

# Sub: Compiler Construction

## Syntax Analysis PART 4

Compiled for: 7th Sem, CE, DDU

Compiled by: Niyati J. Buch

# Topics Covered

- Bottom-up parsing
  - [Introduction to LR Parsing](#)
  - [LR\(0\) automaton](#)
  - LR Parsing Table
    - [Structure of LR Parsing Table](#)
    - [LR Parsing Algorithm](#)
  - Simple LR (SLR)
    - [Constructing SLR parsing table](#)
    - [Construct SLR parsing table for expression grammar](#)
    - [Moves of an LR parser on  \$\text{id} \* \text{id} + \text{id}\$](#)
    - [An unambiguous grammar that is not SLR\(1\)](#)
    - [A grammar that is SLR\(1\) but not LL\(1\)](#)
    - [A grammar that is LL\(1\) but not SLR\(1\)](#)

# Introduction to LR Parsing


- The most prevalent type of bottom-up parser is based on a concept called **LR(k) parsing**
  - the "L" is for left-to-right scanning of the input,
  - the "R" for constructing a rightmost derivation in reverse,
  - and the **k** for the number of input symbols of lookahead that are used in making parsing decisions.
- The cases **k = 0 or k = 1** are of practical interest, and we shall only consider LR parsers with **k = 1** here.
- The easiest method for constructing shift-reduce parsers, called **"simple LR" (or SLR, for short)**.
- Two more complex methods are used in the majority of LR parsers:
  - **canonical-LR**
  - **LALR**

# LR Parsers

- LR parsers are **table-driven**, much like the nonrecursive LL parsers.
- A grammar for which we can construct a parsing table is said to be an LR grammar.
- Intuitively, for a grammar to be LR it is sufficient that a left-to-right shift-reduce parser be able to recognize handles of right-sentential forms when they appear on top of the stack.

# Why LR Parsers ???

- LR parsers can be constructed to **recognize virtually all programming language constructs** for which context-free grammars can be written. Non-LR context-free grammars exist, but these can generally be avoided for typical programming-language constructs.
- The LR-parsing method is the **most general nonbacktracking shift-reduce parsing method** known, yet it can be implemented as efficiently as other, more primitive shift-reduce methods .
- An LR parser can **detect a syntactic error as soon as** it is possible to do so on a left-to-right scan of the input.
- **The class of grammars that can be parsed using LR methods is a proper superset of the class of grammars that can be parsed with predictive or LL methods.**



LR grammars  
can describe  
more  
languages than  
LL grammars.

# Drawback

- The principal drawback of the LR method is that it is **too much work** to construct an LR parser by hand for a typical programming-language grammar.
- A specialized tool, an **LR parser generator (like YACC)**, is needed.
- Such a generator takes a context-free grammar and automatically produces a parser for that grammar.

# How does a shift-reduce parser know when to shift and when to reduce?

- An LR parser makes shift-reduce decisions **by maintaining states** to keep track of where we are in a parse.
- **States represent sets of "items".**
- An LR(0) item (item for short) of a grammar  $G$  is a **production of  $G$  with a dot at some position of the body.**
- Thus, production  $A \rightarrow XYZ$  yields the four items:
  1.  $A \rightarrow \cdot XYZ$
  2.  $A \rightarrow X \cdot YZ$
  3.  $A \rightarrow XY \cdot Z$
  4.  $A \rightarrow XYZ \cdot$
- The production  $A \rightarrow \epsilon$  generates only one item,  $A \rightarrow \cdot$ .

Intuitively, an item indicates how much of a production we have seen at a given point in the parsing process.

For example, the item  $A \rightarrow \cdot XYZ$  indicates that we hope to see a string derivable from  $XYZ$  next on the input.



# LR(0) automaton

- One collection of sets of LR(0) items, called the **canonical LR(0) collection**, provides the basis for constructing a deterministic finite automaton that is used to make parsing decisions.
- Such an automaton is called an **LR(0) automaton**.
- In particular, each state of the LR(0) automaton represents a set of items in the canonical LR(0) collection.

# Canonical LR(0) collection

- To construct the canonical LR(0) collection for a grammar, we define an **augmented grammar** and two functions, **CLOSURE** and **GOTO**.
- If  $G$  is a grammar with start symbol  $S$ , then  $G_0$ , the augmented grammar for  $G$ , is  $G$  with a **new start symbol  $S'$**  and production  $S' \rightarrow S$ .
- The **purpose** of this new starting production is to indicate to the parser when it should stop parsing and announce acceptance of the input.
- That is, acceptance occurs when and only when the parser is about to reduce by  $S' \rightarrow S$

# CLOSURE of Item Sets

- If  $I$  is a set of items for a grammar  $G$ , then  $\text{CLOSURE}(I)$  is the set of items constructed from  $I$  by the two rules:
  - Initially, add every item in  $I$  to  $\text{CLOSURE}(I)$ .
  - If  $A \rightarrow \alpha \cdot B \beta$  is in  $\text{CLOSURE}(I)$  and  $B \rightarrow \gamma$  is a production, then add the item  $B \rightarrow \cdot \gamma$  to  $\text{CLOSURE}(I)$ , if it is not already there. Apply this rule until no more new items can be added to  $\text{CLOSURE}(I)$ .
- For example:  $E' \rightarrow E$

$E' \rightarrow \cdot E$
$E \rightarrow \cdot E + T$
$E \rightarrow \cdot T$
$T \rightarrow \cdot T * F$
$T \rightarrow \cdot F$
$F \rightarrow \cdot ( E )$
$F \rightarrow \cdot \text{id}$

# GOTO

- The second useful function is  $GOTO(I, X)$  where  $I$  is a set of items and  $X$  is a grammar symbol.
- $GOTO(I, X)$  is defined to be the closure of the set of all items  $[A \rightarrow \alpha X \cdot \beta]$  such that  $[A \rightarrow \alpha \cdot X \beta]$  is in  $I$ .
- Intuitively, **the GOTO function is used to define the transitions in the LR(0) automaton for a grammar.**
- The states of the automaton correspond to sets of items, and  $GOTO(I, X)$  specifies the transition from the state for  $I$  under input  $X$ .

$I_0$

$E' \rightarrow \cdot E$

$E \rightarrow \cdot E + T$

$E \rightarrow \cdot T$

$T \rightarrow \cdot T * F$

$T \rightarrow \cdot F$

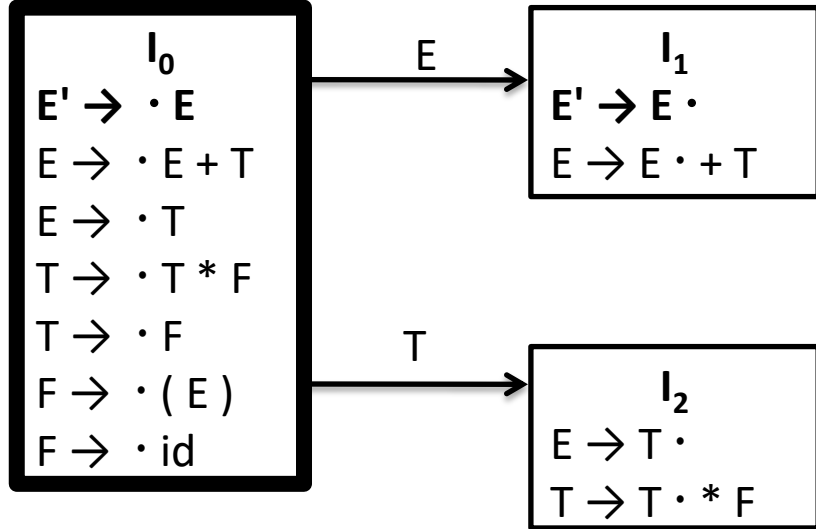
$F \rightarrow \cdot ( E )$

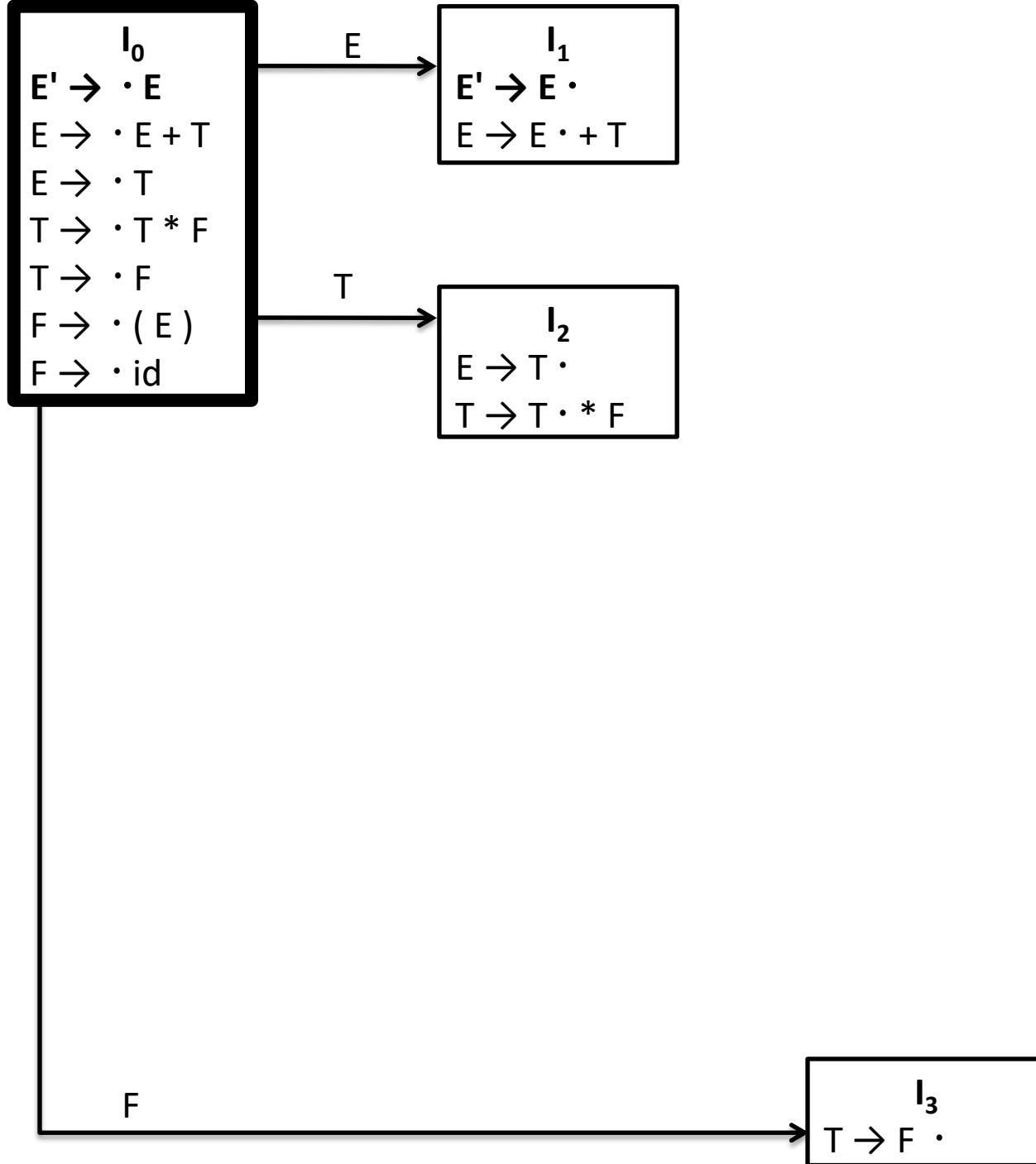
$F \rightarrow \cdot id$

$I_0$   
 $E' \rightarrow \cdot E$   
 $E \rightarrow \cdot E + T$   
 $E \rightarrow \cdot T$   
 $T \rightarrow \cdot T * F$   
 $T \rightarrow \cdot F$   
 $F \rightarrow \cdot ( E )$   
 $F \rightarrow \cdot id$

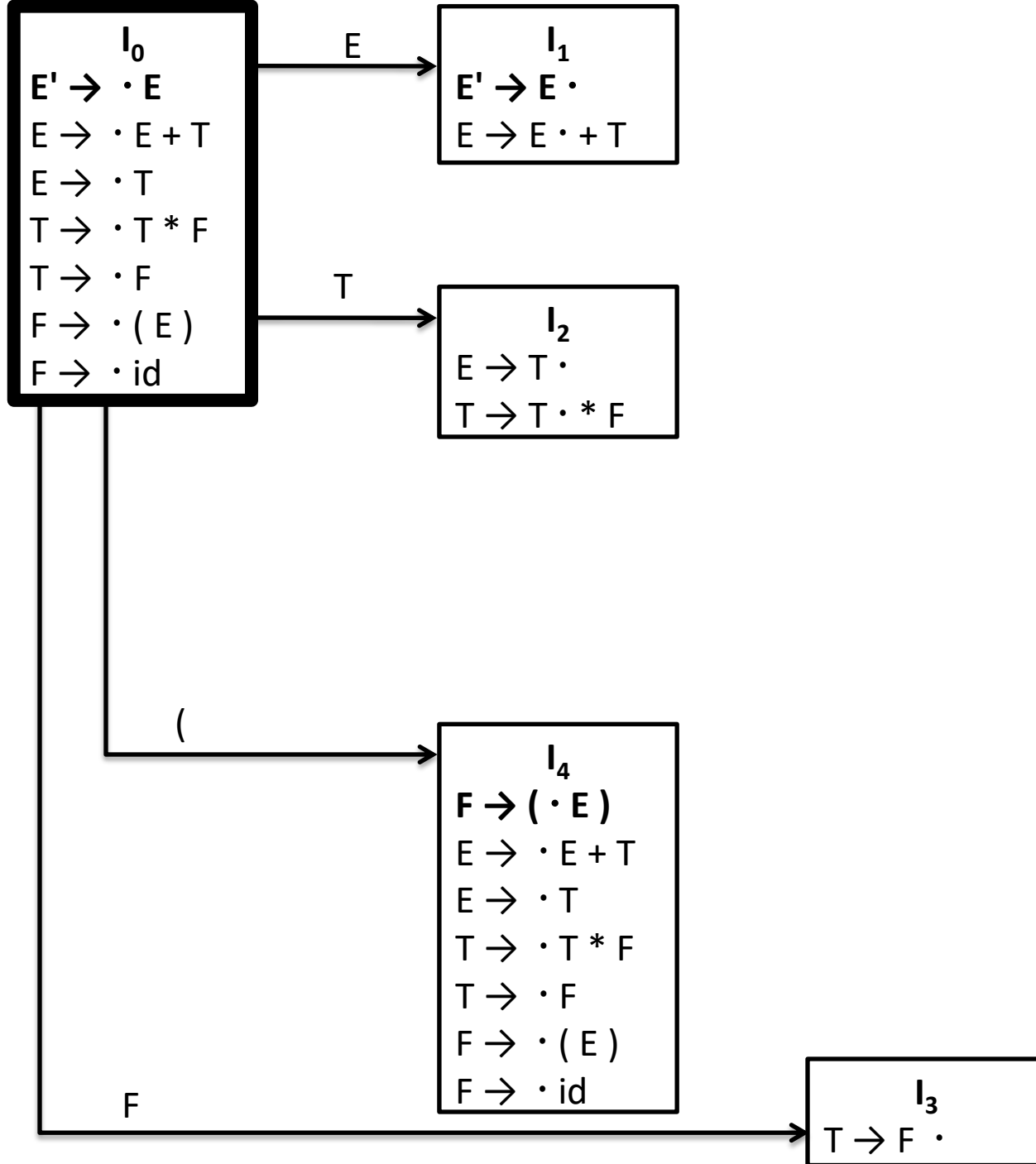
E

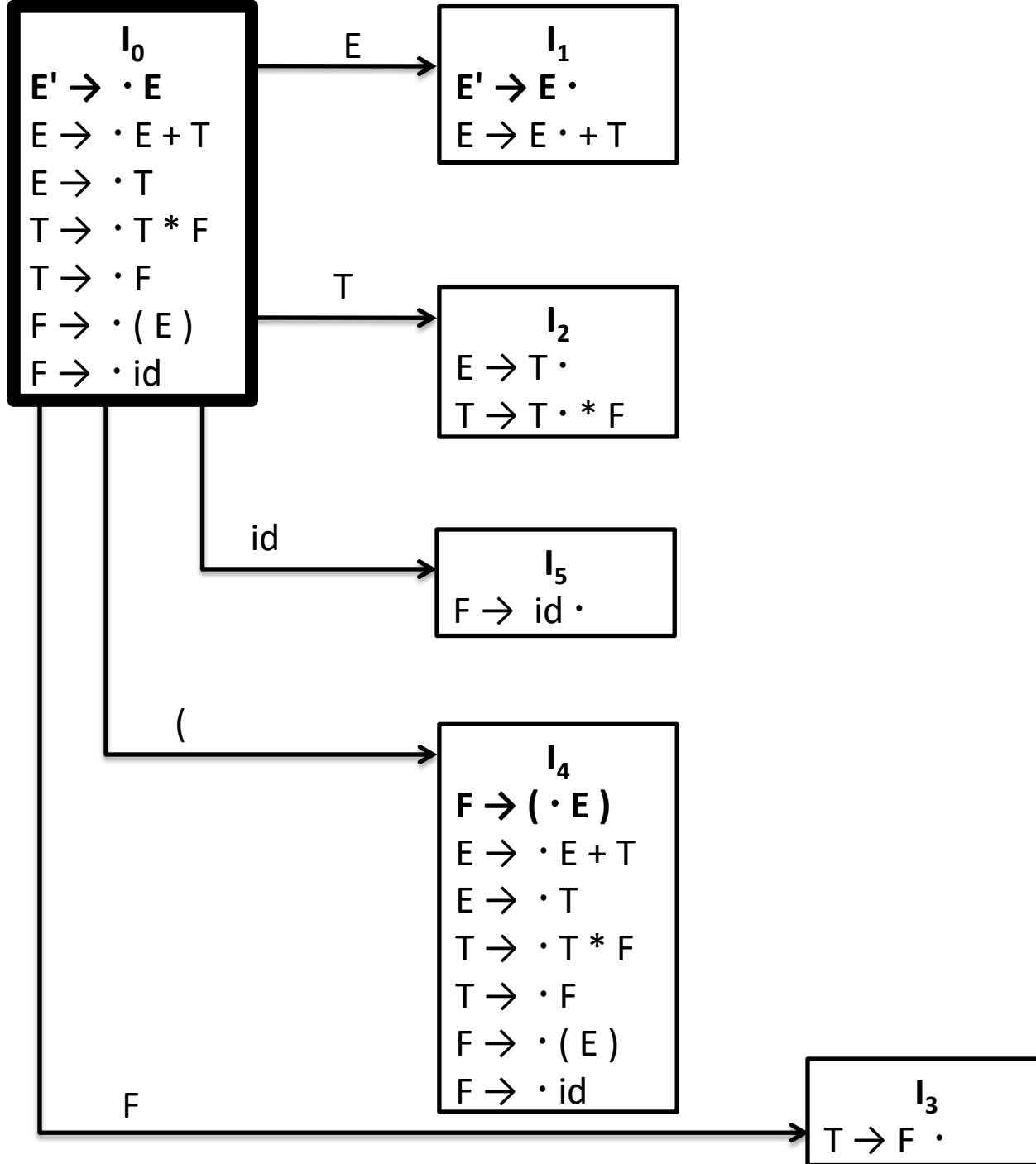
$I_1$   
 $E' \rightarrow E \cdot$   
 $E \rightarrow E \cdot + T$

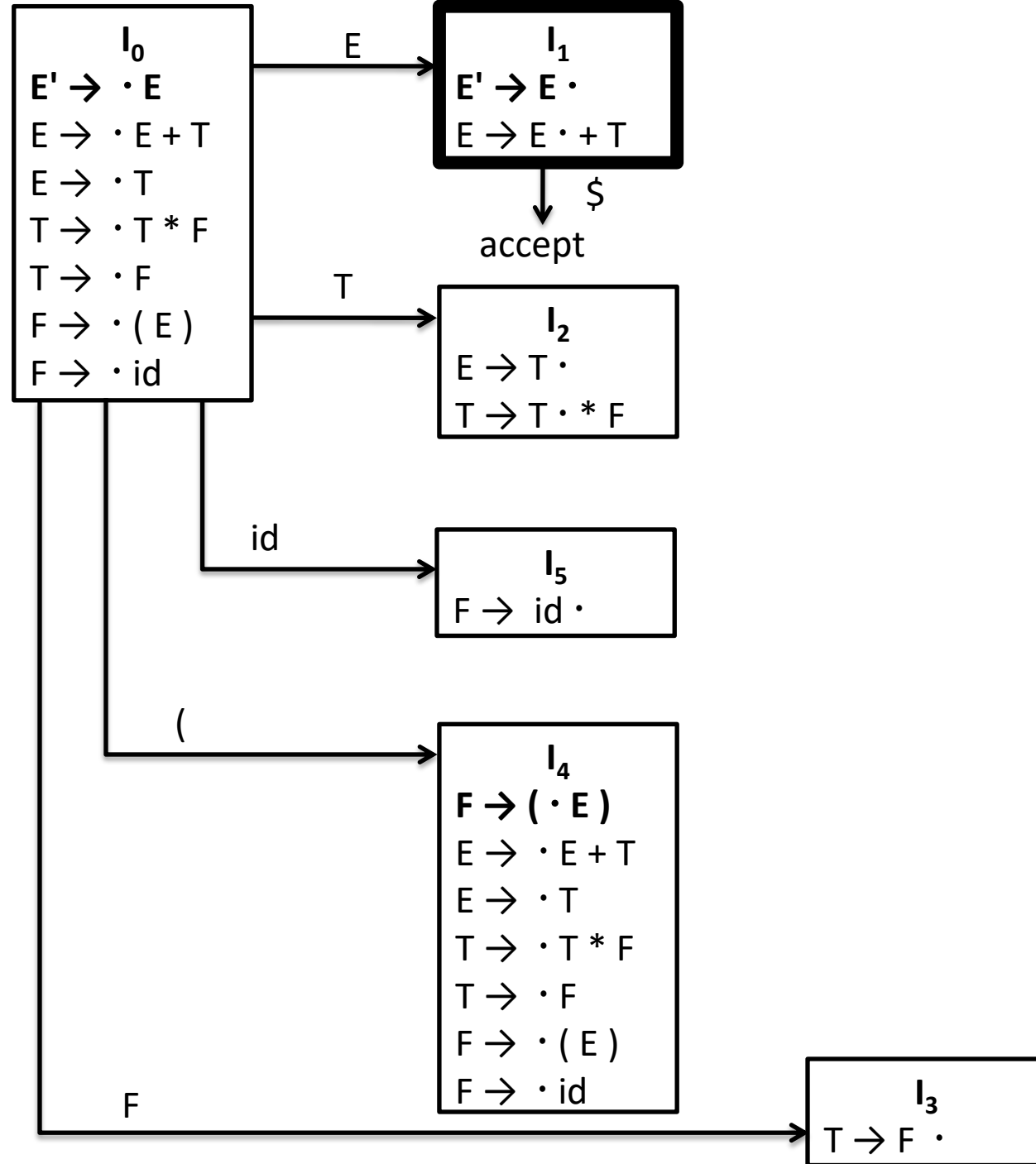


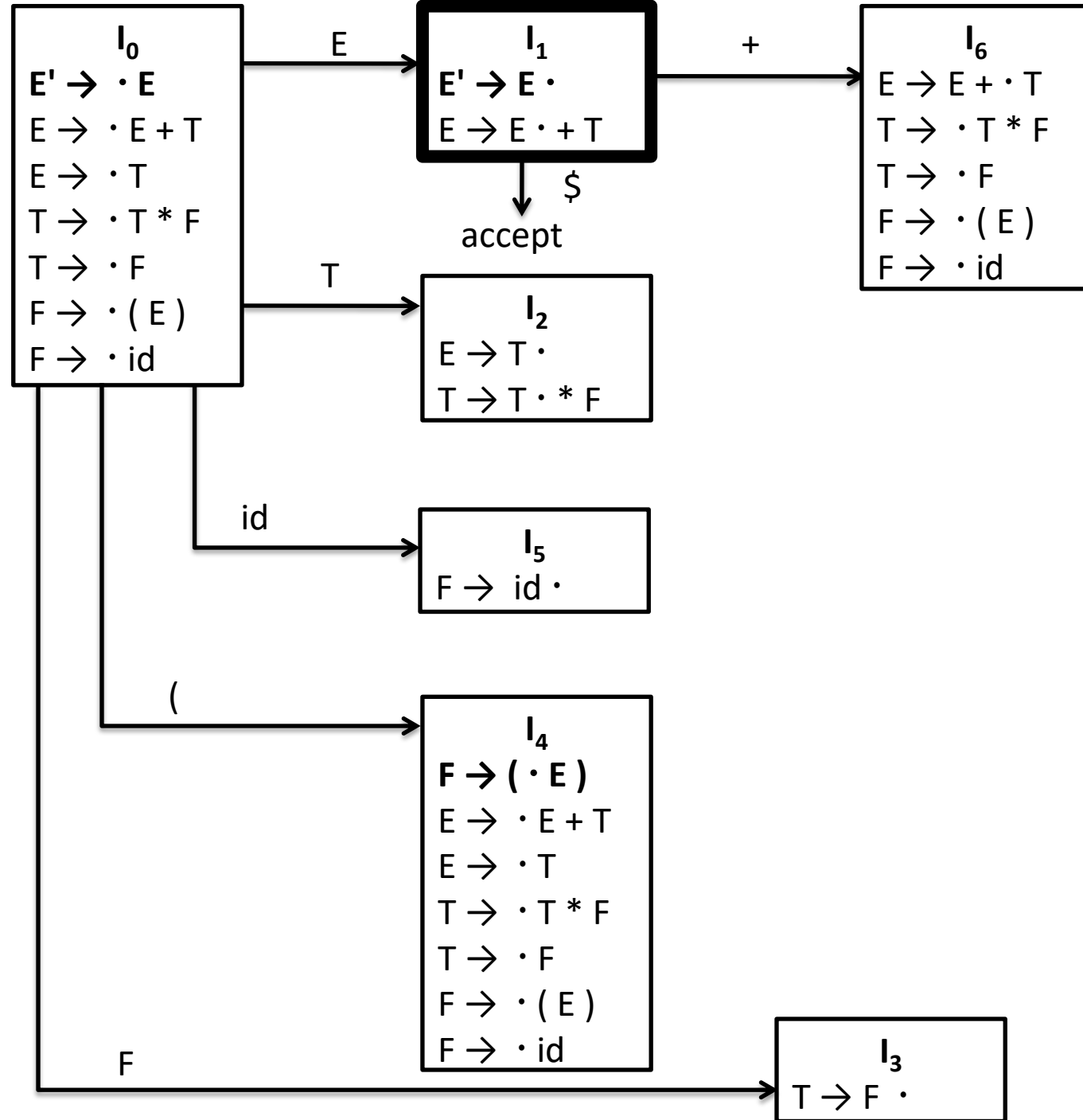


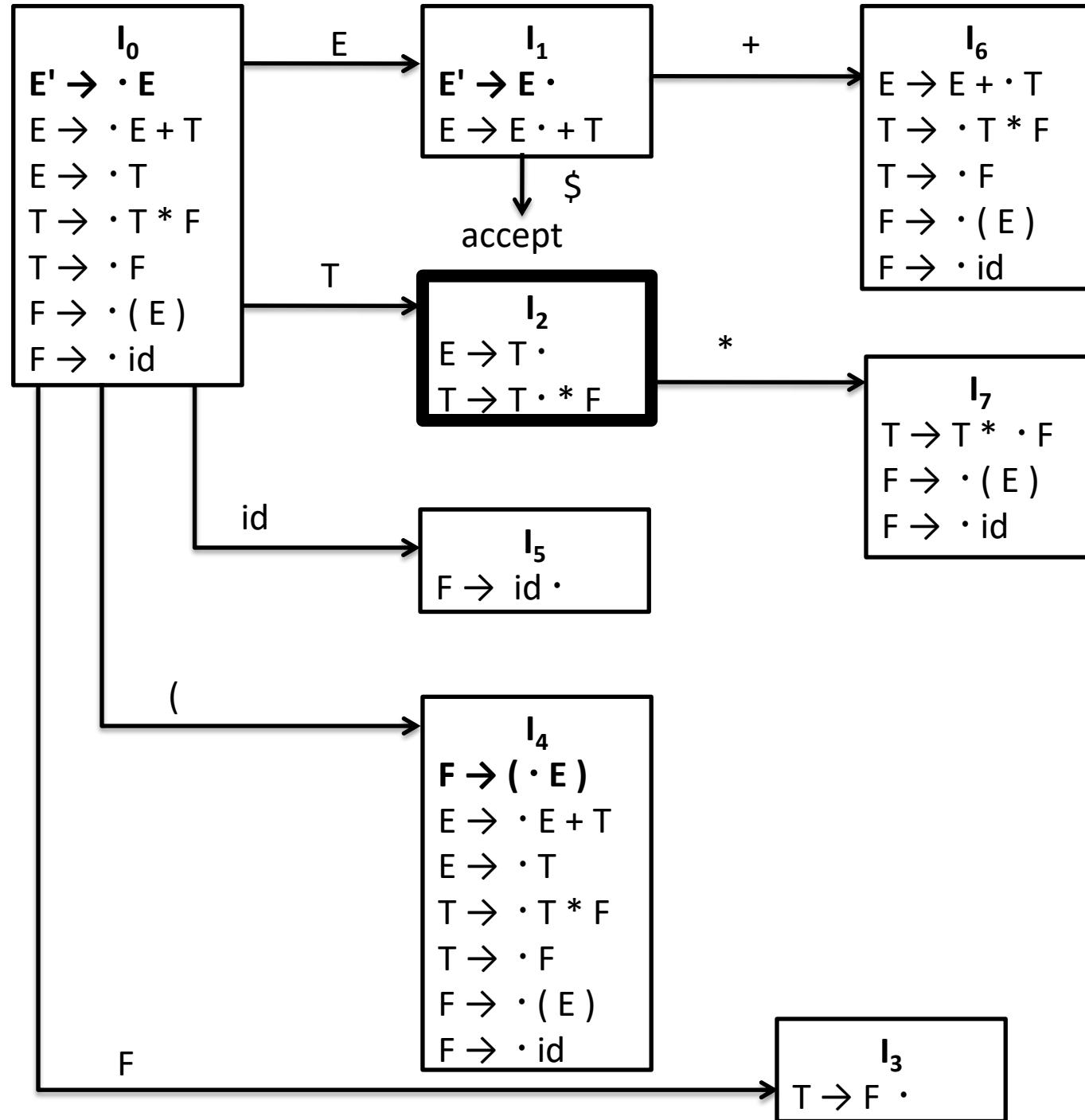


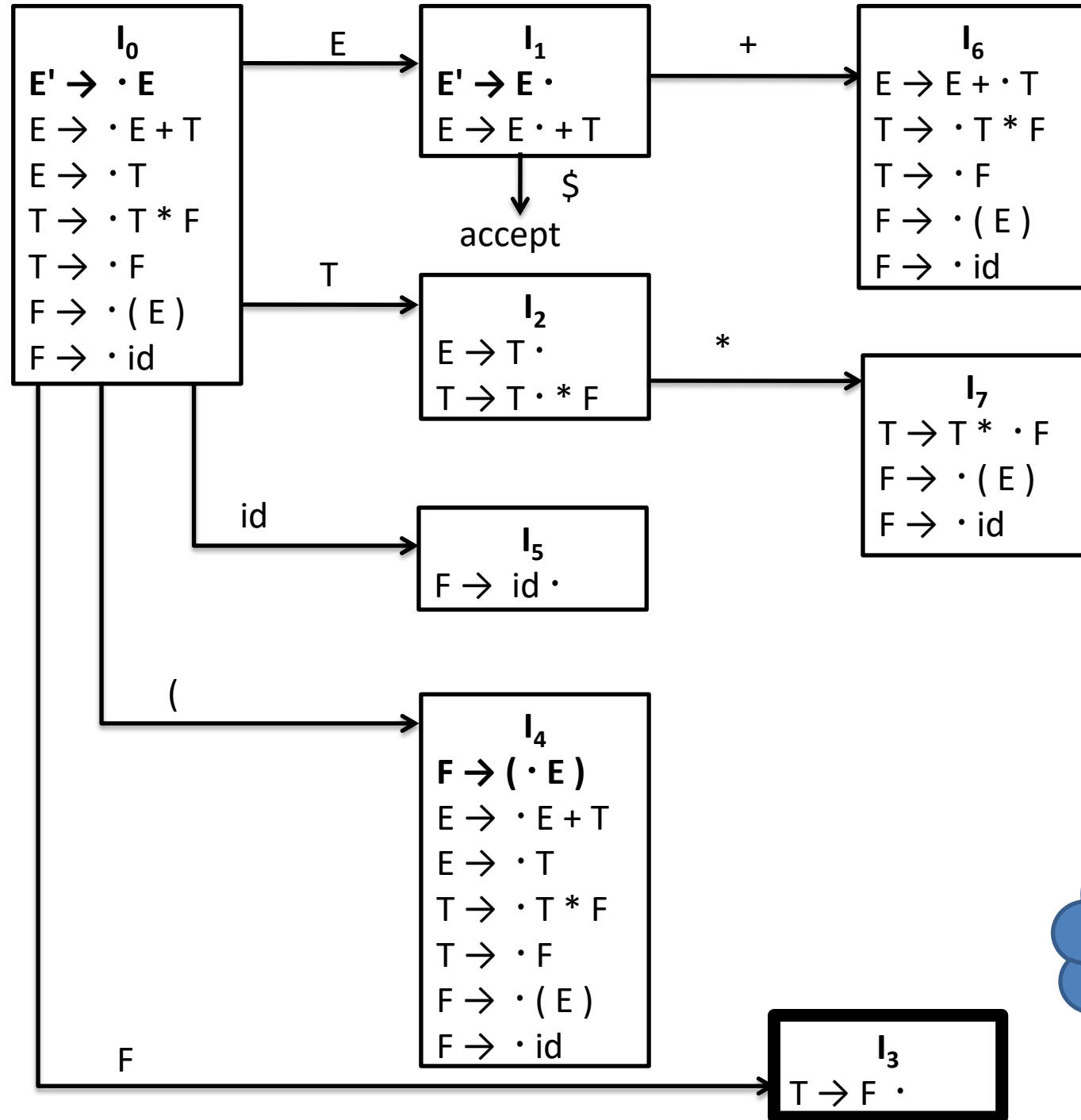




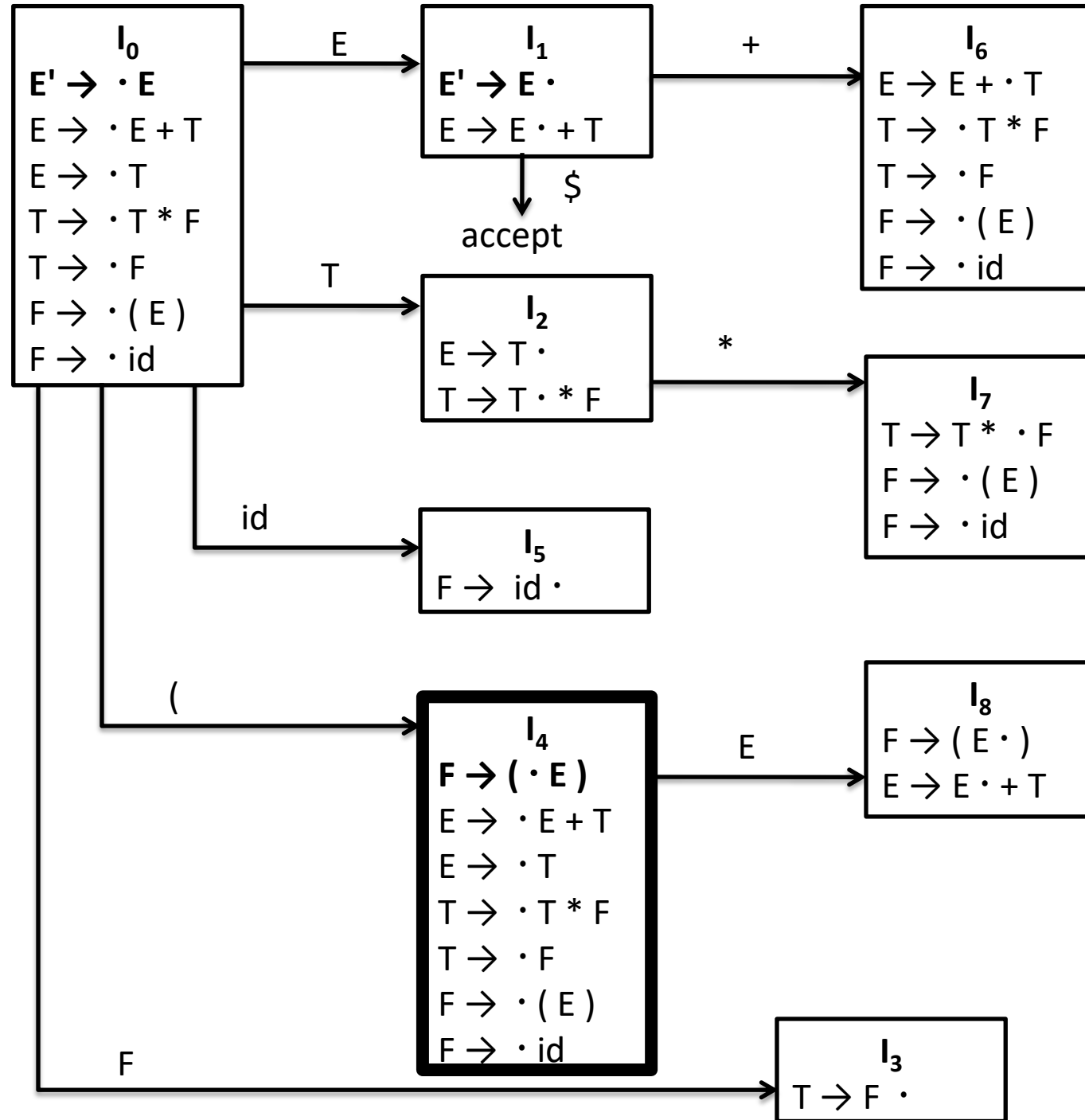


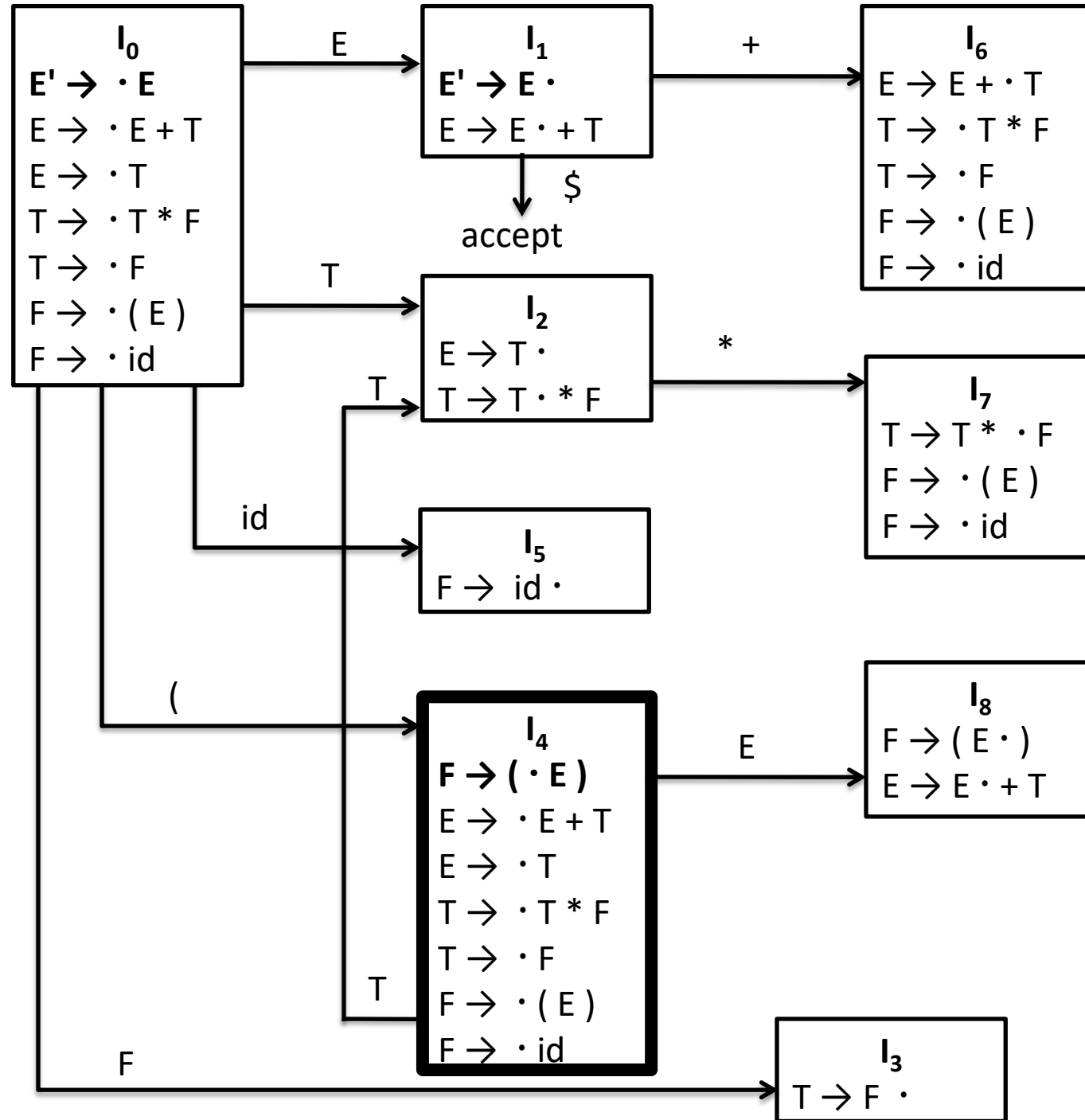




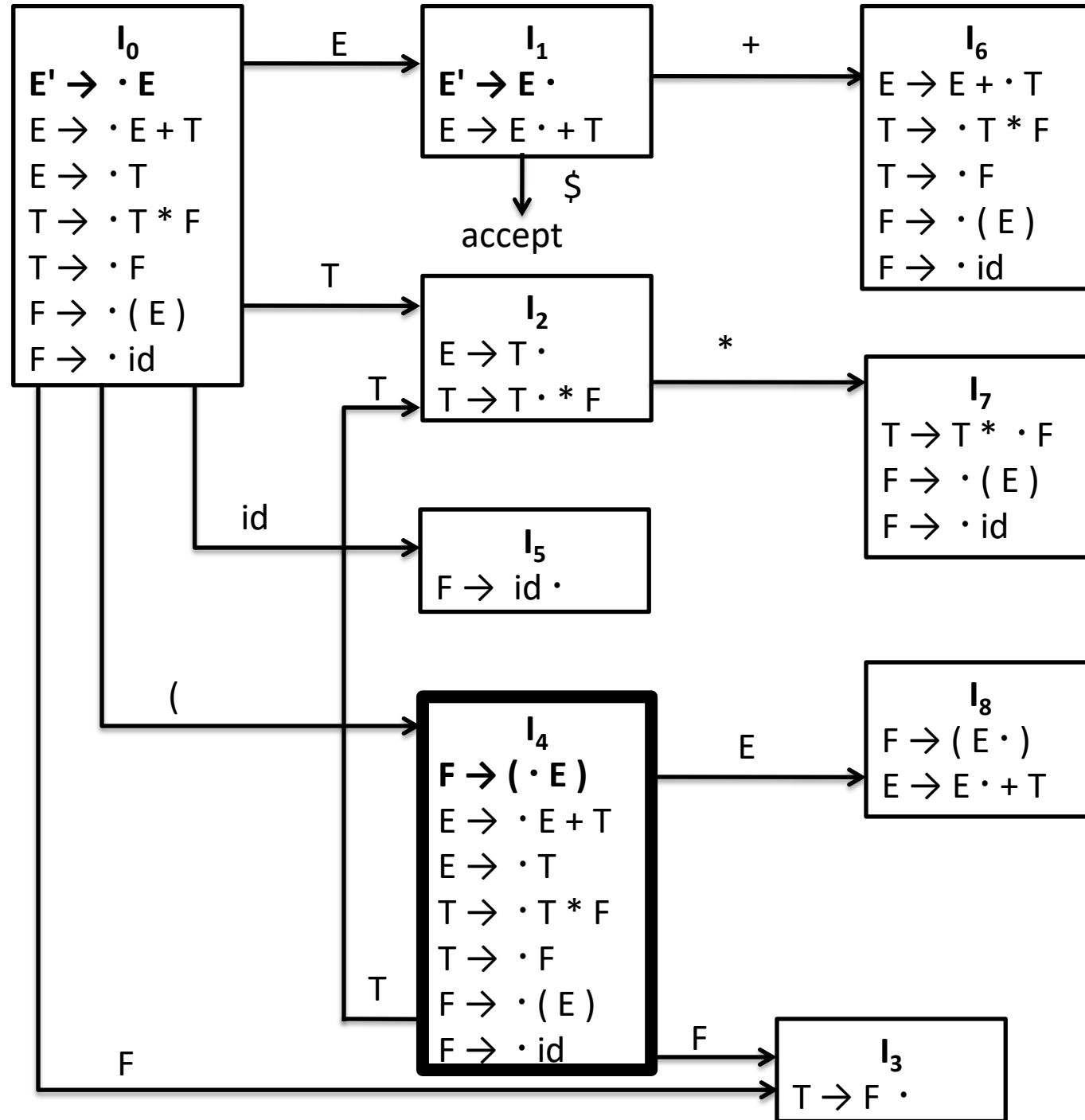


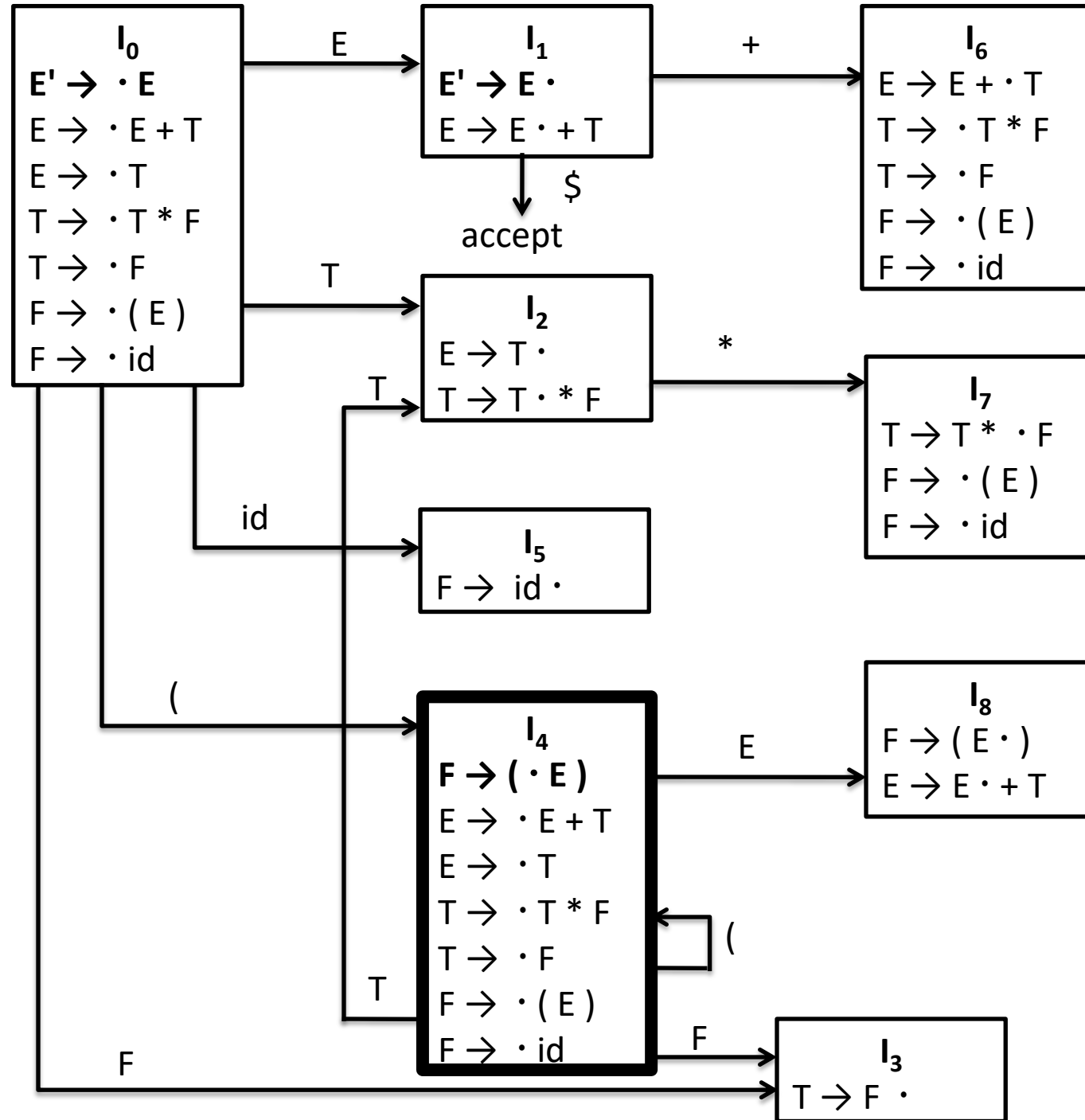
$I_3$  is complete

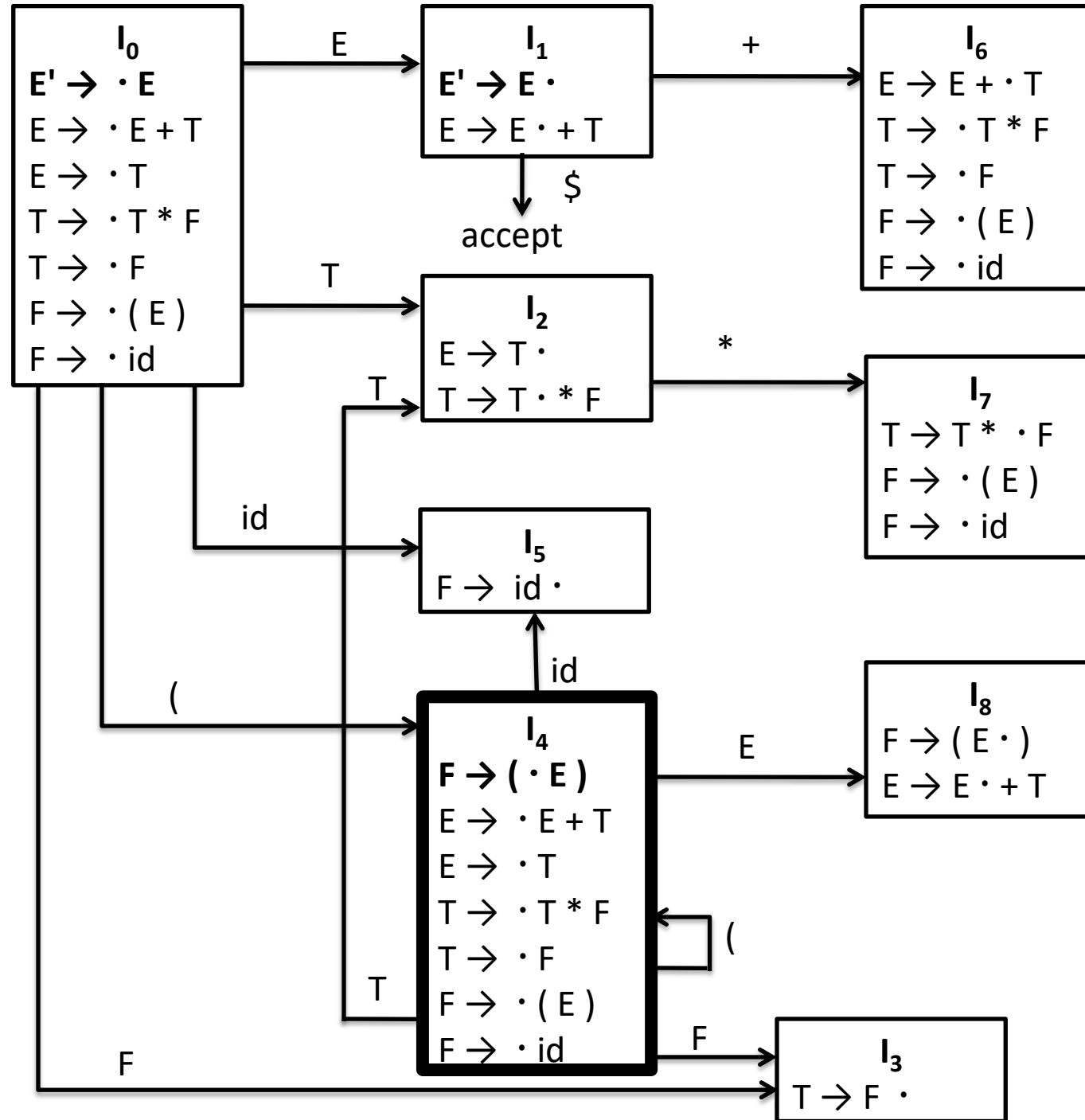


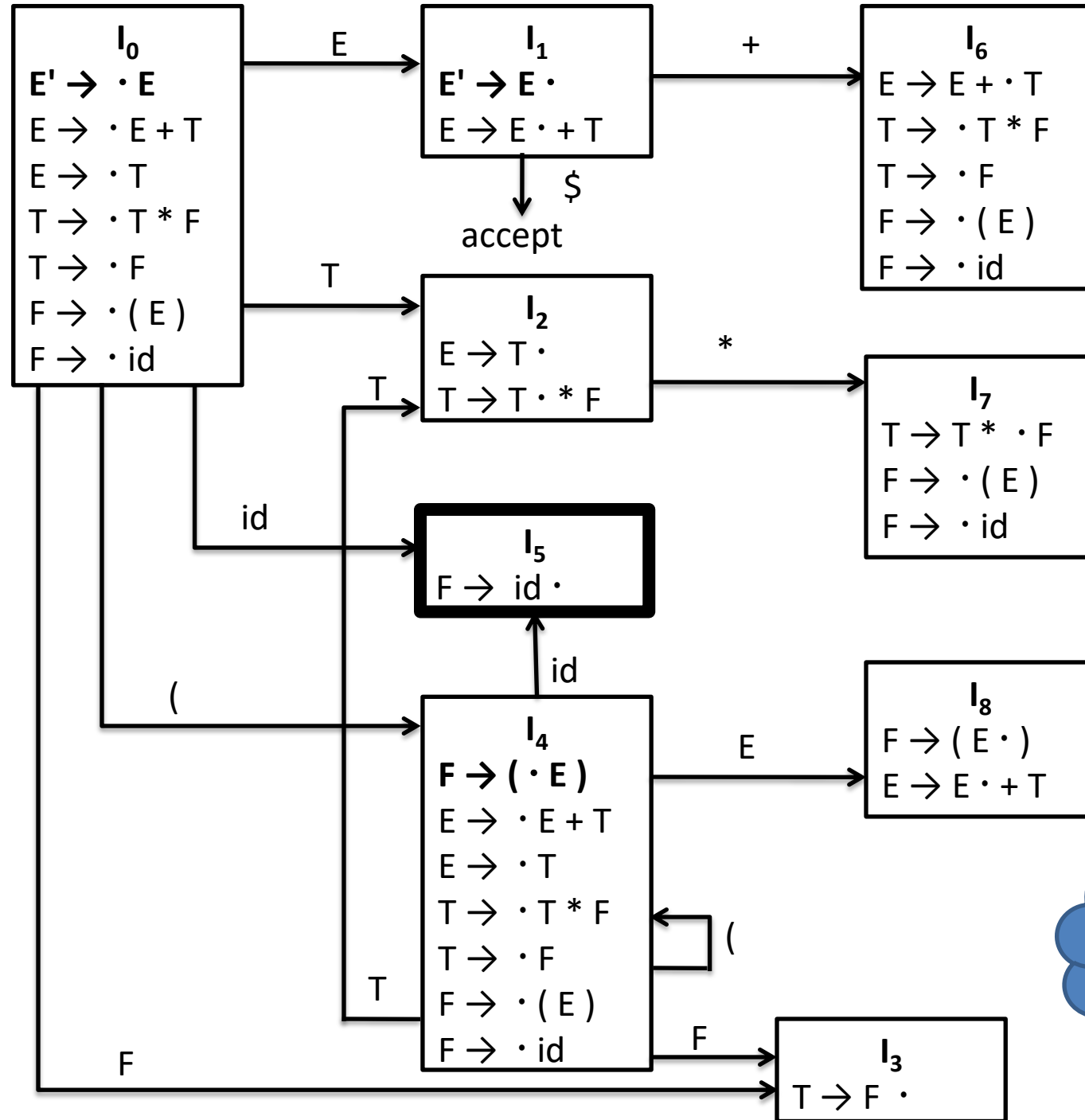




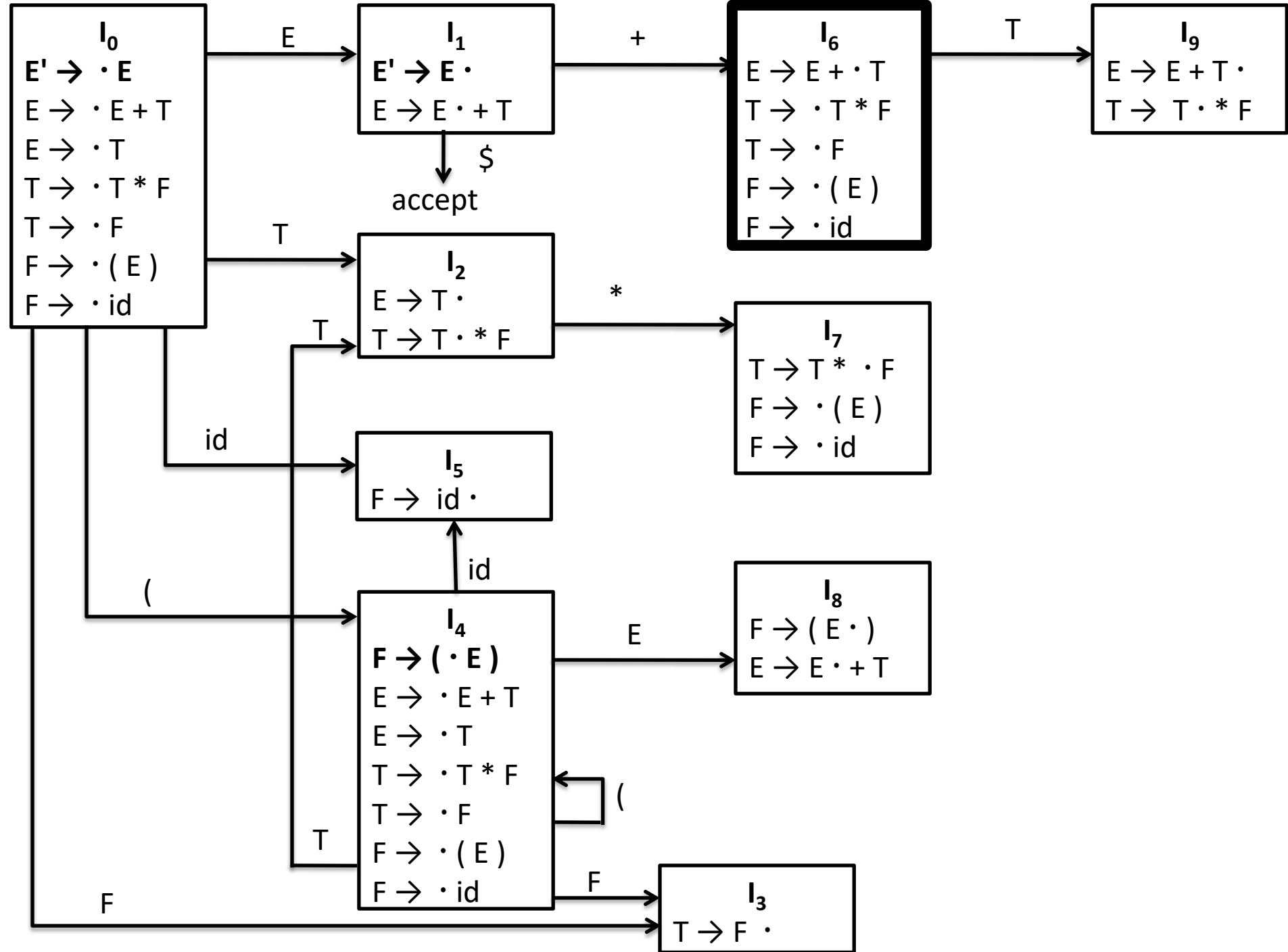


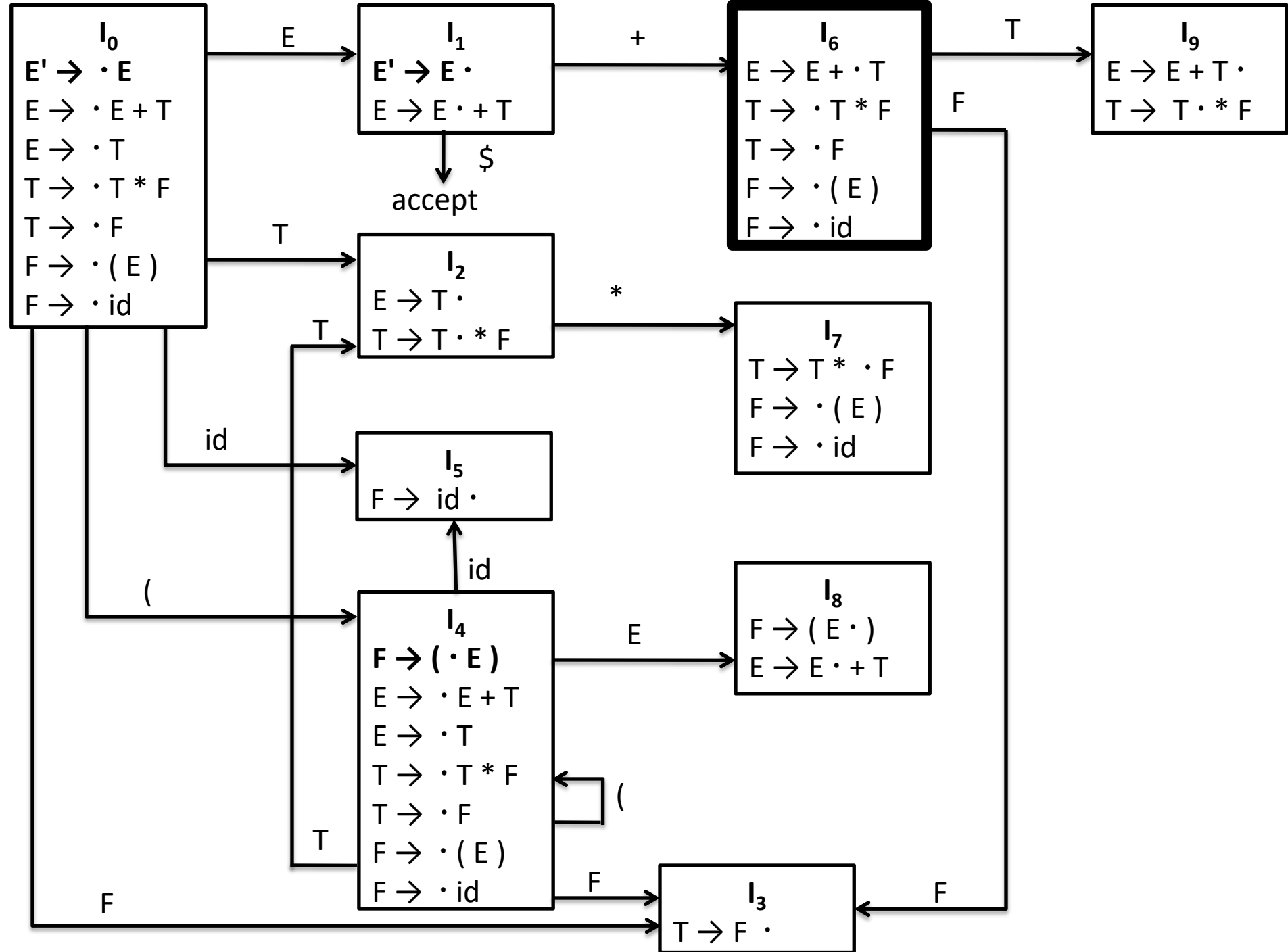


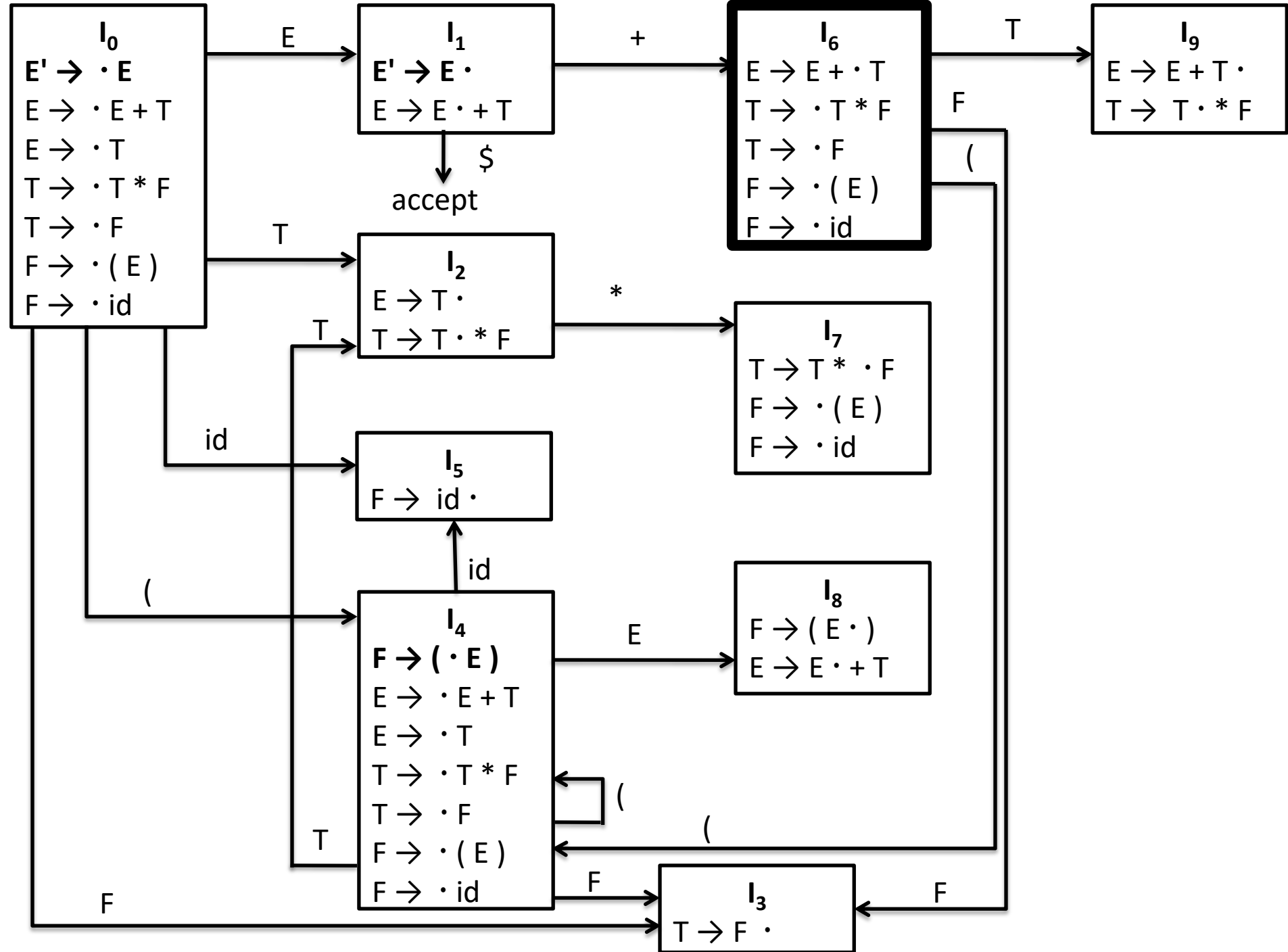


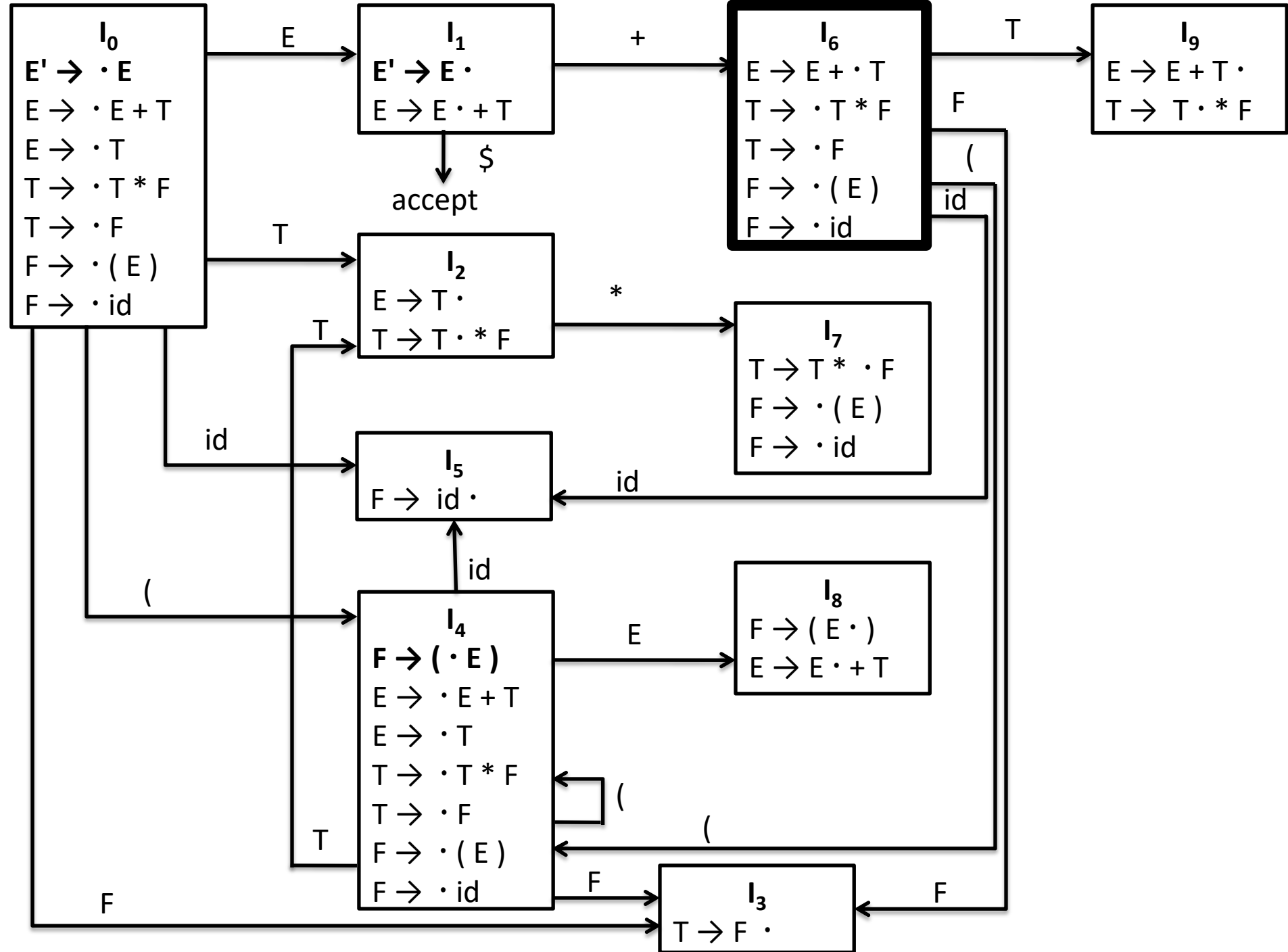


$I_5$  is complete

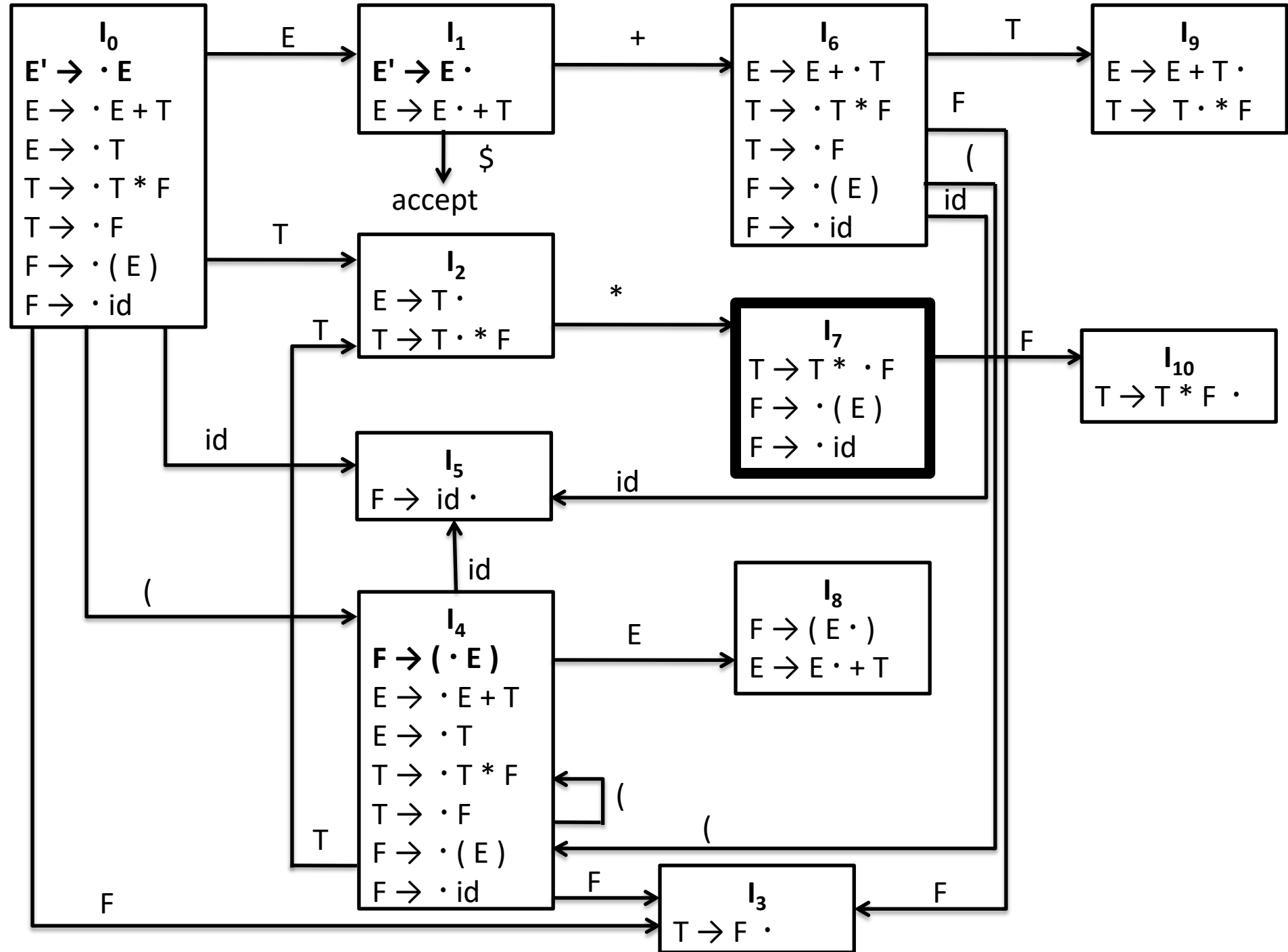


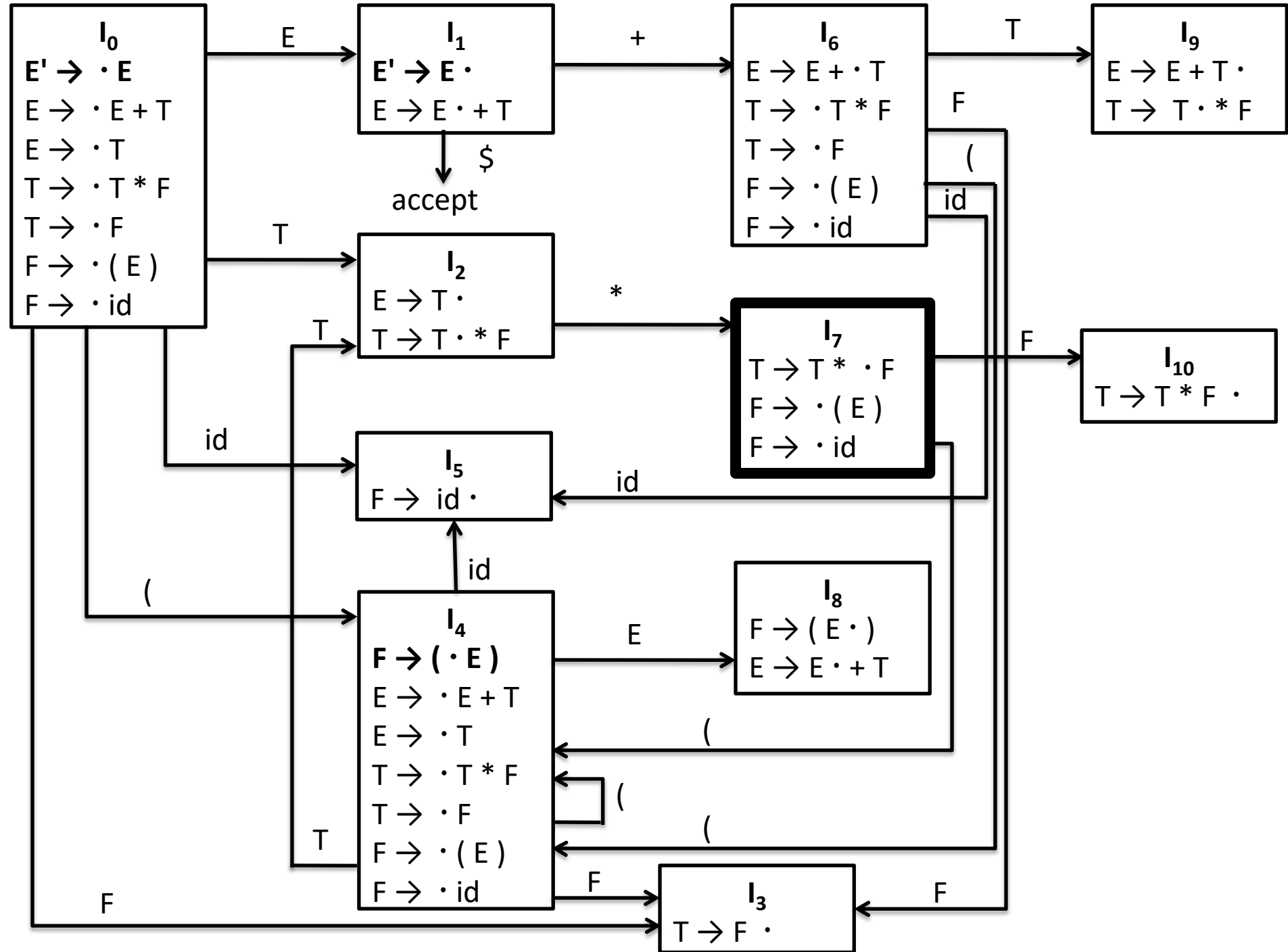


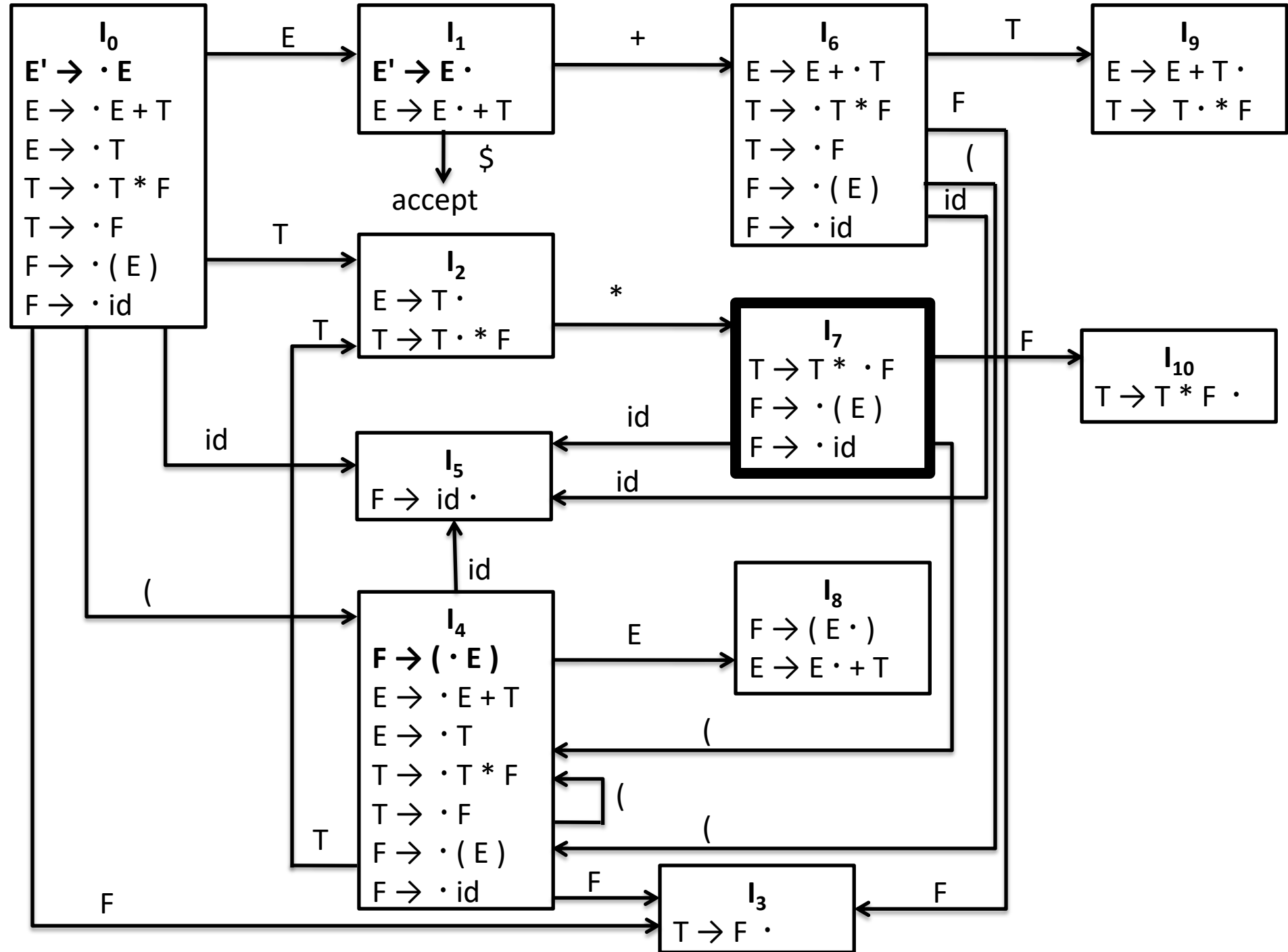


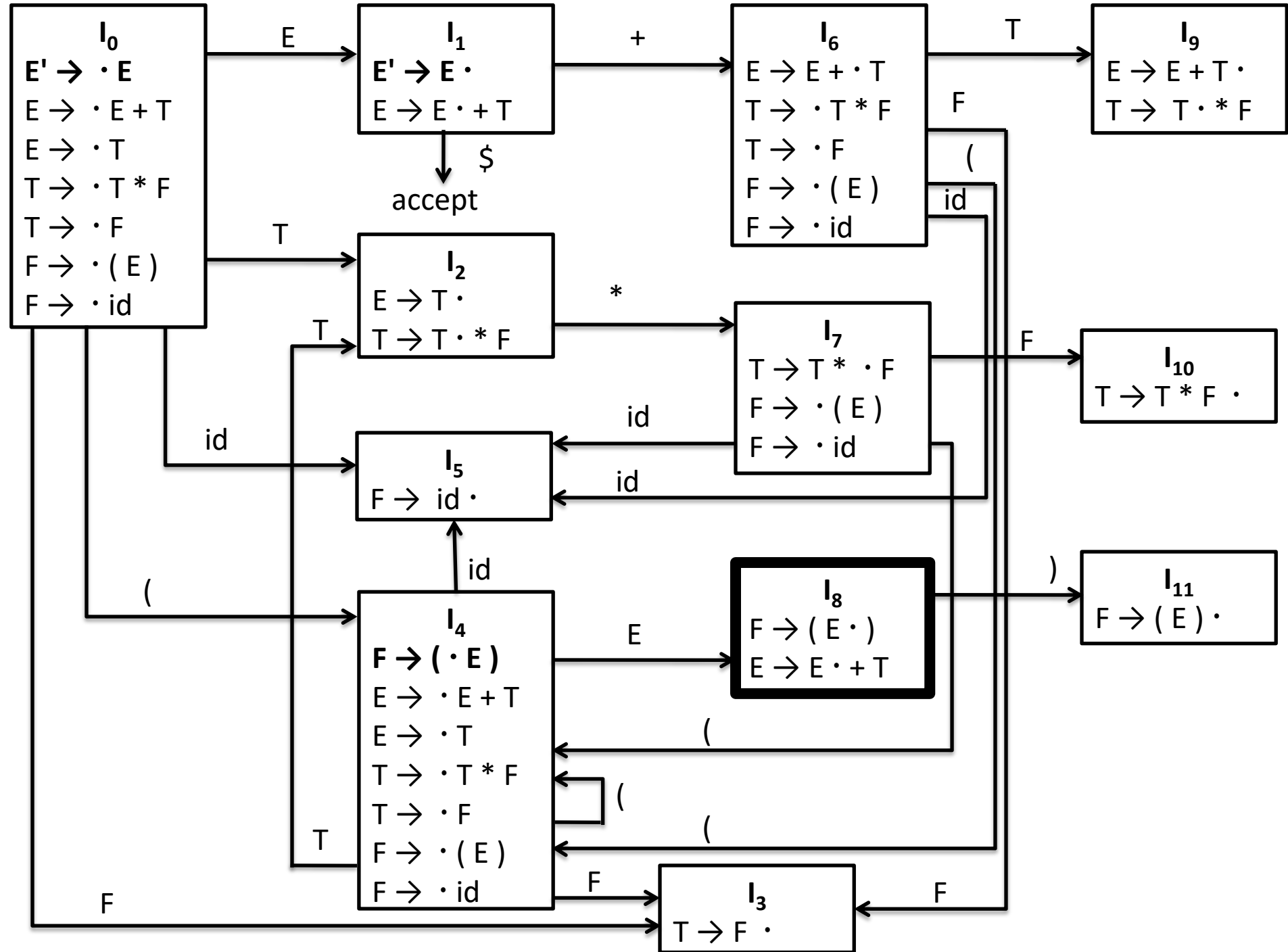


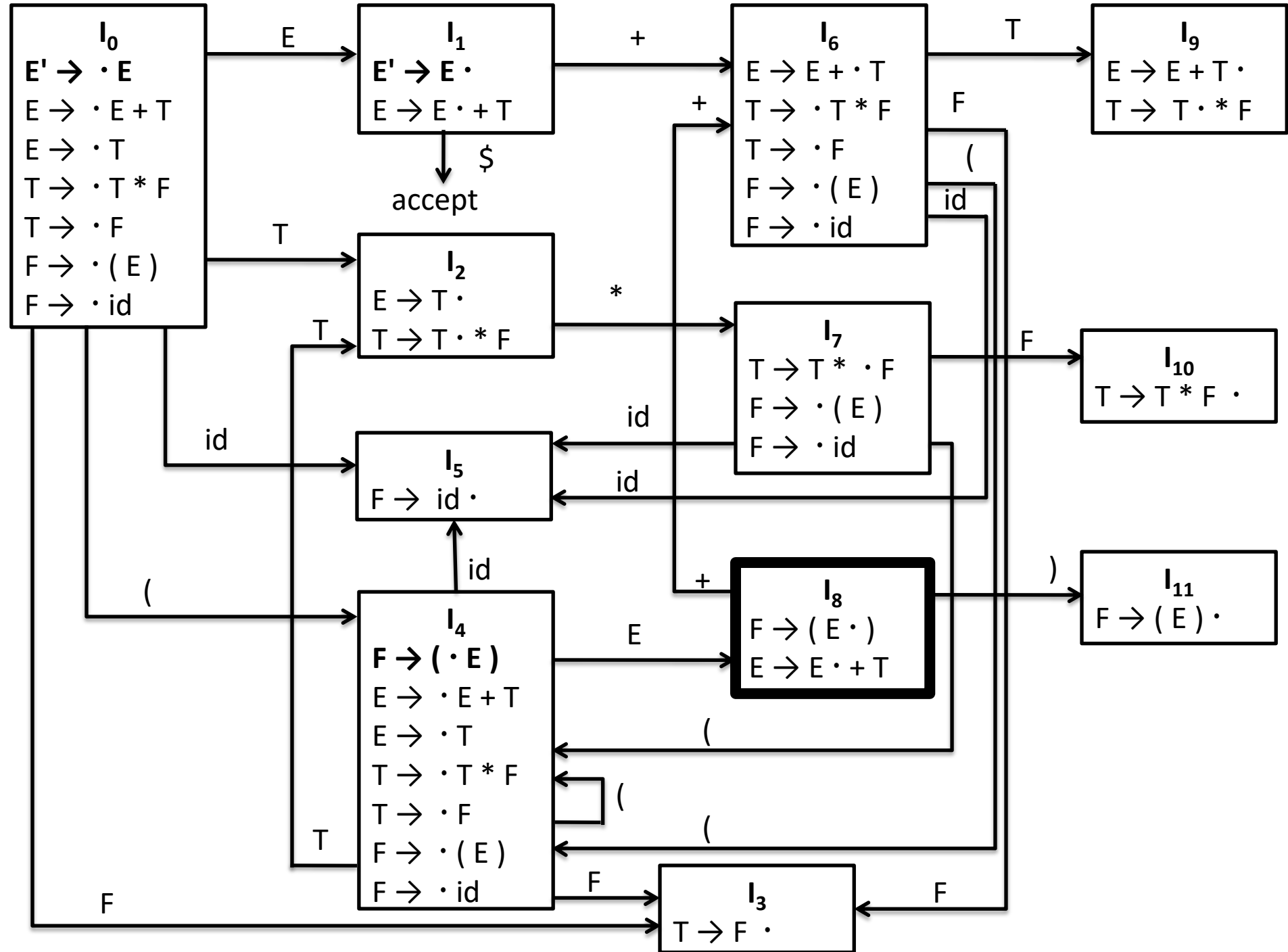


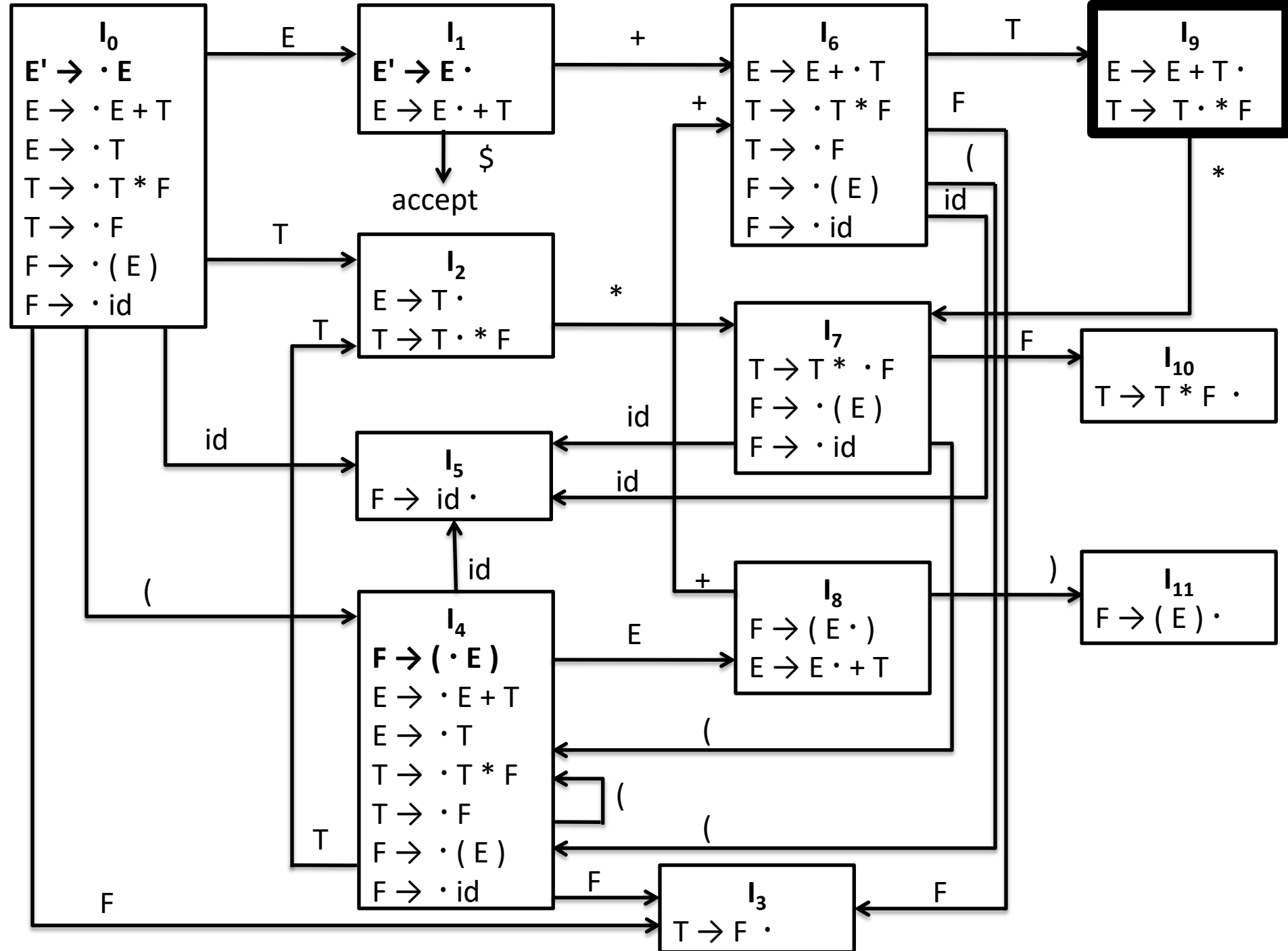


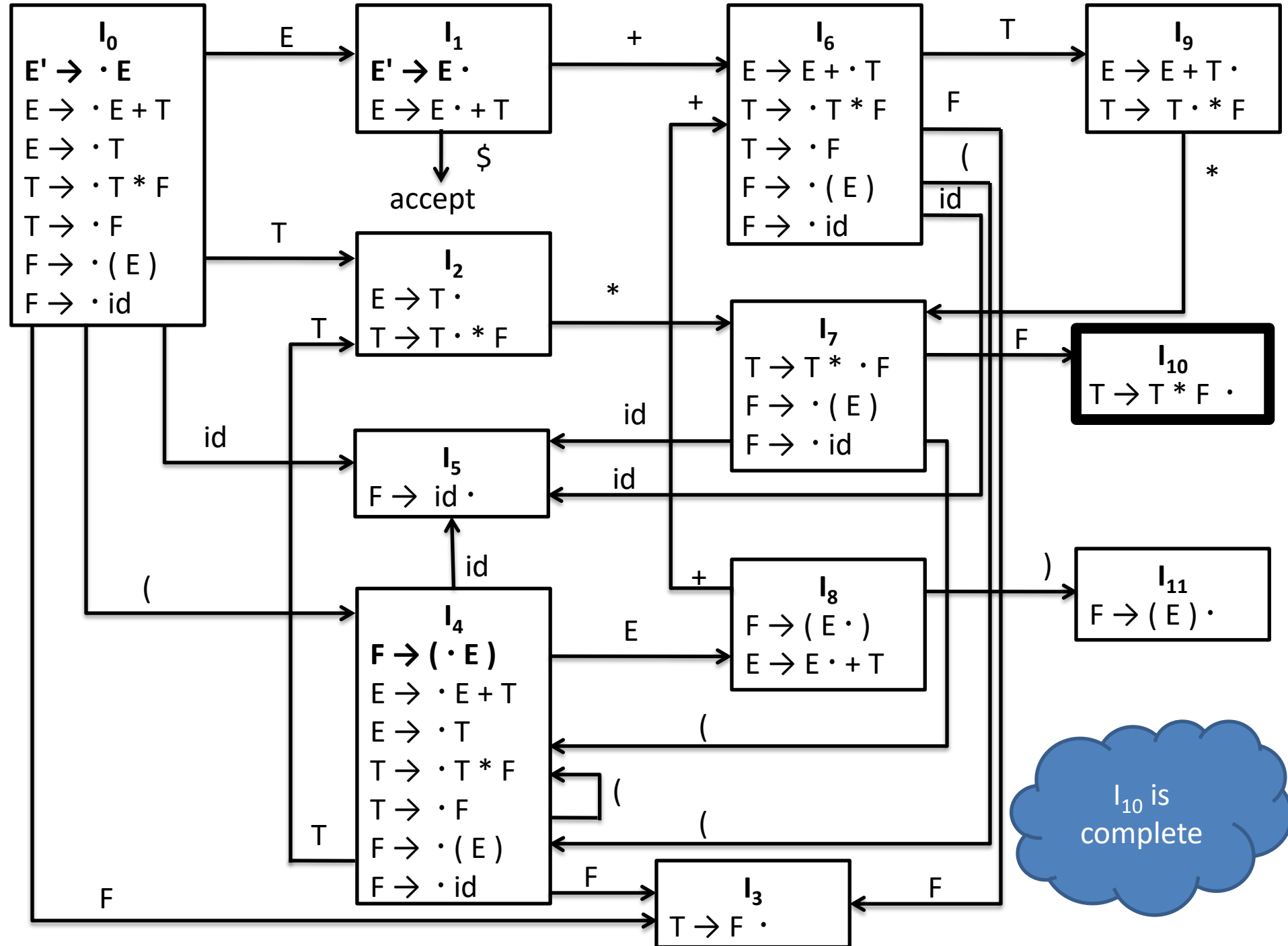


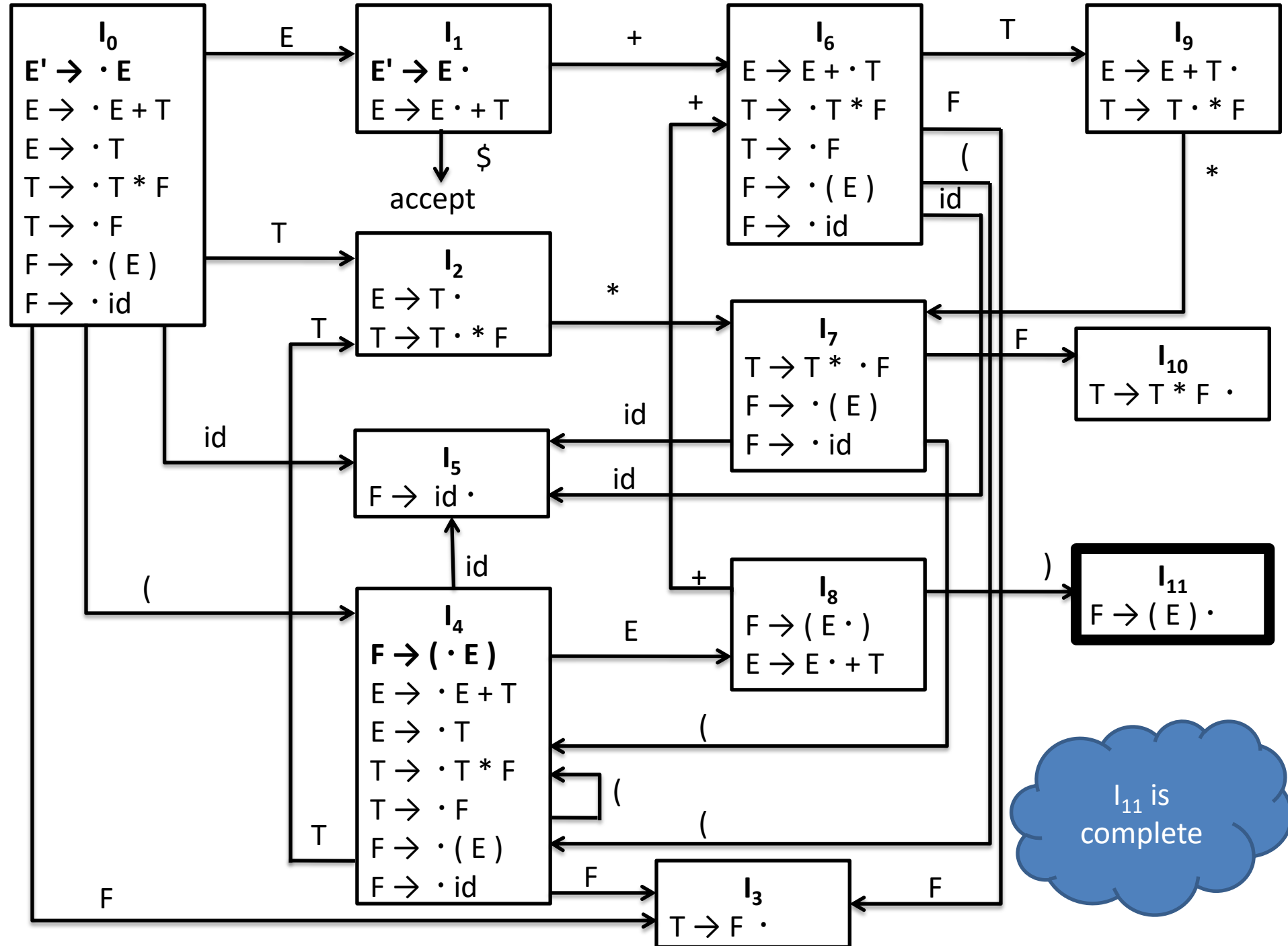




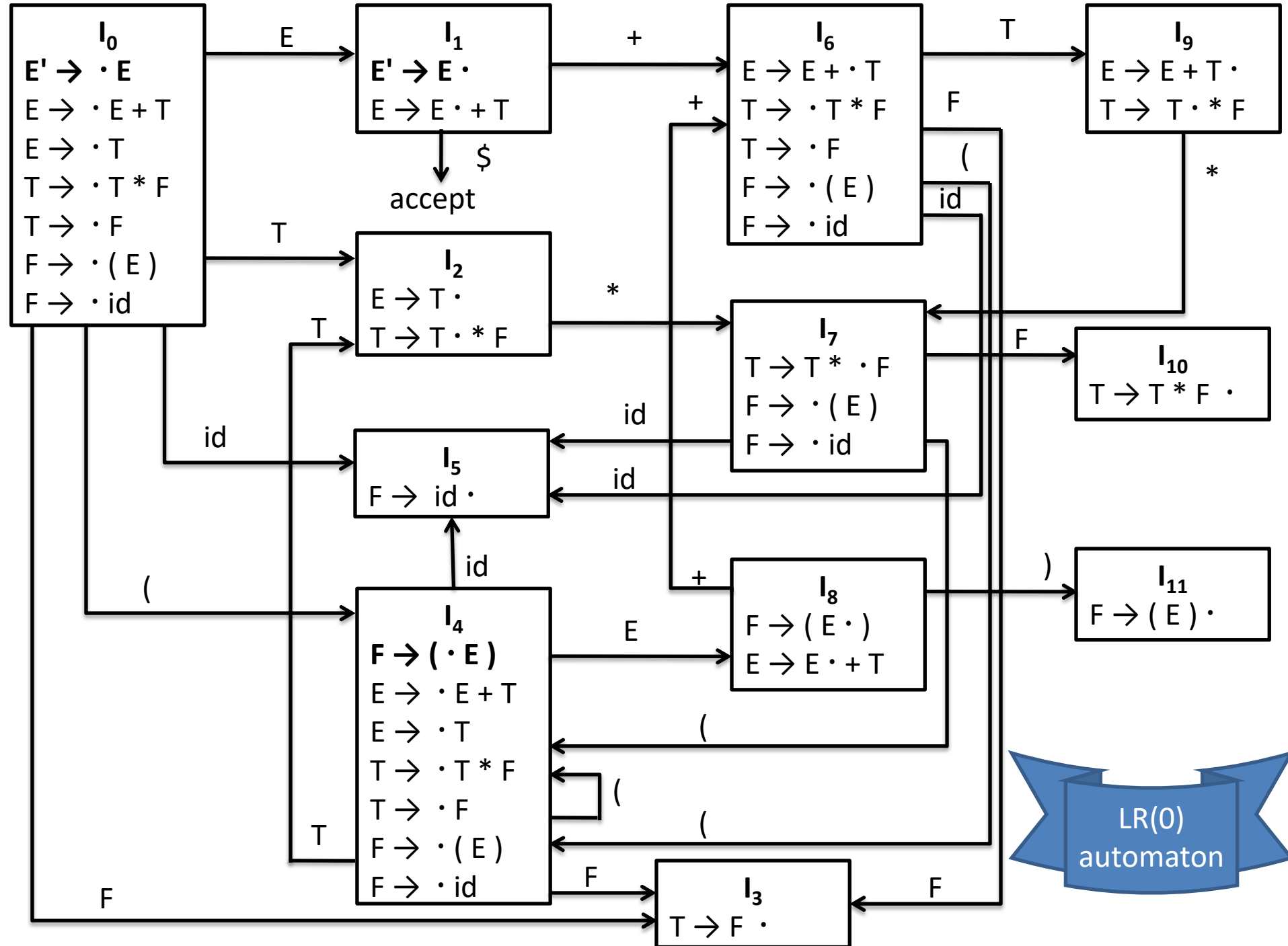












# Use of the LR(0) Automaton

- The central idea behind "Simple LR", or SLR, parsing is the construction from the grammar of the LR(0) automaton.
- The states of this automaton are the sets of items from the canonical LR(0) collection, and the transitions are given by the GOTO function.
- The start state of the LR(0) automaton is  $\text{CLOSURE}(\{[S' \rightarrow \cdot S]\})$ , where  $S'$  is the start symbol of the augmented grammar.
- All states are accepting states.
- We say "state  $j$ " to refer to the state corresponding to the set of items  $I_j$ .

# How can LR(0) automata help with shift-reduce decisions?

- Shift-reduce decisions can be made as follows.
  - Suppose that the string of grammar symbols takes the LR(0) automaton from the start state 0 to some state  $j$ .
  - Then, **shift** on next input symbol  $a$  **if state  $j$  has a transition on  $a$ .**
  - Otherwise, we choose to **reduce**; the items in state  $j$  will **tell us which production to use.**

The actions of a shift-reduce parser on input **id \* id**, using the LR(0) automaton

[illegible]

The actions of a shift-reduce parser on input **id \* id**, using the [LR\(0\) automaton](#)

LINE	STACK	SYMBOLS	INPUT	ACTION
(1)	0	\$	<b>id</b> * id \$	shift to 5
(2)	0 5	\$ id	* id \$	

The next input symbol is **id** and state 0 has a transition on id to state 5. We therefore shift.

The actions of a shift-reduce parser on input **id \* id**, using the [LR\(0\) automaton](#)

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 \$	

At line (2), state 5 (symbol id) has been pushed onto the stack. There is no transition from state 5 on input , so we reduce. From item  $[F \rightarrow id \cdot]$  in state 5, the reduction is by production  $F \rightarrow id$ .

With symbols, a reduction is implemented by popping the body of the production from the stack (on line (2), the body is id) and pushing the head of the production (in this case, F ). With states, we pop state 5 for symbol id, which brings state 0 to the top and look for a transition on F , the head of the production. In [LR\(0\) automaton](#), state 0 has a transition on F to state 3, so we push state 3, with corresponding symbol F ; see line (3).

The actions of a shift-reduce parser on input **id \* id**, using the [LR\(0\) automaton](#)

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 \$	

The actions of a shift-reduce parser on input **id \* id**, using the [LR\(0\) automaton](#)

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 \$	



The actions of a shift-reduce parser on input **id \* id**, using the [LR\(0\) automaton](#)

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	\$	

consider line (5), with state 7 (symbol ) on top of the stack. This state has a transition to state 5 on input id, so we push state 5 symbol id). State 5 has no transitions, so we reduce by  $F \rightarrow id$ . When we pop state 5 for the body id, state 7 comes to the top of the stack. Since state 7 as a transition on F to state 10, we push state 10 (symbol F ).

The actions of a shift-reduce parser on input **id \* id**, using the [LR\(0\) automaton](#)

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 5 10	\$ T * F	\$	

The actions of a shift-reduce parser on input **id \* id**, using the [LR\(0\) automaton](#)

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 5 10	\$ T * F	\$	reduce by $T \rightarrow T * F$
(8)	0 2	\$ T	\$	

The actions of a shift-reduce parser on input **id \* id**, using the [LR\(0\) automaton](#)

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 5 10	\$ T * F	\$	reduce by $T \rightarrow T * F$
(8)	0 2	\$ T	\$	reduce by $E \rightarrow T$
(9)	0 1	\$ E	\$	

The actions of a shift-reduce parser on input **id \* id**, using the [LR\(0\) automaton](#)

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 5 10	\$ T * F	\$	reduce by $T \rightarrow T * F$
(8)	0 2	\$ T	\$	reduce by $E \rightarrow T$
(9)	0 1	\$ E	\$	accept

# LR-Parsing

- The driver program is the same for all LR parsers; only **the parsing table changes from one parser to another.**
- The parsing program reads characters from an input buffer one at a time.
- **Where a shift-reduce parser would shift a symbol, an LR parser shifts a state.**
- Each state summarizes the information contained in the stack below it. The stack holds a sequence of states,  $s_0s_1 \dots s_m$ , where  $s_m$  is on top.
- **In the SLR method, the stack holds states from the LR(0) automaton.**

# Structure of LR Parsing Table

The parsing table consists of two parts: a parsing-action function ACTION and a goto function GOTO.

1. The ACTION function takes as arguments a state  $i$  and a terminal  $a$  (or  $\$,$  the input endmarker). The value of  $\text{ACTION}[i, a]$  can have one of four forms:
  - a) Shift  $j$ , where  $j$  is a state. The action taken by the parser effectively shifts input  $a$  to the stack, but uses state  $j$  to represent  $a$ .
  - b) Reduce  $A \rightarrow \beta$ . The action of the parser effectively reduces on the top of the stack to head  $A$ .
  - c) Accept. The parser accepts the input and finishes parsing.
  - d) Error. The parser discovers an error in its input and takes some corrective action.
2. We extend the GOTO function, defined on sets of items, to states: if  $\text{GOTO}[I_i, A] = I_j$ , then GOTO also maps a state  $i$  and a nonterminal  $A$  to state  $j$ .

# LR Parsing Algorithm

INPUT: An input string  $w$  and an LR-parsing table with functions ACTION and GOTO for a grammar.

OUTPUT: If  $w$  is in  $L(G)$ , the reduction steps of a bottom-up parse for  $w_i$  otherwise, an error indication.

METHOD: Initially, the parser has  $s_0$  on its stack, where  $s_0$  is the initial state, and  $w\$$  in the input buffer.



# 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  $A \rightarrow \beta$  ) {  
        pop  $|\beta|$  symbols off the stack;  
        let state  $t$  now be on top of the stack;  
        push GOTO[ $t, A$ ] onto the stack;  
        output the production  $A \rightarrow \beta$ ;  
    } else if ( ACTION[ $s, a$ ] = accept ) break; /* parsing is done */  
    else call error-recovery routine;  
}
```

# Constructing an SLR-parsing table

INPUT: An augmented grammar  $G'$ .

OUTPUT: The SLR-parsing table functions ACTION and GOTO for  $G'$ .

METHOD:

1. Construct  $C = \{ I_0, I_1, \dots, I_n \}$ , 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  $[A \rightarrow \alpha \cdot a \beta]$  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  $[A \rightarrow \alpha \cdot]$  is in  $I_i$ , then set  $\text{ACTION}[i, a]$  to "reduce  $A \rightarrow \alpha$ " for all  $a$  in  $\text{FOLLOW}(A)$ ; here  $A$  may not be  $S'$ .
  - c) If  $[S \rightarrow S' \cdot]$  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  $[S \rightarrow S' \cdot]$ .

































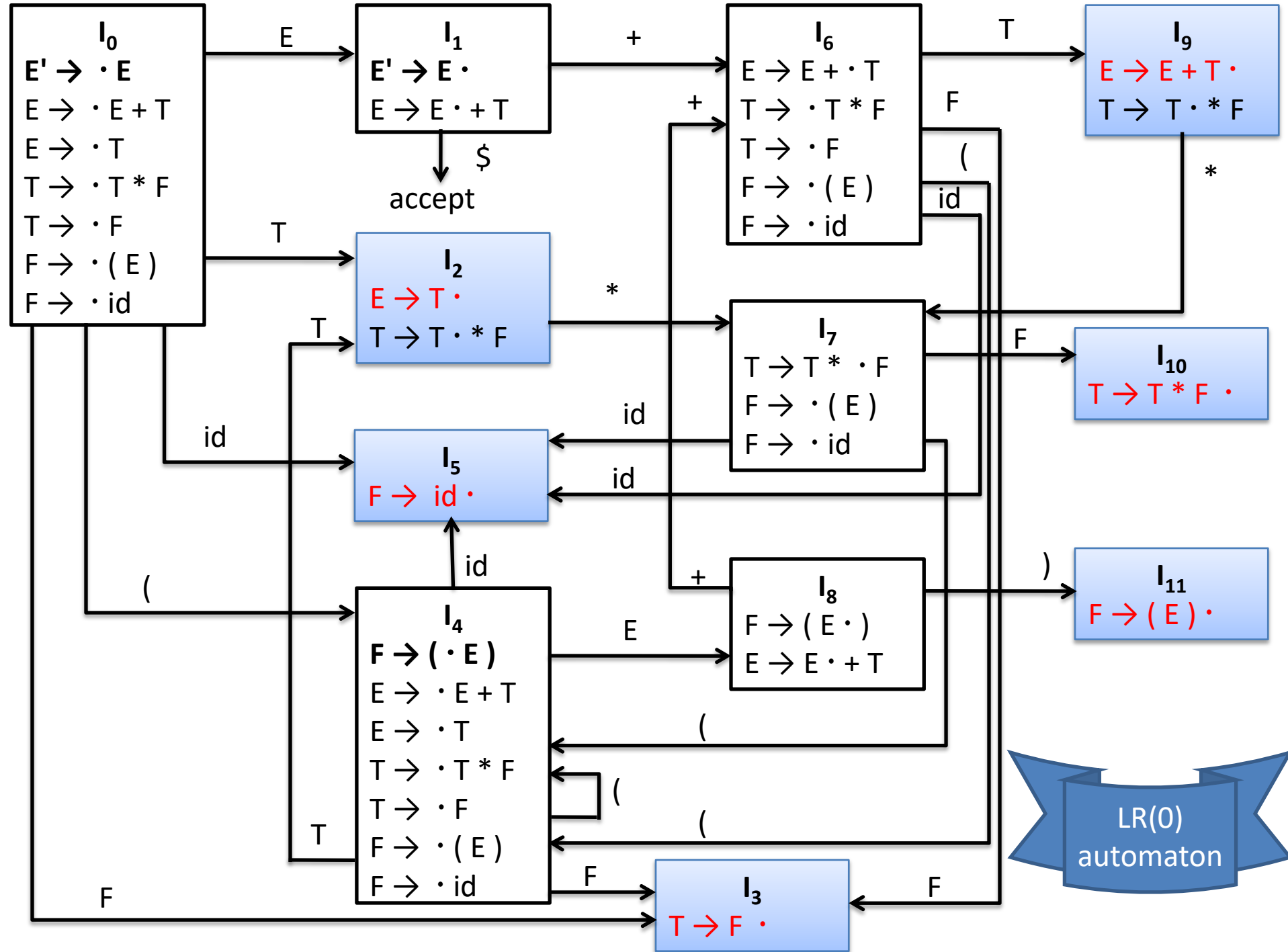














1.  $E \rightarrow E + T$
2.  $E \rightarrow T$
3.  $T \rightarrow T * F$
4.  $T \rightarrow F$
5.  $F \rightarrow ( E )$
6.  $F \rightarrow \text{id}$

# on grammar

[illegible]











$I_{11}$   
 $F \rightarrow ( E ) \cdot$

FOLLOW(E) = {+, ), \$}  
 FOLLOW(T) = {\*, +, ), \$}  
 FOLLOW(F) = {\*, +, ), \$}

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

# Parsing table for LR(0) automaton on grammar

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			

# Parsing table for expression grammar

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			

# Moves of an LR parser on 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.  $F \rightarrow id$

	STACK	SYMBOLS	INPUT	ACTION
1	0		id * id + id \$	Shift (s5)
2	0 5	id	* id + id \$	reduce by $F \rightarrow id$ (r6)
3	0 3 (0 $\rightarrow$ 3 has label F)	F	* id + id \$	reduce by $T \rightarrow F$ (r4)
4	0 2 (0 $\rightarrow$ 2 has label T)	T	* id + id \$	Shift (s7)
5	0 2 7	T *	id + id \$	Shift (s5)
6	0 2 7 5	T * id	+ id \$	reduce by $F \rightarrow id$ (r6)
7	0 2 7 10 (7 $\rightarrow$ 10 has label F)	T * F	+ id \$	reduce by $T \rightarrow T * F$ (r3)
8	0 2 (0 $\rightarrow$ 2 has label T)	T	+ id \$	reduce by $E \rightarrow T$ (r2)
9	0 1 (0 $\rightarrow$ 1 has label E)	E	+ id \$	Shift (s6)
10	0 1 6	E +	id \$	Shift (s5)
11	0 1 6 5	E + id	\$	reduce by $F \rightarrow id$ (r6)
12	0 1 6 3 (6 $\rightarrow$ 3 has label F)	E + F	\$	reduce by $T \rightarrow F$ (r4)
13	0 1 6 9 (6 $\rightarrow$ 9 has label T)	E + T	\$	reduce by $E \rightarrow E + T$ (r1)
14	0 1 (0 $\rightarrow$ 1 has label E)	E	\$	accept

Every SLR(1) grammar is unambiguous, but there are many unambiguous grammars that are not SLR(1).

- Consider the grammar with productions

$S \rightarrow L = R \mid R$

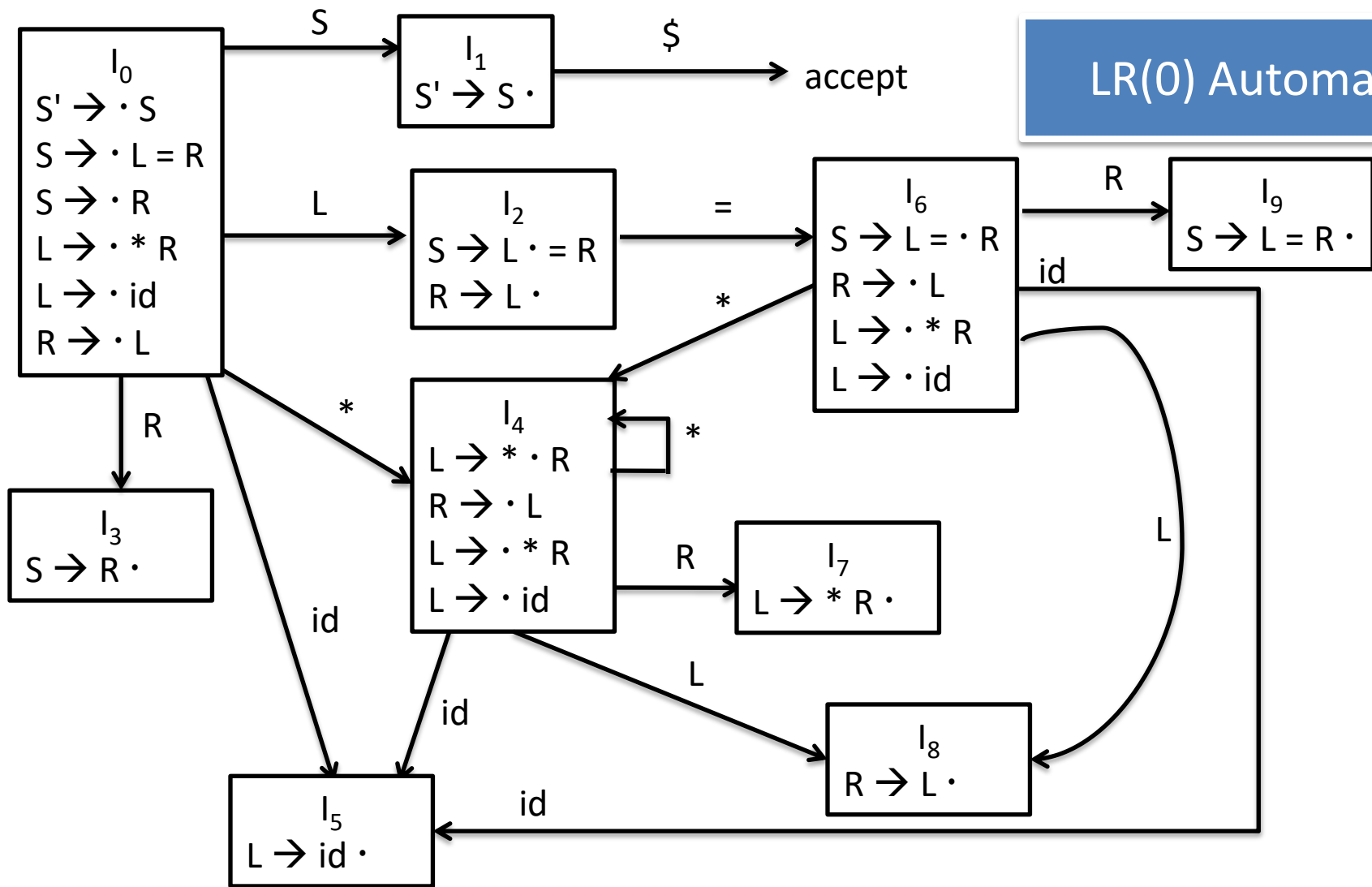
$L \rightarrow * R \mid \text{id}$

$R \rightarrow L$

- Find the Canonical LR(0) collection for grammar.
- Draw LR(0) automation
- Draw parse table with ACTION and GOTO



# LR(0) Automaton



# Parse Table

STATE	ACTION				GOTO		
	id	=	*	\$	S	L	R
0							
1							
2							
3							
4							
5							
6							
7							
8							
9							

# Parse Table (filling GOTO and shift)

STATE	ACTION				GOTO		
	id	=	*	\$	S	L	R
0	s5		s4		1	2	3
1				acc			
2		s6					
3							
4	s5		s4			8	7
5							
6	s5		s4			8	9
7							
8							
9							

$\text{FOLLOW}(S) = \{\$, \}$   
 $\text{FOLLOW}(L) = \{=, \$\}$   
 $\text{FOLLOW}(R) = \{=, \$\}$

# Parse Table (filling reduce)

STATE	ACTION				GOTO		
	id	=	*	\$	S	L	R
0	s5		s4		1	2	3
1				acc			
2		s6 / r5					
3				r2			
4	s5		s4			8	7
5		r4		r4			
6	s5		s4			8	9
7		r3		r3			
8		r5		r5			
9				r1			

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

# Parse Table

The grammar is not  
SLR(1) grammar.

STATE	ACTION				GOTO		
	id	=	*	.	S	L	R
0	s5		s4		1	2	3
1				acc			
2		s6 / r5					
3				r2			
4	s5		s4			8	7
5		r4		r4			
6	s5		s4			8	9
7		r3		r3			
8		r5		r5			
9				r1			

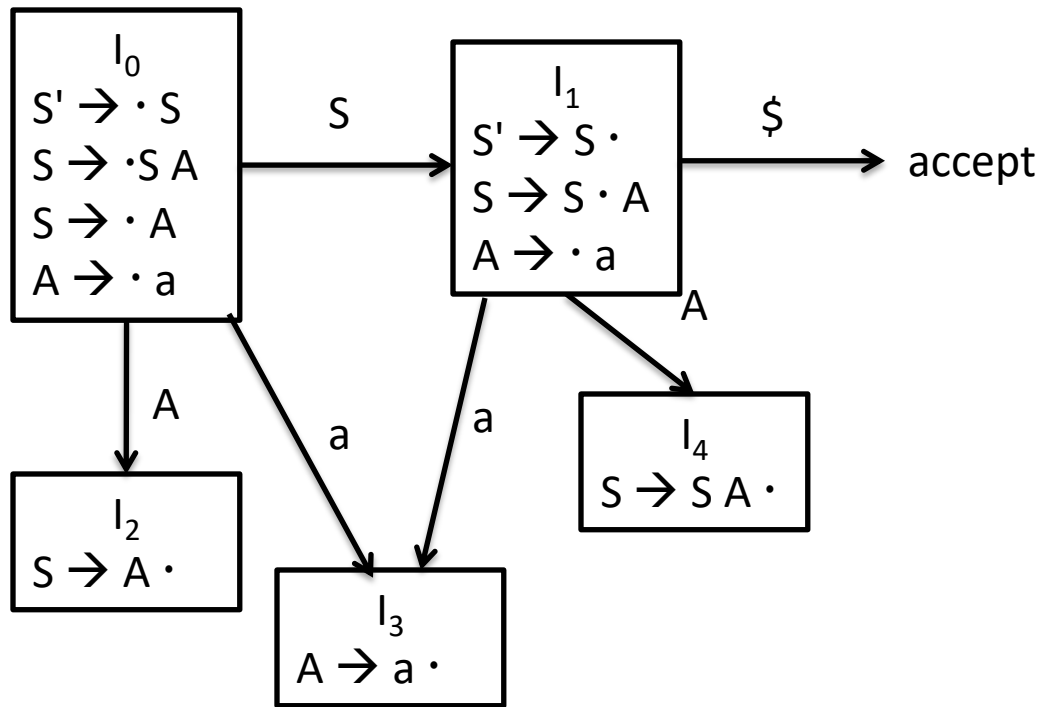
Show that the given grammar is SLR(1) but not LL(1)

$S \rightarrow SA \mid A$

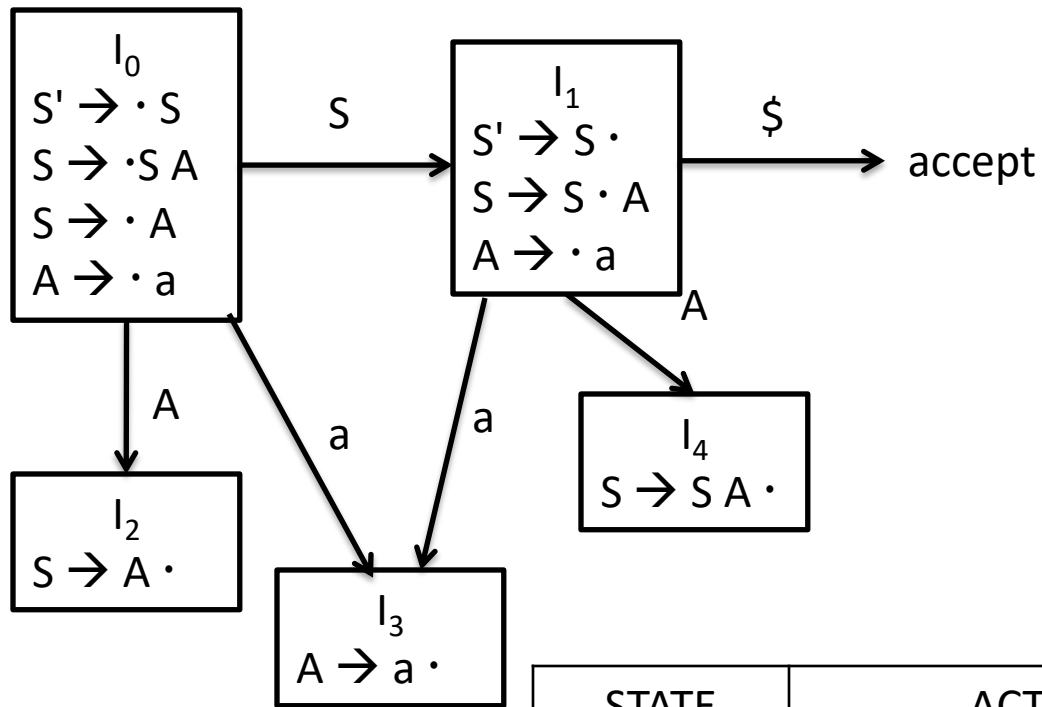
$A \rightarrow a$

- For a grammar to be LL(1), it must not be left recursive or ambiguous.
  - But the given grammar is left recursive so not LL(1).  $S \rightarrow SA$
- For a grammar to be SLR(1), construct LR(0) automaton and parsing table. And there must be no conflict in any entry.

# LR(0) Automaton



# LR(0) Automaton

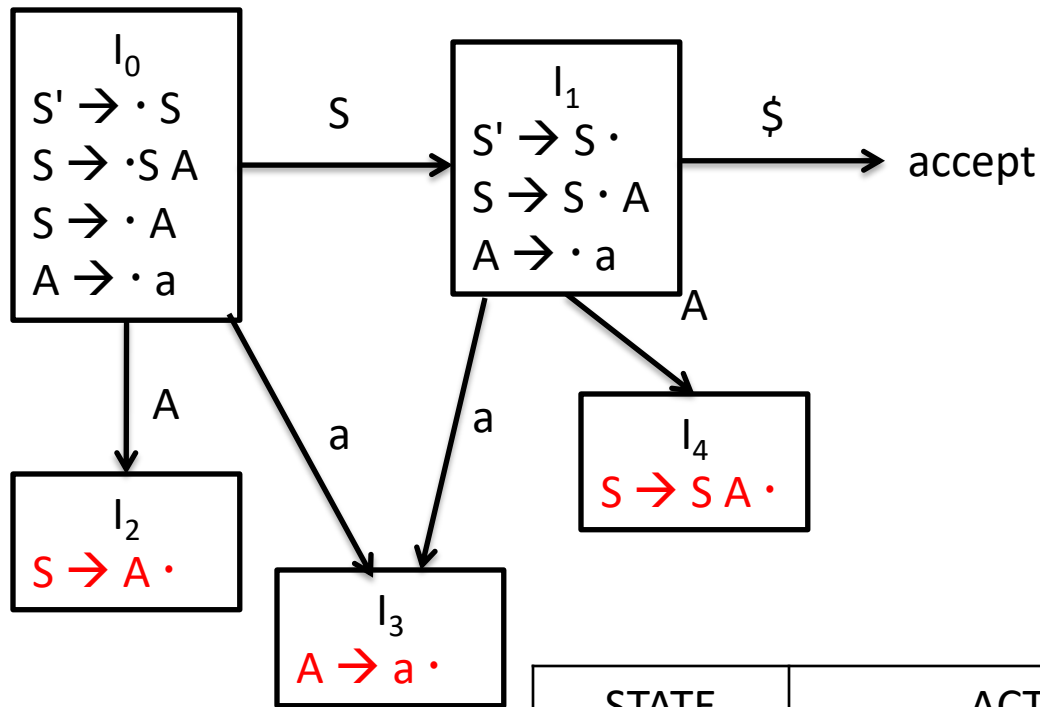


Filling shift and goto entries

STATE	ACTION		GOTO	
	a	\$	S	A
0	s3		1	2
1	s3	acc		4
2				
3				
4				



# LR(0) Automaton



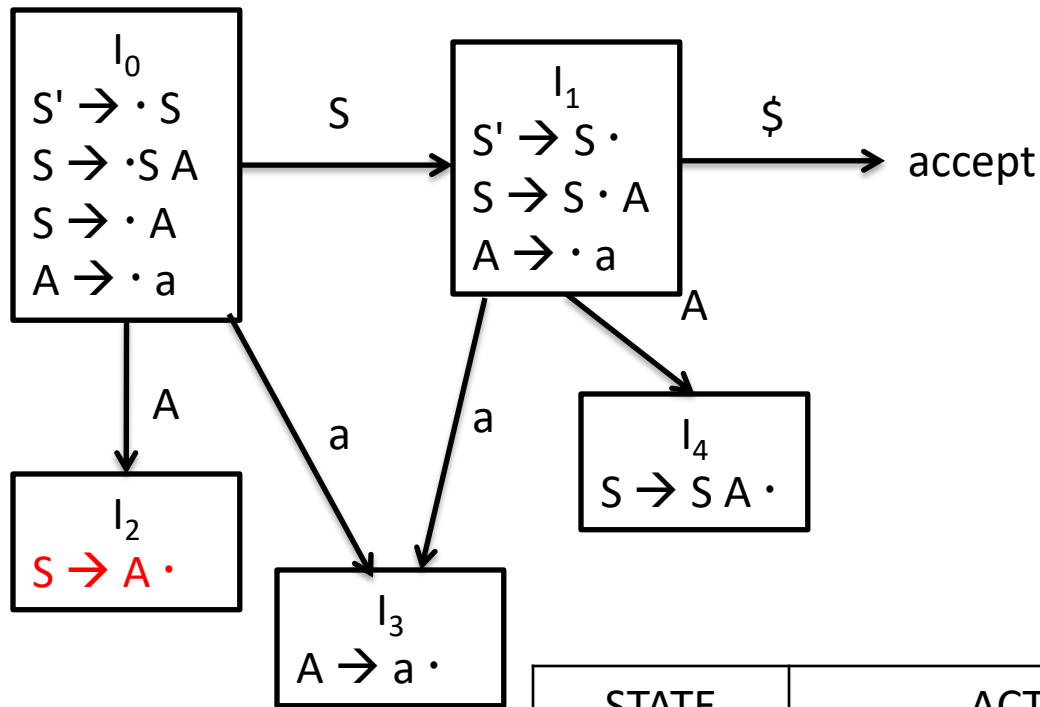
1.  $S \rightarrow S A$
2.  $S \rightarrow A$
3.  $A \rightarrow a$

$\text{FOLLOW}(S) = \{\$, \} \cup \text{FIRST}(A) = \{\$, a\}$   
 $\text{FOLLOW}(A) = \text{FOLLOW}(S) = \{\$, a\}$

Filling reduce entries

STATE	ACTION		GOTO	
	a	\$	S	A
0	s3		1	2
1	s3	acc		4
2				
3				
4				

# LR(0) Automaton



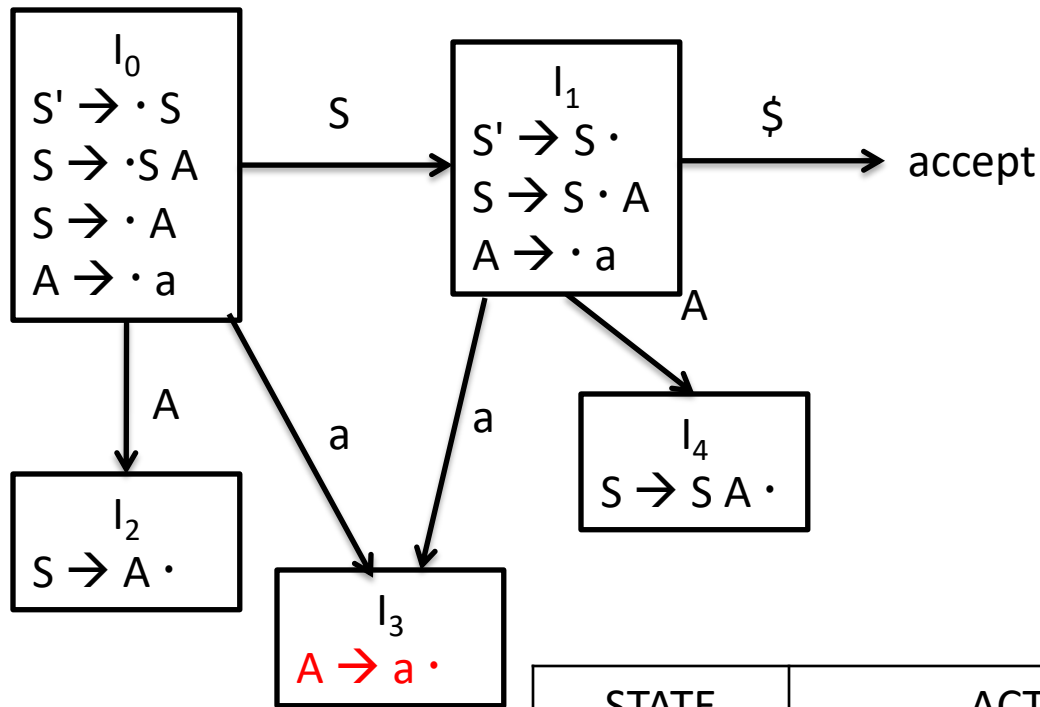
1.  $S \rightarrow S A$
2.  $S \rightarrow A$
3.  $A \rightarrow a$

$\text{FOLLOW}(S) = \{\$, \}$  U  $\text{FIRST}(A) = \{\$, a\}$   
 $\text{FOLLOW}(A) = \text{FOLLOW}(S) = \{\$, a\}$

Filling reduce entries

STATE	ACTION		GOTO	
	a	\$	S	A
0	s3		1	2
1	s3	acc		4
2	r2	r2		
3				
4				

# LR(0) Automaton



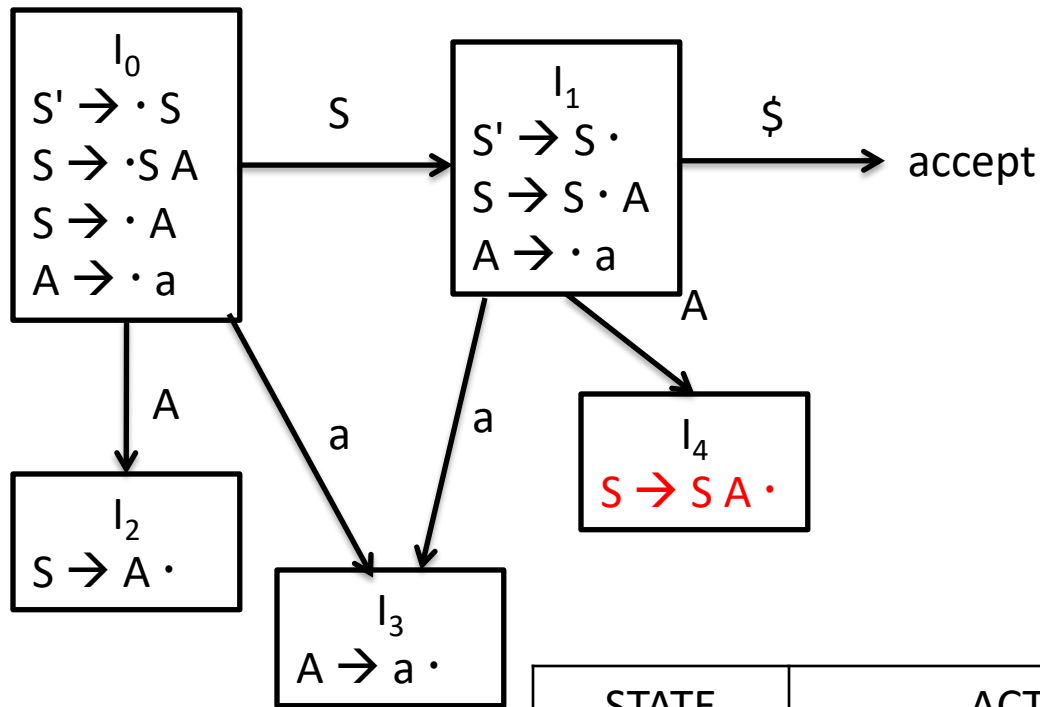
1.  $S \rightarrow S A$
2.  $S \rightarrow A$
3.  $A \rightarrow a$

$\text{FOLLOW}(S) = \{\$, \}$  U  $\text{FIRST}(A) = \{\$, a\}$   
 $\text{FOLLOW}(A) = \text{FOLLOW}(S) = \{\$, a\}$

Filling reduce entries

STATE	ACTION		GOTO	
	a	\$	S	A
0	s3		1	2
1	s3	acc		4
2	r2	r2		
3	r3	r3		
4				

# LR(0) Automaton



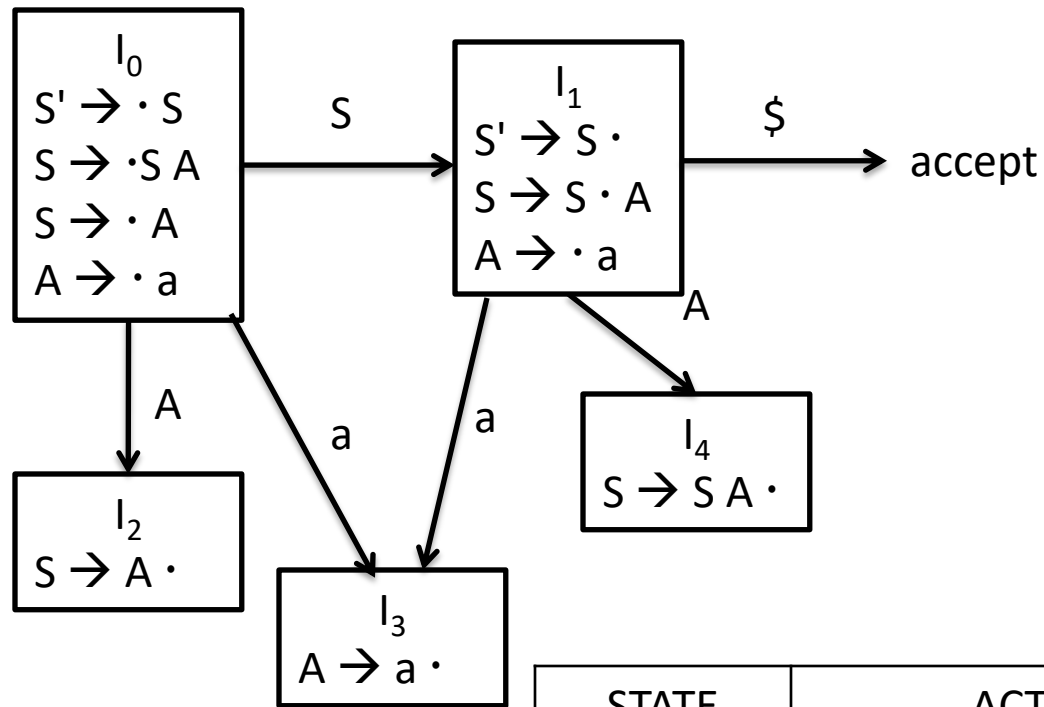
1.  $S \rightarrow S A$
2.  $S \rightarrow A$
3.  $A \rightarrow a$

$\text{FOLLOW}(S) = \{\$, \}$  U  $\text{FIRST}(A) = \{\$, a\}$   
 $\text{FOLLOW}(A) = \text{FOLLOW}(S) = \{\$, a\}$

Filling reduce entries

STATE	ACTION		GOTO	
	a	\$	S	A
0	s3		1	2
1	s3	acc		4
2	r2	r2		
3	r3	r3		
4	r1	r1		

# LR(0) Automaton



As there is no conflict so the grammar is SLR(1).

STATE	ACTION		GOTO	
	a	\$	S	A
0	s3		1	2
1	s3	acc		4
2	r2	r2		
3	r3	r3		
4	r1	r1		

Show that the given grammar is LL(1) but not SLR(1)

$S \rightarrow AaAb \mid BbBa$

$A \rightarrow \epsilon$

$B \rightarrow \epsilon$

- To check if grammar is LL(1),  
$$\text{FIRST}(AaAb) \cap \text{FIRST}(BbBa) = \{a\} \cap \{b\} = \emptyset$$
- Hence, **it is LL(1) grammar**. (this is first condition and no need to check second condition).
- For a grammar to be SLR(1), construct LR(0) automaton and parsing table. And there must be no conflict in any entry.

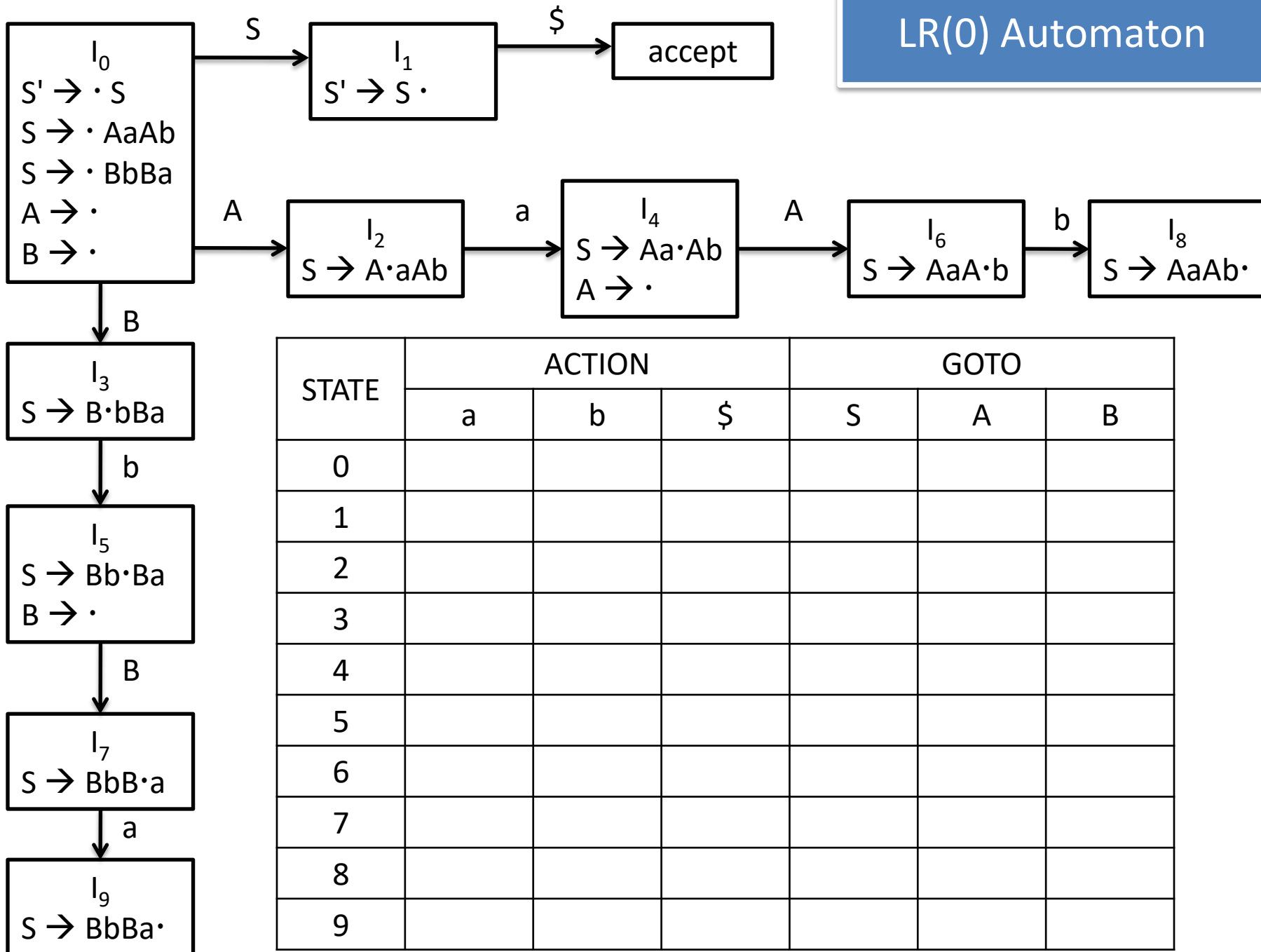
In  $A \rightarrow a \mid b$

If  $\text{FIRST}(a) \cap \text{FIRST}(b) \neq \emptyset$  : not LL(1)

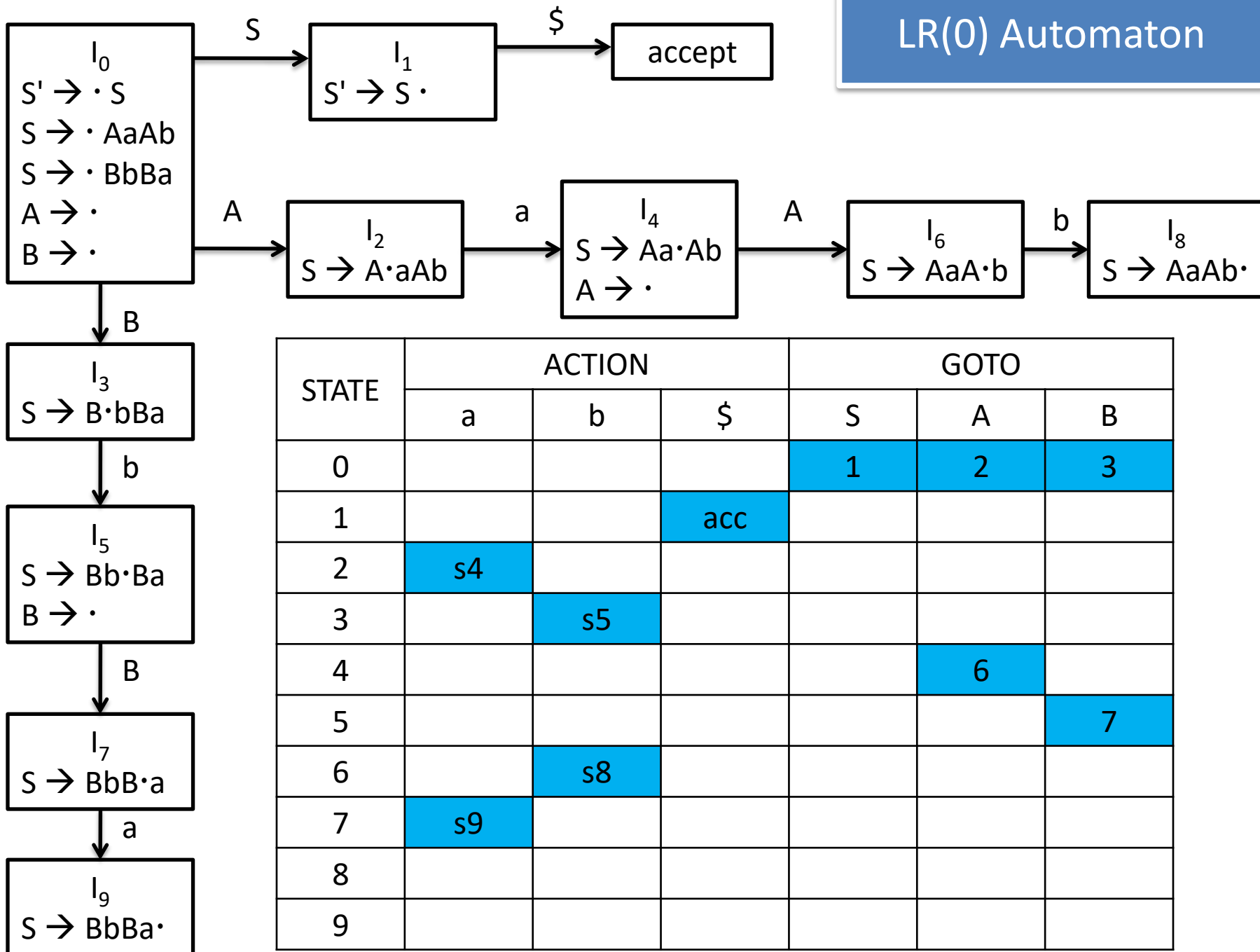
If  $\text{FIRST}(a) \cap \text{FIRST}(b) = \emptyset$ ,

then  $\text{FIRST}(a) = \epsilon$  **OR**  $\text{FOLLOW}(A) \cap \text{FIRST}(b) = \emptyset$  : then LL(1).

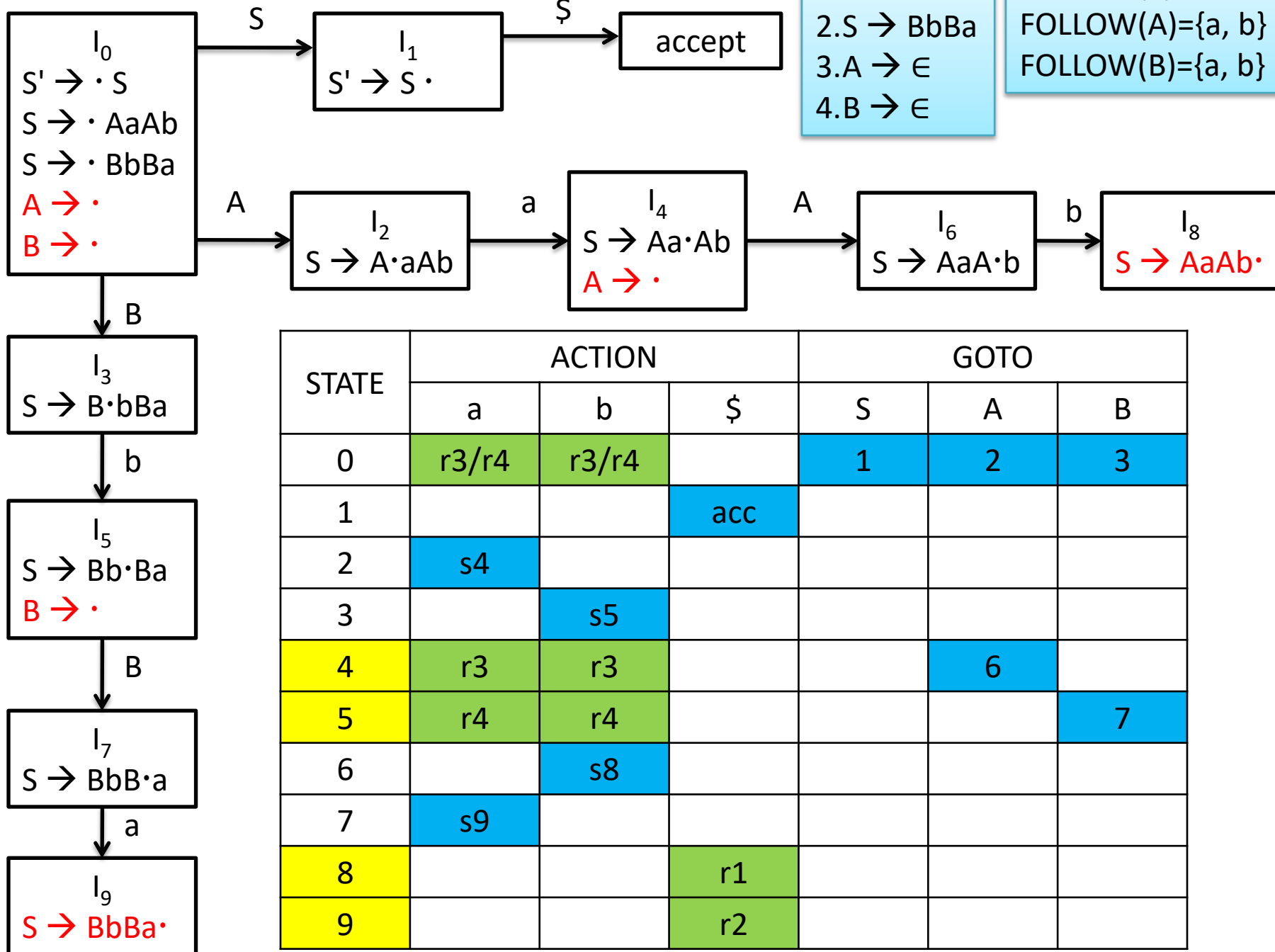
# LR(0) Automaton

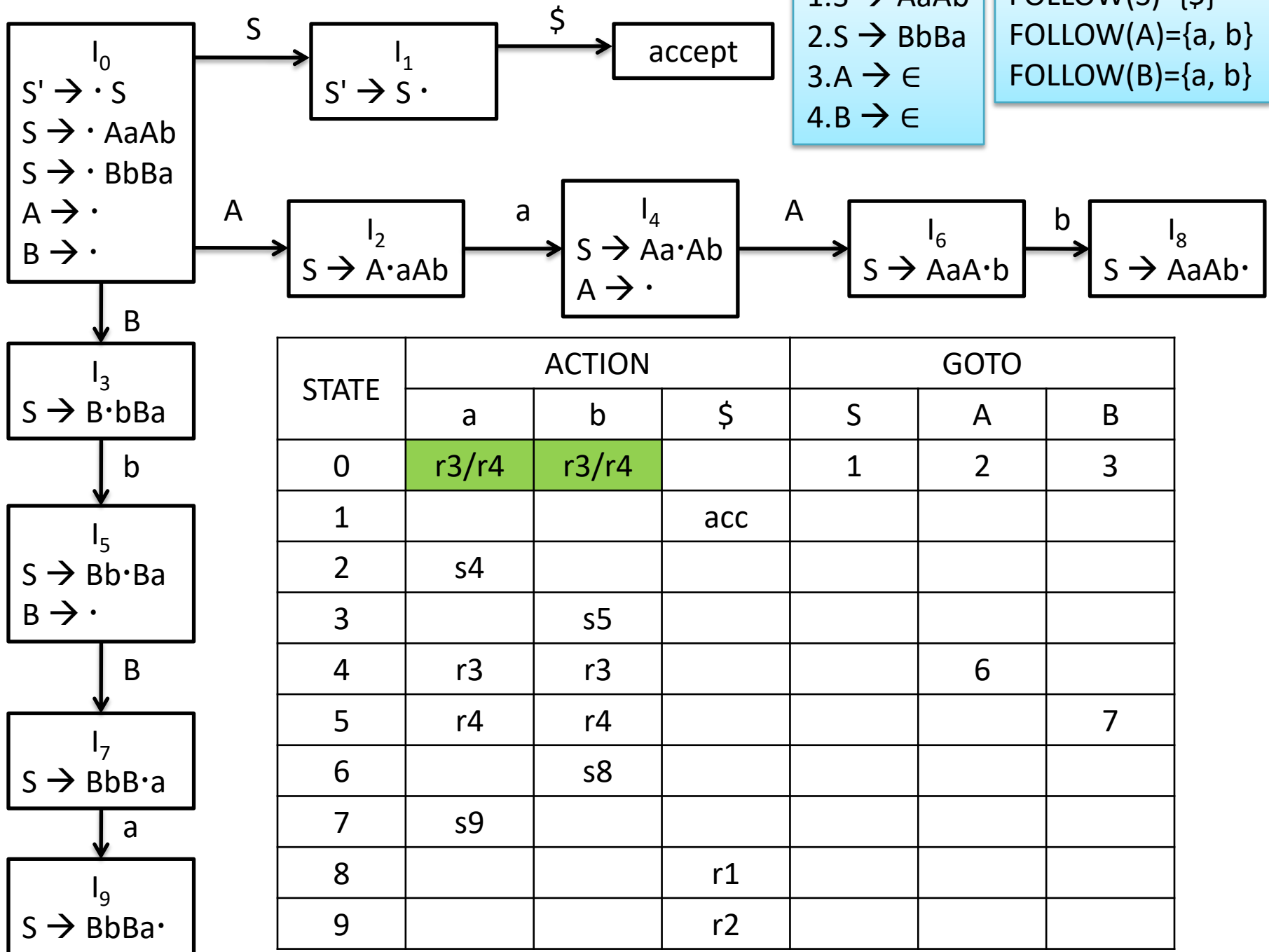


# LR(0) Automaton









Due to reduce/reduce conflict, it is not SLR(1)

