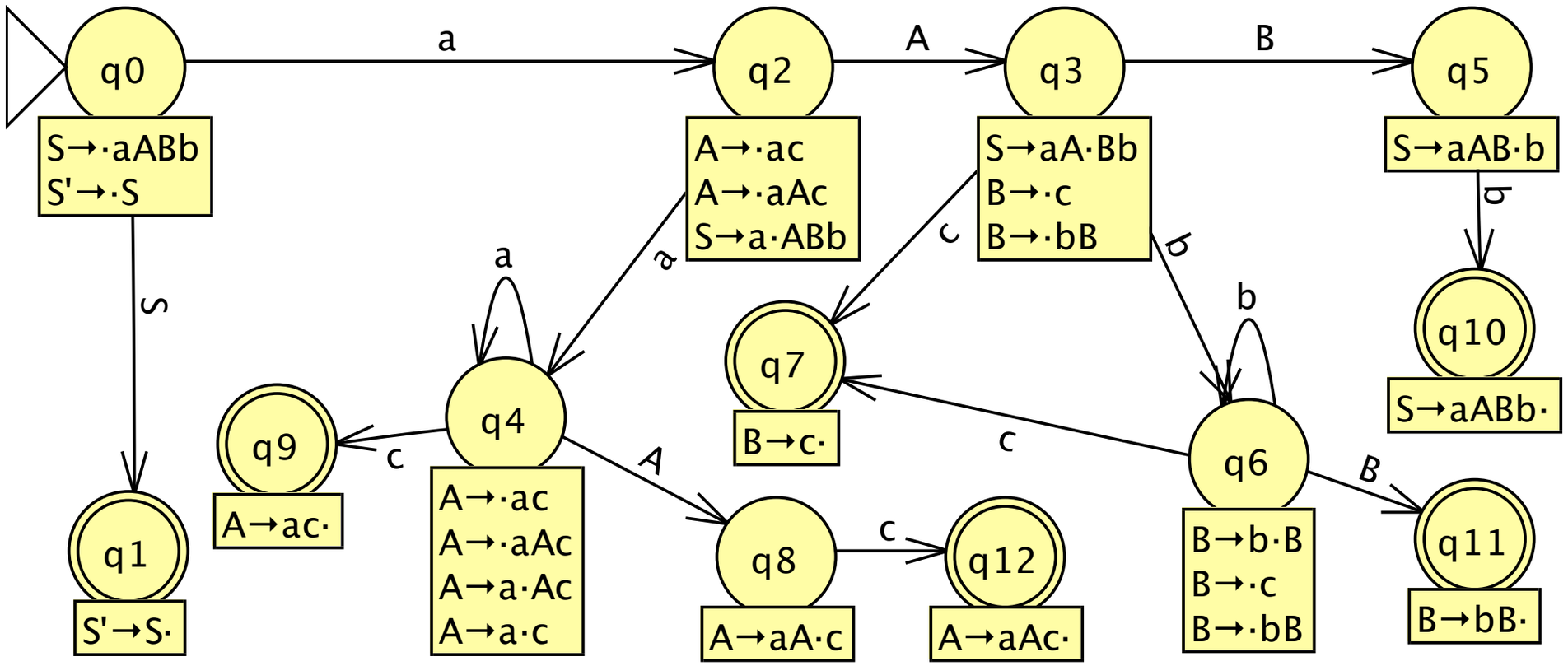
	FIRST	FOLLOW
A	{ a }	{ b, c }
B	{ b, c }	{ b }
S	{ a }	{ $ }



S'	→	S
S	→	aABb
A	→	aAc
A	→	ac
B	→	bB
B	→	c

	FIRST	FOLLOW
A	{ a }	{ b, c }
B	{ b, c }	{ b }
S	{ a }	{ $ }

	a	b	c	$	A	B	S
0	s2						1
1				acc			
2	s4				3		
3		s6	s7			5	
4	s4		s9		8		
5		s10					
6		s6	s7			11	
7		r5					
8			s12				
9		r3	r3				
10				r1			
11		r4					
12		r2	r2				



$$
\begin{array}{lcl}
S & \rightarrow & L = R \mid R \\
L & \rightarrow & *R \mid \mathbf{id} \\
R & \rightarrow & L
\end{array}
$$

it is not ambiguous

Item set ?

$$
\begin{array}{lcl}
S & \rightarrow & L = R \mid R \\
L & \rightarrow & *R \mid \mathbf{id} \\
R & \rightarrow & L
\end{array}
$$

Item set :

I_0:
$S' \rightarrow \cdot S$
$S \rightarrow \cdot L = R$
$S \rightarrow \cdot R$
$L \rightarrow \cdot * R$
$L \rightarrow \cdot \mathbf{id}$
$R \rightarrow \cdot L$

I_1: $S' \rightarrow S\cdot$

I_2:
$S \rightarrow L\cdot = R$
$R \rightarrow L\cdot$

I_3: $S \rightarrow R\cdot$

I_4:
$L \rightarrow *\cdot R$
$R \rightarrow \cdot L$
$L \rightarrow \cdot * R$
$L \rightarrow \cdot \mathbf{id}$

I_5: $L \rightarrow \mathbf{id}\cdot$

I_6:
$S \rightarrow L = \cdot R$
$R \rightarrow \cdot L$
$L \rightarrow \cdot * R$
$L \rightarrow \cdot \mathbf{id}$

I_7: $L \rightarrow *R\cdot$

I_8: $R \rightarrow L\cdot$

I_9: $S \rightarrow L = R\cdot$

Parsing table?

$$
\begin{array}{lcl}
S & \rightarrow & L = R \mid R \\
L & \rightarrow & *R \mid \mathbf{id} \\
R & \rightarrow & L
\end{array}
$$

Item set :

I_0:
$S' \rightarrow \cdot S$
$S \rightarrow \cdot L = R$
$S \rightarrow \cdot R$
$L \rightarrow \cdot * R$
$L \rightarrow \cdot \mathbf{id}$
$R \rightarrow \cdot L$

I_1: $S' \rightarrow S\cdot$

I_2:
$S \rightarrow L\cdot = R$
$R \rightarrow L\cdot$

I_3: $S \rightarrow R\cdot$

I_4:
$L \rightarrow *\cdot R$
$R \rightarrow \cdot L$
$L \rightarrow \cdot * R$
$L \rightarrow \cdot \mathbf{id}$

=

I_5: $L \rightarrow \mathbf{id}\cdot$

I_6:
$S \rightarrow L = \cdot R$
$R \rightarrow \cdot L$
$L \rightarrow \cdot * R$
$L \rightarrow \cdot \mathbf{id}$

I_7: $L \rightarrow *R\cdot$

I_8: $R \rightarrow L\cdot$

I_9: $S \rightarrow L = R\cdot$

```
if (A→ α• ∈ Iᵢ && A ≠ S′)
 foreach (a ∈ FOLLOW(A))
 add reduce p⁻¹(A → α) to ACTION[i,a];
```

parsing table

shift 6 ∈ ACTION[2,=]
symbol **=** ∈ FOLLOW(R)
and therefore:
reduce R → L ∈ ACTION[2,=]

conflict!

Viable prefix

A prefix of right-sentential forms that can appear on the top of the stack

Not all prefixes of right-sentential forms can appear on the stack, however, since the parser must not shift past the handle

Example: $E \underset{rm}{\overset{*}{\Rightarrow}} F * \mathbf{id} \underset{rm}{\Rightarrow} (E) * \mathbf{id}$

STACK: **(**, **(E**, and **(E)**, **but not (E)***

(E) is a handle (the parser must reduce it to *F* before shifting *)

Viable prefix:

- a prefix of right-sentential forms that can appear on the top of the stack and that does not continue past the right end of the rightmost handle of that sentential form.
- a prefix γ of $\alpha\beta$, where

$$S' \underset{rm}{\overset{*}{\Rightarrow}} \alpha A w \underset{rm}{\Rightarrow} \alpha\beta w$$

$A \rightarrow \beta_1 \bullet \beta_2$ is **valid** for a viable prefix $\alpha\beta_1$ if

$$S' \underset{rm}{\overset{*}{\Rightarrow}} \alpha A w \underset{rm}{\Rightarrow} \alpha\beta_1\beta_2 w$$

$A \rightarrow \beta_1 \bullet \beta_2$ is **valid** for a viable prefix $\alpha\beta_1$ if

$$S' \underset{rm}{\overset{*}{\Rightarrow}} \alpha A w \underset{rm}{\Rightarrow} \alpha\beta_1\beta_2 w$$

if $\beta_2 \neq \varepsilon$ then shift

if $\beta_2 = \varepsilon$ then reduce $A \rightarrow \beta_1$

possible conflicts

Two valid items may tell us to do different things for the same viable prefix. Some of these conflicts can be resolved by looking at the next input symbol

THEOREM: The set of valid items for a viable prefix γ is exactly the set of items reached from the initial state along the path labeled γ in the LR(0) automaton for the grammar.

$$
\begin{array}{lcl}
S & \rightarrow & L = R \mid R \\
L & \rightarrow & *R \mid \mathbf{id} \\
R & \rightarrow & L
\end{array}
$$

Item set :

I_0:
$S' \rightarrow \cdot S$
$S \rightarrow \cdot L = R$
$S \rightarrow \cdot R$
$L \rightarrow \cdot * R$
$L \rightarrow \cdot \mathbf{id}$
$R \rightarrow \cdot L$

I_1: $S' \rightarrow S\cdot$

I_2:
$S \rightarrow L\cdot = R$
$R \rightarrow L\cdot$

I_3: $S \rightarrow R\cdot$

I_4:
$L \rightarrow *\cdot R$
$R \rightarrow \cdot L$
$L \rightarrow \cdot * R$
$L \rightarrow \cdot \mathbf{id}$

I_5: $L \rightarrow \mathbf{id}\cdot$

I_6:
$S \rightarrow L = \cdot R$
$R \rightarrow \cdot L$
$L \rightarrow \cdot * R$
$L \rightarrow \cdot \mathbf{id}$

=

I_7: $L \rightarrow *R\cdot$

I_8: $R \rightarrow L\cdot$

I_9: $S \rightarrow L = R\cdot$

Parsing table

shift 6 ∈ ACTION[2,=]
symbol = ∈ FOLLOW(R)
and therefore:
reduce R → L ∈ ACTION[2,=]

Conflict!

LR(1) PARSING

Grammar:

- $S \rightarrow L = R \mid R$
- $L \rightarrow *R \mid id$
- $R \rightarrow L$

I_0: $S' \rightarrow \cdot S$
$S \rightarrow \cdot L = R$
$S \rightarrow \cdot R$
$L \rightarrow \cdot * R$
$L \rightarrow \cdot \mathbf{id}$
$R \rightarrow \cdot L$

I_1: $S' \rightarrow S\cdot$

I_2: $S \rightarrow L\cdot = R$
$R \rightarrow L\cdot$

I_3: $S \rightarrow R\cdot$

I_4: $L \rightarrow *\cdot R$
$R \rightarrow \cdot L$
$L \rightarrow \cdot * R$
$L \rightarrow \cdot \mathbf{id}$

I_5: $L \rightarrow \mathbf{id}\cdot$

I_6: $S \rightarrow L = \cdot R$
$R \rightarrow \cdot L$
$L \rightarrow \cdot * R$
$L \rightarrow \cdot \mathbf{id}$

I_7: $L \rightarrow *R\cdot$

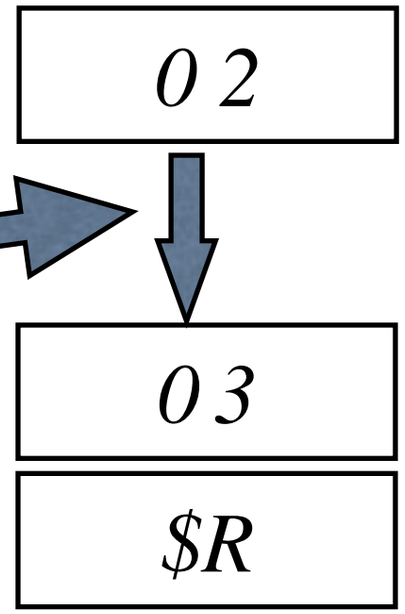
I_8: $R \rightarrow L\cdot$

I_9: $S \rightarrow L = R\cdot$

$\$L$

shift 6 ∈ ACTION[2,=]
symbol **=** ∈ FOLLOW(R)
and therefore:

~~**reduce R → L ∈ ACTION[2,=]**~~



there is no right-sentential form of the grammar that begins *R* = ...

Thus state 2, which is the state corresponding to viable prefix L only, should not really call for reduction of that L to *R*

In SLR(1) parsing,
if $A \rightarrow \alpha\bullet \in S_i$, **and** $a \in FOLLOW(A)$,
then we perform the reduction $A \rightarrow \alpha$

However, it is possible that when state **i** is on the top of the stack, we have the viable prefix $\beta\alpha$ on the top of the stack, and βA cannot be followed by a.

In this case, we cannot perform the reduction $A \rightarrow \alpha$.

LR(1) items

An LR(1) item is in the form of

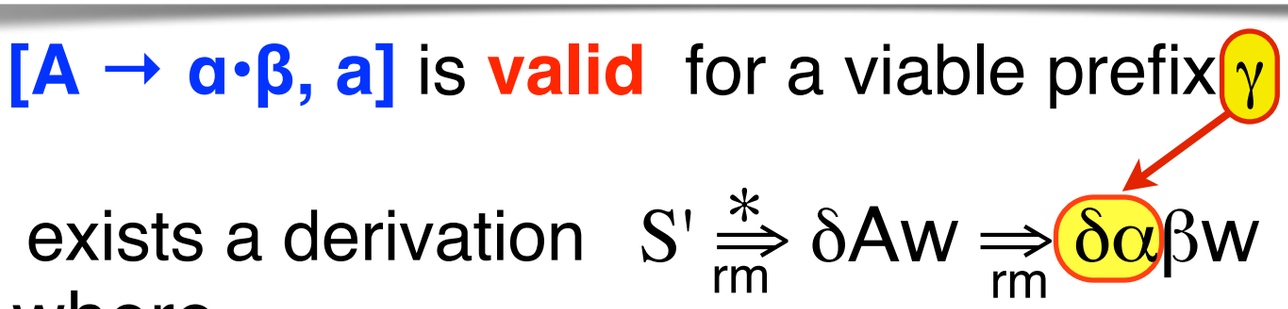
[A → α·β,a]

1. the **first field A → α·β** is an LR(0) item
2. the **second field a** is a terminal belonging to a subset (possibly proper) of **FOLLOW**[A].

Intuition: **perform a reduction based on an LR(1) item [A → α·, a] only when the next symbol is a.**

Instead of maintaining FOLLOW sets of viable prefixes, we maintain FIRST sets of possible future extensions of the current viable prefix.

[A → α·β, a] is **valid** for a viable prefix γ if there



exists a derivation $S' \underset{rm}{\overset{*}{\Rightarrow}} \delta Aw \underset{rm}{\Rightarrow} \delta\alpha\beta w$

where

- $\gamma = \delta\alpha$
- $w = aw'$ or $w=\varepsilon$ and $a=\$$.

Examples of $LR(1)$ items

- **Grammar:**
  - $S \to BB$
  - $B \to aB \mid b$
-

$$S \underset{rm}{\overset{*}{\Longrightarrow}} aaBab \underset{rm}{\Longrightarrow} aaaBab$$

viable prefix aaa **can reach** $[B \to a \cdot B, a]$

-

$$S \underset{rm}{\overset{*}{\Longrightarrow}} BaB \underset{rm}{\Longrightarrow} BaaB$$

viable prefix Baa **can reach** $[B \to a \cdot B, \$]$

[A → α•β, a] is **valid** for a viable prefix γ if there exists a derivation $S' \underset{rm}{\overset{*}{\Rightarrow}} \delta Aw \underset{rm}{\Rightarrow} \delta\alpha\beta w$

where

- $\gamma = \delta\alpha$
- w = aw' or w=ε and a=$.

Finding all $LR(1)$ items

- **Ideas: redefine the closure function.**
  - **Suppose $[A \to \alpha \cdot B\beta, a]$ is valid for a viable prefix $\gamma \equiv \delta\alpha$.**
  - **In other words,**

$$S \underset{rm}{\overset{*}{\Longrightarrow}} \delta \boxed{A} a\omega \underset{rm}{\Longrightarrow} \delta \boxed{\alpha B \beta} a\omega.$$

γ

    - ▷ *ω is ϵ or a sequence of terminals.*
  - **Then for each production $B \to \eta$, assume $\beta a\omega$ derives the sequence of terminals $bea\omega$.**

$$S \underset{rm}{\overset{*}{\Longrightarrow}} \underbrace{\delta\alpha}_{\gamma} B \boxed{\beta a\omega} \underset{rm}{\overset{*}{\Longrightarrow}} \underbrace{\delta\alpha}_{\gamma} B \boxed{bea\omega} \underset{rm}{\overset{*}{\Longrightarrow}} \underbrace{\delta\alpha\eta}_{\gamma} \boxed{bea\omega}$$

Thus $[B \to \cdot\eta, b]$ is also valid for γ for each $b \in$ FIRST(βa).
Note a is a terminal. So FIRST$(\beta a) =$ FIRST$(\beta a\omega)$.

[A → α·β, a] is **valid** for a viable prefix γ if there exists a derivation $S' \underset{rm}{\overset{*}{\Rightarrow}} \delta A w \underset{rm}{\Rightarrow} \underbrace{\delta\alpha}_{\gamma}\beta w$

where

- $\gamma = \delta\alpha$
- w = aw' or w=ε and a=$.

```
set_of_items CLOSURE(I: set_of_items){
  J=I;
  repeat{
    foreach ( [A → α•Bβ,a] ∈ J)
      foreach ( B → γ ∈ G)
        foreach (b ∈ FIRST[βa]) J = J ∪ {[B → •γ,b]};
  }until (no more new items are added to J)
  return J;
}
```

```
set_of_items GOTO(I set_of_items, X:symbol){
        J=∅;
        foreach([A → α•Xβ,a] ∈ I)
                J = J ∪ {[A → αX•β,a]};
        return CLOSURE(J);
}
```

```
CC_LR(1)_I items(G':augmented_grammar){
     C = {CLOSURE({[S' → •S,$]})} ;
     repeat{
          foreach(I ∈ C)
               foreach(grammar symbol X)
                    if(GOTO(I,X)≠∅ && GOTO(I,X) ∉ C)
                                        C = C ∪ {GOTO(I,X)};
     }until(no new sets of items are added to C)
     return C;
}
```

1. G′ : augmented grammar, with enumeration **p** of productions;
2. construct the LR(1) automaton (canonical item set $C=\{I_0,...,I_n\}$, I_0=CLOSURE[[S′→ •S,$]] GOTO function);
3. **foreach** state i:
   (a) **if** ([A→ α•aβ,b] ∈ I_i && GOTO[I_i,a]=I_j)
        add shift j **to** ACTION[i,a]; // a is terminal
   (b) **if** ([A→ α•,a] ∈ I_i && A ≠ S′)
        add reduce $\mathbf{p^{-1}}$(A → α) **to** ACTION[i,a];
   (c) **if** ([S′→ S•,$] ∈ I_i)
        add accept **to** ACTION[i,$]
4. **foreach** (state i,k && symbol A)
      if (GOTO[I_j,A]=I_k)
           add k **to** GOTO[s,A]
5. **if** (ACTION[i,a] is undefined) **add** error **to** ACTION[i,a]
6. **if** (GOTO[i,A] is undefined) **add** error **to** GOTO[i,A]

Example for constructing $LR(1)$ closures

- **Grammar:**
  - $S' \rightarrow S$
  - $S \rightarrow CC$
  - $C \rightarrow cC \mid d$
- $closure_1(\{[S' \rightarrow \cdot S, \$]\}) =$

```
set_of_items CLOSURE(I: set_of_items){
  J=I;
  repeat{
    foreach ( [A → α•Bβ,a] ∈ J)
      foreach ( B → γ ∈ G)
        foreach (b ∈ FIRST[βa]) J = J ∪ {[B → •γ,b]};
  }until (no more new items are added to J)
  return J;
}
```

Example for constructing $LR(1)$ closures

- **Grammar:**
  - $S' \to S$
  - $S \to CC$
  - $C \to cC \mid d$
- $closure\ (\{[S' \to \cdot S, \$]\}) =$

- **Note:**
  - **FIRST**$(\epsilon\$) = \{\$\}$
  - **FIRST**$(C\$) = \{c, d\}$
  - $[C \to \cdot cC, c/d]$ **means**
    - $[C \to \cdot cC, c]$ ***and***
    - $[C \to \cdot cC, d]$**.**

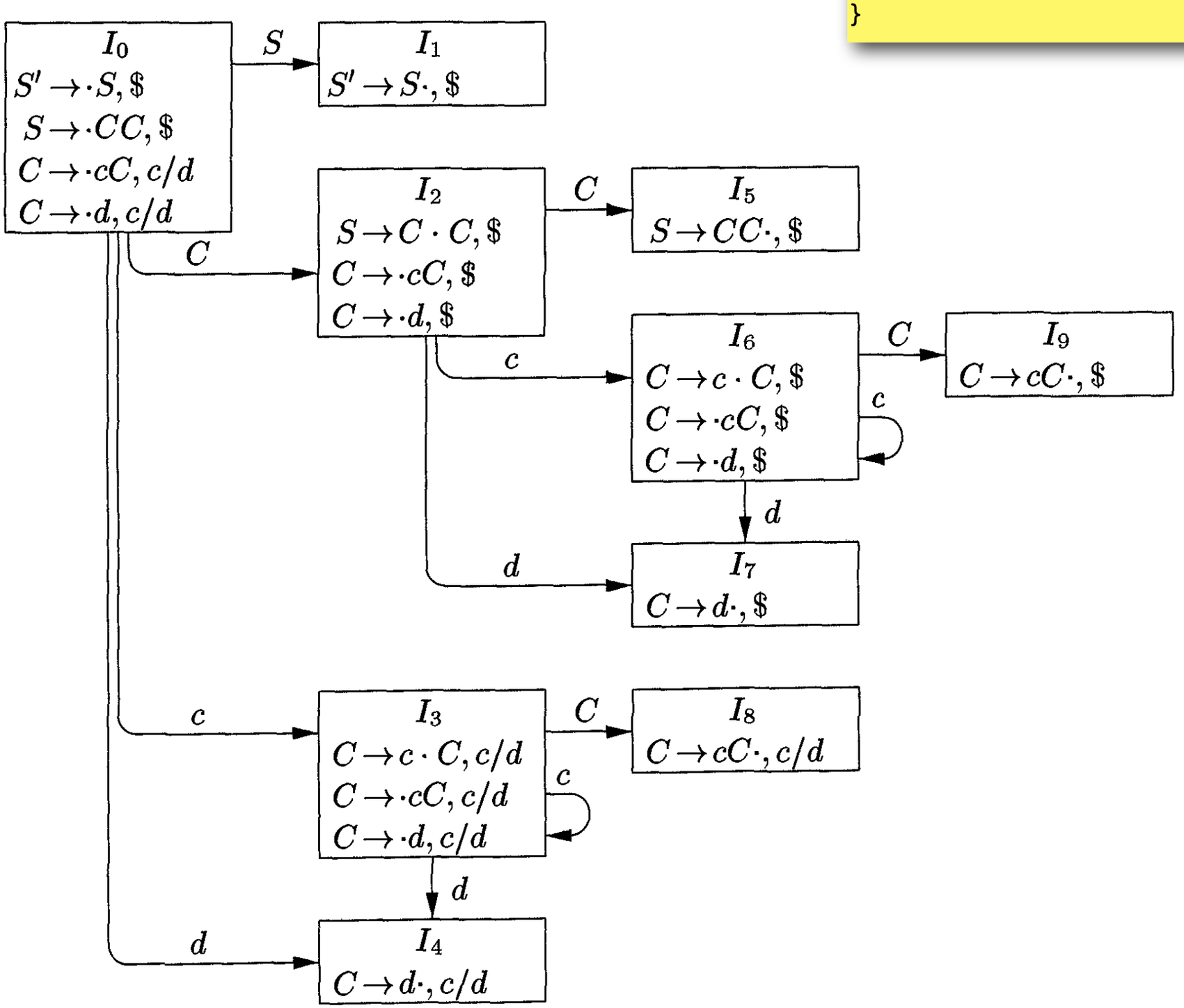
```
set_of_items CLOSURE(I: set_of_items){
  J=I;
  repeat{
    foreach ( [A → α•Bβ,a] ∈ J)
      foreach ( B → γ ∈ G)
        foreach (b ∈ FIRST[βa]) J = J ∪ {[B → •γ,b]};
  }until (no more new items are added to J)
  return J;
}
```

Example for constructing $LR(1)$ closures

- **Grammar:**
  - $S' \to S$
  - $S \to CC$
  - $C \to cC \mid d$
- $closure\ (\{[S' \to \cdot S, \$]\}) =$
  - $\{[S' \to \cdot S, \$],$
  - $[S \to \cdot CC, \$],$
  - $[C \to \cdot cC, c/d],$
  - $[C \to \cdot d, c/d]\}$
- **Note:**
  - $\textbf{FIRST}(\epsilon\$) = \{\$\}$
  - $\textbf{FIRST}(C\$) = \{c, d\}$
  - $[C \to \cdot cC, c/d]$ **means**
    - $[C \to \cdot cC, c]$ ***and***
    - $[C \to \cdot cC, d]$.

```
set_of_items CLOSURE(I: set_of_items){
  J=I;
  repeat{
    foreach ( [A → α•Bβ,a] ∈ J)
      foreach ( B → γ ∈ G)
        foreach (b ∈ FIRST[βa]) J = J ∪ {[B → •γ,b]};
  }until (no more new items are added to J)
  return J;
}
```

```
set_of_items CLOSURE(I: set_of_items){
  J=I;
  repeat{
    foreach ( [A → α•Bβ,a] ∈ J)
      foreach ( B → γ ∈ G)
        foreach (b ∈ FIRST[βa]) J = J ∪ {[B → •γ,b]};
  }until (no more new items are added to J)
  return J;
}
```



Example for constructing $LR(1)$ closures

- **Grammar:**
  - $S' \to S$
  - $S \to CC$
  - $C \to cC \mid d$
- $closure_1(\{[S' \to \cdot S, \$]\}) =$
  - $\{[S' \to \cdot S, \$]$,
  - $[S \to \cdot CC, \$]$,
  - $[C \to \cdot cC, c/d]$,
  - $[C \to \cdot d, c/d]\}$
- **Note:**
  - **FIRST**$(\epsilon\$) = \{\$\}$
  - **FIRST**$(C\$) = \{c, d\}$
  - $[C \to \cdot cC, c/d]$ **means**
    - $[C \to \cdot cC, c]$ ***and***
    - $[C \to \cdot cC, d]$**.**

	action$_1$			GOTO$_1$	
state	c	d	$	S	C
0	s3	s4		1	2
1			accept		
2	s6	s7			5
3	s3	s4			8
4	r3	r3			
5			r1		
6	s6	s7			9
7			r3		
8	r2	r2			
9			r2		

$LALR(1)$ parser — Lookahead LR

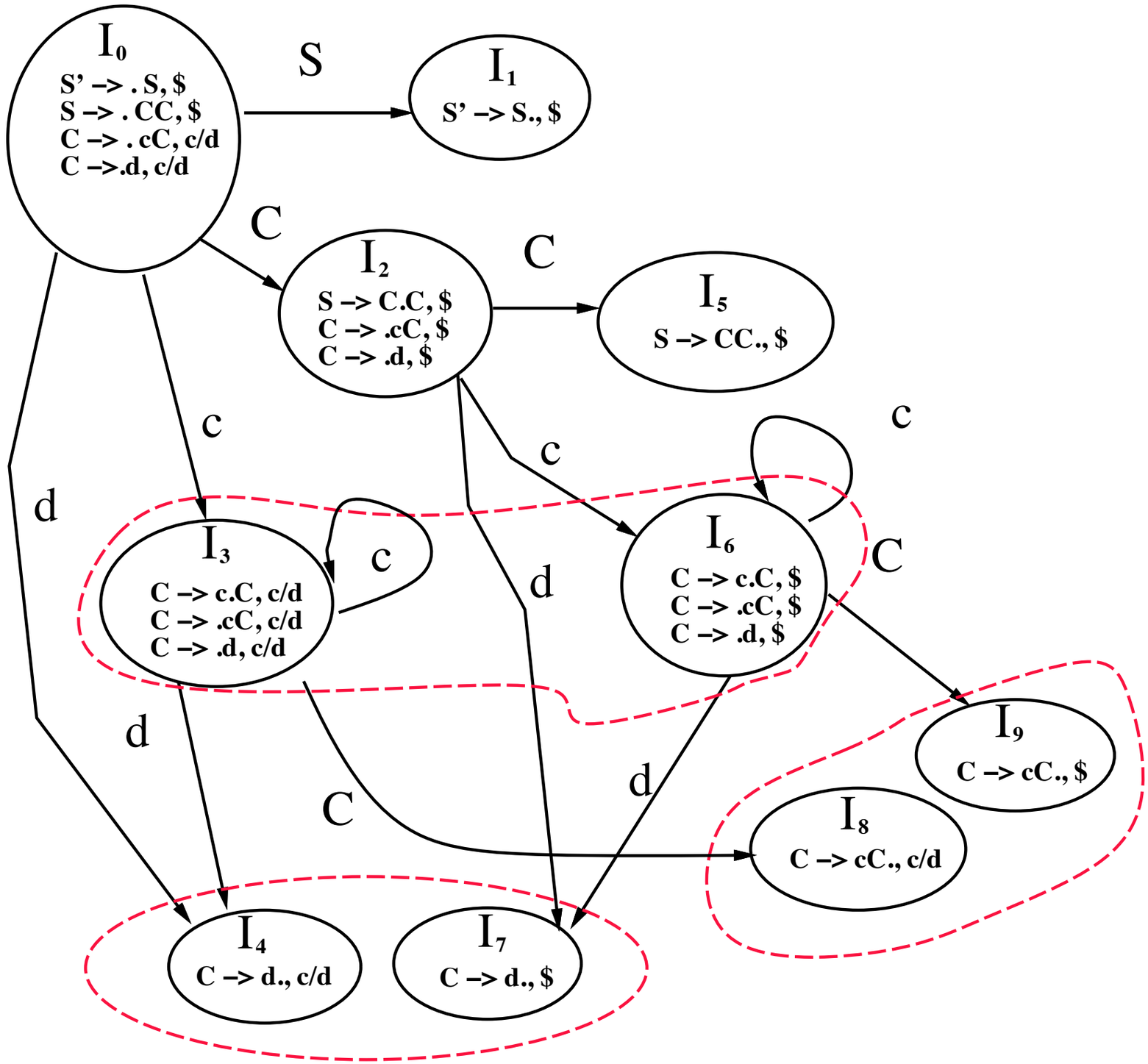
- **The method that is often used in practice.**
- **Most common syntactic constructs of programming languages can be expressed conveniently by an $LALR(1)$ grammar [DeRemer 1969].**
- **$SLR(1)$ and $LALR(1)$ always have the same number of states.**
- **Number of states is about 1/10 of that of $LR(1)$.**
- **Simple observation:**
  - **an $LR(1)$ item is of the form $[A \to \alpha \cdot \beta, c]$**
- **We call $A \to \alpha \cdot \beta$ the first component .**
- **Definition: in an $LR(1)$ state, set of first components is called its core .**

1. G′ : augmented grammar, with enumeration **p** of productions;
2. construct the LR(1) automaton (canonical item set $C=\{I_0,...,I_n\}$, I_0=CLOSURE[[S′→ •S,$]] GOTO function);
3. **foreach** state i:
   (a) **if** ([A→ α•aβ,b] ∈ I_i && GOTO[I_i,a]=I_j)
        add shift j **to** ACTION[i,a]; // a is terminal
   (b) **if** ([A→ α•,a] ∈ I_i && A ≠ S′)
        add reduce $\mathbf{p^{-1}}$(A → α) **to** ACTION[i,a];
   (c) **if** ([S′→ S•,$] ∈ I_i)
        add accept **to** ACTION[i,$]
4. **foreach** (state i,k && symbol A)
      if (GOTO[I_j,A]=I_k)
           add k **to** GOTO[s,A]
5. **if** (ACTION[i,a] is undefined) **add** error **to** ACTION[i,a]
6. **if** (GOTO[i,A] is undefined) **add** error **to** GOTO[i,A]

Intuition for $LALR(1)$ grammars

- **In an $LR(1)$ parser, it is a common thing that several states only differ in lookahead symbols, but have the same core.**
- **To reduce the number of states, we might want to merge states with the same core.**
  - **If I_4 and I_7 are merged, then the new state is called $I_{4,7}$.**
  - **After merging the states, revise the $GOTO_1$ table accordingly.**
- **Merging of states can never produce a shift-reduce conflict that was not present in one of the original states.**
  - $I_1 = \{[A \to \alpha\cdot, a], \ldots\}$
    - ▷ ***For I_1, one of the actions is to perform a reduce when the lookahead symbol is "a".***
  - $I_2 = \{[B \to \beta \cdot a\gamma, b], \ldots\}$
    - ▷ ***For I_2, one of the actions is to perform a shift on input "a".***
  - **Merging I_1 and I_2, the new state $I_{1,2}$ has shift-reduce conflicts.**
  - **However, we merge I_1 and I_2 because they have the same core.**
    - ▷ ***That is, $[A \to \alpha\cdot, c] \in I_2$ and $[B \to \beta \cdot a\gamma, d] \in I_1$.***
    - ▷ ***The shift-reduce conflict already occurs in I_1 and I_2.***
- **Merging of states can produce a new reduce-reduce conflict.**

$LALR(1)$ transition diagram



Possible new conflicts from $LALR(1)$

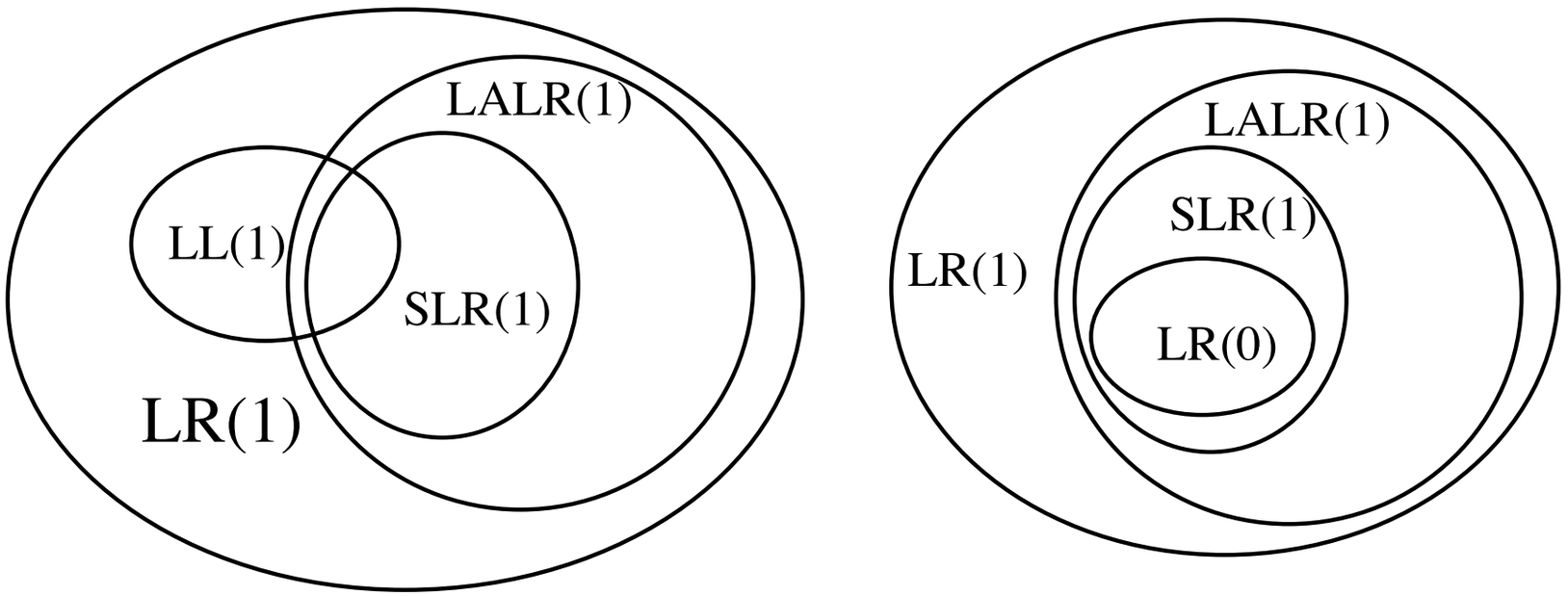
- **May produce a new reduce-reduce conflict.**
- **For example (textbook page 267, Example 4.58), grammar:**
  - $S' \to S$
  - $S \to aAd \mid bBf \mid aBe \mid bAe$
  - $A \to c$
  - $B \to c$
- **The language recognized by this grammar is $\{acd, ace, bcd, bce\}$.**
- **You may check that this grammar is $LR(1)$ by constructing the sets of items.**
- **You will find the set of items $\{[A \to c\cdot, d], [B \to c\cdot, e]\}$ is valid for the viable prefix ac, and $\{[A \to c\cdot, e], [B \to c\cdot, d]\}$ is valid for the viable prefix bc.**
- **Neither of these sets generates a conflict, and their cores are the same. However, their union, which is**
  - $\{[A \to c\cdot, d/e],$
  - $[B \to c\cdot, d/e]\}$,

  generates a reduce-reduce conflict, since reductions by both $A \to c$ and $B \to c$ are called for on inputs d and e.

How to construct $LALR(1)$ parsing table

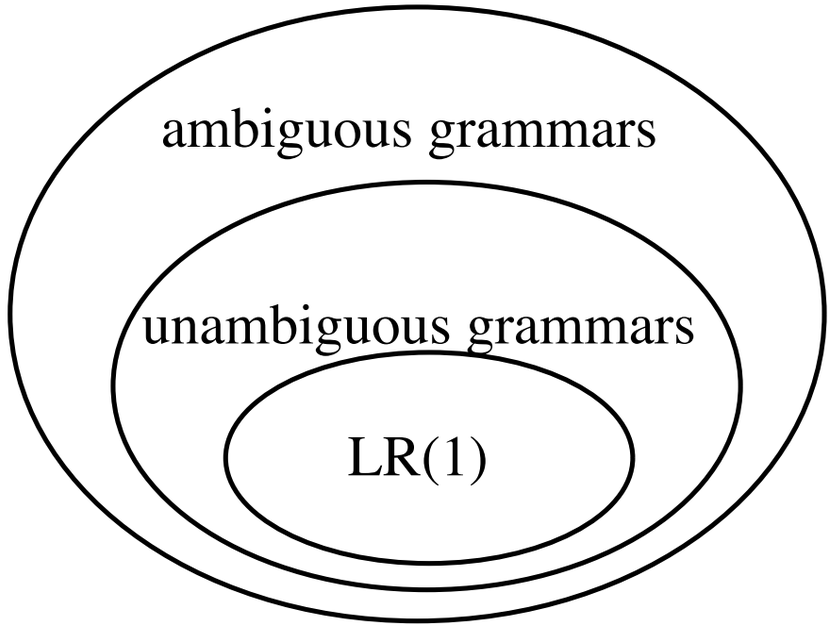
- **Naive approach:**
  - **Construct $LR(1)$ parsing table, which takes lots of intermediate spaces.**
  - **Merging states.**
- **Space and/or time efficient methods to construct an $LALR(1)$ parsing table are known.**
  - **Constructing and merging on the fly.**
  - $\cdots$

Summary



- $LR(1)$ **and** $LALR(1)$ **can almost express all important programming languages issues, but** $LALR(1)$ **is easier to write and uses much less space.**
- $LL(1)$ **is easier to understand and uses much less space, but cannot express some important common-language features.**

Ambiguous grammars are not too bad...



Ambiguous grammars often provide a shorter, more natural specification than their equivalent unambiguous grammars.

Ambiguity from precedence and associativity

- **Precedence and associativity are important language constructs.**
- **Example:**
  - G_1**:**
    - ▷ $E \longrightarrow E + E \mid E * E \mid (E) \mid id$
    - ▷ ***Ambiguous, but easy to understand and maintain!***
  - G_2**:**
    - ▷ $E \longrightarrow E + T \mid T$
    - ▷ $T \longrightarrow T * F \mid F$
    - ▷ $F \longrightarrow (E) \mid id$
    - ▷ ***Unambiguous, but difficult to understand and maintain!***

LR(0) states

I_0: $E' \to \cdot E$
$E \to \cdot E + E$
$E \to \cdot E * E$
$E \to \cdot (E)$
$E \to \cdot \mathbf{id}$

I_1: $E' \to E\cdot$
$E \to E\cdot + E$
$E \to E\cdot * E$

I_2: $E \to (\cdot E)$
$E \to \cdot E + E$
$E \to \cdot E * E$
$E \to \cdot (E)$
$E \to \cdot \mathbf{id}$

I_3: $E \to \mathbf{id}\cdot$

I_4: $E \to E + \cdot E$
$E \to \cdot E + E$
$E \to \cdot E * E$
$E \to \cdot (E)$
$E \to \cdot \mathbf{id}$

I_5: $E \to E * \cdot E$
$E \to \cdot E + E$
$E \to \cdot E * E$
$E \to \cdot (E)$
$E \to \cdot \mathbf{id}$

I_6: $E \to (E\cdot)$
$E \to E\cdot + E$
$E \to E\cdot * E$

I_7: $E \to E + E\cdot$
$E \to E\cdot + E$
$E \to E\cdot * E$

I_8: $E \to E * E\cdot$
$E \to E\cdot + E$
$E \to E\cdot * E$

I_9: $E \to (E)\cdot$

input: **id+id*id**

the parser enters state 7 after processing **id+id**

PREFIX	STACK	INPUT
$E + E$	0 1 4 7	$*$ **id** \$

If $*$ takes precedence over $+$, we know the parser should shift $*$ onto the stack, preparing to reduce the $*$ and its surrounding **id** symbols to an expression. This choice was made by the SLR parser of Fig. 4.37, based on an unambiguous grammar for the same language. On the other hand, if $+$ takes precedence over $*$, we know the parser should reduce $E + E$ to E.

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			

Figure 4.37: Parsing table for expression grammar

State	ACTION						GOTO
	id	+	*	(	)	$	E
0	s3			s2			1
1		s4	s5			acc	
2	s3			s2			6
3		r4	r4		r4	r4	
4	s3			s2			7
5	s3			s2			8
6		s4	s5		s9		
7		r1	s5		r1	r1	
8		r2	r2		r2	r2	
9		r3	r3		r3	r3	

Figure 4.49: Parsing table for grammar (4.3)

Ambiguity from dangling-else

- **Grammar:**
  - **Statement** $\rightarrow$ **Other_Statement**
    $\mid$ if **Condition** $then$ **Statement**
    $\mid$ if **Condition** $then$ **Statement** $else$ **Statement**
- **When seeing**

$$\underline{if\ \textbf{C}\ then\ \textbf{S}}\ else\ \textbf{S}$$

  - **there is a shift/reduce conflict,**
  - **we always favor a shift.**
  - **Intuition: favor a longer match.**
- **Need a mechanism to let user specify the default conflict-handling rule when there is a shift/reduce conflict.**

I_0: $S' \to \cdot S$
$S \to \cdot iSeS$
$S \to \cdot iS$
$S \to \cdot a$

I_1: $S' \to S\cdot$

I_2: $S \to i\cdot SeS$
$S \to i\cdot S$
$S \to \cdot iSeS$
$S \to \cdot iS$
$S \to \cdot a$

I_3: $S \to a\cdot$

I_4: $S \to iS\cdot eS$

I_5: $S \to iSe\cdot S$
$S \to \cdot iSeS$
$S \to \cdot iS$
$S \to \cdot a$

I_6: $S \to iSeS\cdot$

Figure 4.50: LR(0) states for augmented grammar (4.67)

State	ACTION				GOTO
	i	e	a	\$	S
0	s2		s3		1
1				acc	
2	s2		s3		4
3		r3		r3	
4		s5		r2	
5	s2		s3		6
6		r1		r1	

Figure 4.51: LR parsing table for the "dangling-else" grammar

	Stack	Symbols	Input	Action
(1)	0		$i\,i\,a\,e\,a\,\$$	shift
(2)	0 2	i	$i\,a\,e\,a\,\$$	shift
(3)	0 2 2	$i\,i$	$a\,e\,a\,\$$	shift
(4)	0 2 2 3	$i\,i\,a$	$e\,a\,\$$	shift
(5)	0 2 2 4	$i\,i\,S$	$e\,a\,\$$	reduce by $S \to a$
(6)	0 2 2 4 5	$i\,i\,S\,e$	$a\,\$$	shift
(7)	0 2 2 4 5 3	$i\,i\,S\,e\,a$	\$	reduce by $S \to a$
(8)	0 2 2 4 5 6	$i\,i\,S\,e\,S$	\$	reduce by $S \to iSeS$
(9)	0 2 4	$i\,S$	\$	reduce by $S \to iS$
(10)	0 1	S	\$	accept

Figure 4.52: Parsing actions on input $iiaea$