4.6.2 Items and the LR(0) Automaton

How does a shift-reduce parser know when to shift and when to reduce? For example, with stack contents $\$T$ and next input symbol $*$ in Fig. 4.28, how does the parser know that T on the top of the stack is not a handle, so the appropriate action is to shift and not to reduce T to E?

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 $A \to XYZ$ yields the four items

$$
\begin{aligned}
A &\to \cdot XYZ \\
A &\to X \cdot YZ \\
A &\to XY \cdot Z \\
A &\to XYZ \cdot
\end{aligned}
$$

The production $A \to \epsilon$ generates only one item, $A \to \cdot$.

Intuitively, an item indicates how much of a production we have seen at a given point in the parsing process. For example, the item $A \to \cdot XYZ$ indicates that we hope to see a string derivable from XYZ next on the input. Item

$A \to X \cdot YZ$ indicates that we have just seen on the input a string derivable from X and that we hope next to see a string derivable from YZ. Item $A \to XYZ \cdot$ indicates that we have seen the body XYZ and that it may be time to reduce XYZ to A.

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.*[3] In particular, each state of the LR(0) automaton represents a set of items in the canonical LR(0) collection. The automaton for the expression grammar (4.1), shown in Fig. 4.31, will serve as the running example for discussing the canonical LR(0) collection for a grammar.

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', the *augmented grammar* for G, is G with a new start symbol S' and production $S' \to S$. 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 $S' \to S$.

Closure of Item Sets

If I is a set of items for a grammar G, then CLOSURE(I) is the set of items constructed from I by the two rules:

1. Initially, add every item in I to CLOSURE(I).
2. If $A \to \alpha \cdot B\beta$ is in CLOSURE(I) and $B \to \gamma$ is a production, then add the item $B \to \cdot\gamma$ to CLOSURE(I), if it is not already there. Apply this rule until no more new items can be added to CLOSURE(I).

[3]Technically, the automaton misses being deterministic according to the definition of Section 3.6.4, because we do not have a dead state, corresponding to the empty set of items. As a result, there are some state-input pairs for which no next state exists.

Example 4.40: Consider the augmented expression grammar:

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

If I is the set of one item $\{[E' \rightarrow \cdot E]\}$, then CLOSURE($I$) contains the set of items I_0 in Fig. 4.31.

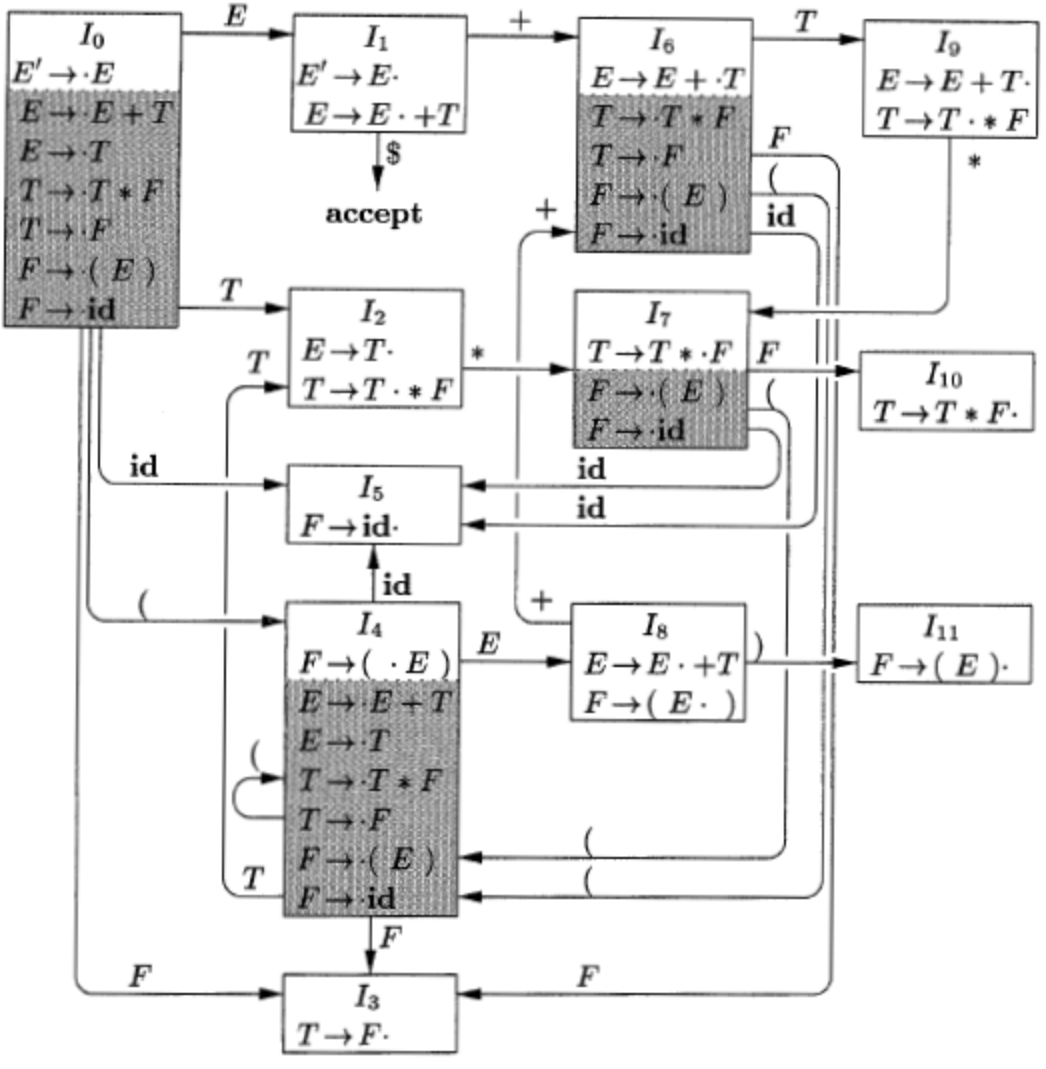


Figure 4.31: LR(0) automaton for the expression grammar (4.1)

To see how the closure is computed, $E' \to \cdot E$ is put in CLOSURE(I) by rule (1). Since there is an E immediately to the right of a dot, we add the E-productions with dots at the left ends: $E \to \cdot E + T$ and $E \to \cdot T$. Now there is a T immediately to the right of a dot in the latter item, so we add $T \to \cdot T * F$ and $T \to \cdot F$. Next, the F to the right of a dot forces us to add $F \to \cdot(E)$ and $F \to \cdot \mathbf{id}$, but no other items need to be added. □

The Function 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 $[A \to \alpha X \cdot \beta]$ such that $[A \to \alpha \cdot X\beta]$ 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.

Example 4.41: If I is the set of two items $\{[E' \to E\cdot],\ [E \to E\cdot + T]\}$, then GOTO$(I, +)$ contains the items

$$
\begin{aligned}
E &\to E + \cdot T \\
T &\to \cdot T * F \\
T &\to \cdot F \\
F &\to \cdot(E) \\
F &\to \cdot \mathbf{id}
\end{aligned}
$$

We computed GOTO$(I, +)$ by examining I for items with $+$ immediately to the right of the dot. $E' \to E\cdot$ is not such an item, but $E \to E\cdot + T$ is. We moved the dot over the $+$ to get $E \to E + \cdot T$ and then took the closure of this singleton set. □

4.6.3 The LR-Parsing Algorithm

A schematic of an LR parser is shown in Fig. 4.35. It consists of an input, an output, a stack, a driver program, and a parsing table that has two parts (ACTION and GOTO). 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.

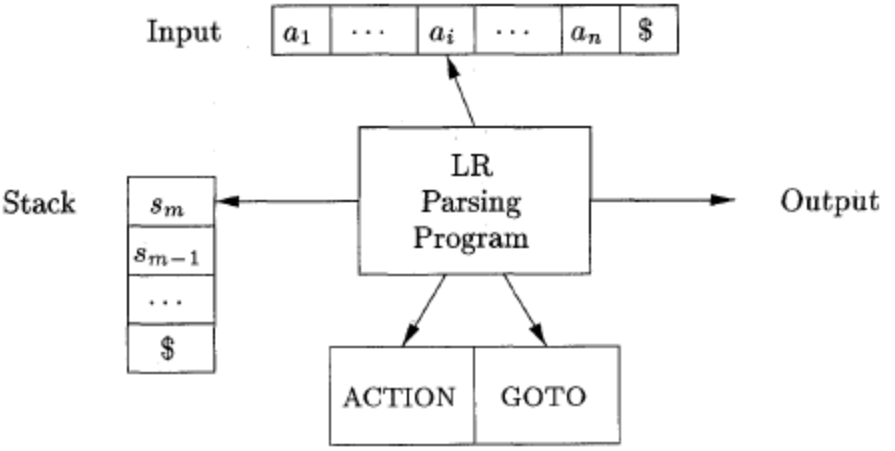


Figure 4.35: Model of an LR parser

Structure of the 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 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 β 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. We shall have more to say about how such error-recovery routines work in Sections 4.8.3 and 4.9.4.

2. We extend the GOTO function, defined on sets of items, to states: if GOTO$[I_i, A] = I_j$, then GOTO also maps a state i and a nonterminal A to state j.

Algorithm 4.44: LR-parsing algorithm.

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

OUTPUT: If w is in $L(G)$, the reduction steps of a bottom-up parse for w; 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. The parser then executes the program in Fig. 4.36. □

```
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;
}
```

Figure 4.36: LR-parsing program

Example 4.45: Figure 4.37 shows the ACTION and GOTO functions of an LR-parsing table for the expression grammar (4.1), repeated here with the productions numbered:

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

The codes for the actions are:

1. si means shift and stack state i,
2. rj means reduce by the production numbered j,
3. acc means accept,
4. blank means error.

Note that the value of GOTO[s, a] for terminal a is found in the ACTION field connected with the shift action on input a for state s. The GOTO field gives GOTO[s, A] for nonterminals A. Although we have not yet explained how the entries for Fig. 4.37 were selected, we shall deal with this issue shortly.

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

4.6.4 Constructing SLR-Parsing Tables

On input **id** $*$ **id** $+$ **id**, the sequence of stack and input contents is shown in Fig. 4.38. Also shown for clarity, are the sequences of grammar symbols corresponding to the states held on the stack. For example, at line (1) the LR parser is in state 0, the initial state with no grammar symbol, and with **id** the first input symbol. The action in row 0 and column **id** of the action field of Fig. 4.37 is s5, meaning shift by pushing state 5. That is what has happened at line (2): the state symbol 5 has been pushed onto the stack, and **id** has been removed from the input.

Then, $*$ becomes the current input symbol, and the action of state 5 on input $*$ is to reduce by $F \rightarrow$ **id**. One state symbol is popped off the stack. State 0 is then exposed. Since the goto of state 0 on F is 3, state 3 is pushed onto the stack. We now have the configuration in line (3). Each of the remaining moves is determined similarly. $\square$

	STACK	SYMBOLS	INPUT	ACTION
(1)	0		**id** $*$ **id** $+$ **id** \$	shift
(2)	0 5	**id**	$*$ **id** $+$ **id** \$	reduce by $F \rightarrow$ **id**
(3)	0 3	F	$*$ **id** $+$ **id** \$	reduce by $T \rightarrow F$
(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 $F \rightarrow$ **id**
(7)	0 2 7 10	$T*F$	$+$ **id** \$	reduce by $T \rightarrow T*F$
(8)	0 2	T	$+$ **id** \$	reduce by $E \rightarrow T$
(9)	0 1	E	$+$ **id** \$	shift
(10)	0 1 6	$E+$	**id** \$	shift
(11)	0 1 6 5	$E+$ **id**	\$	reduce by $F \rightarrow$ **id**
(12)	0 1 6 3	$E+F$	\$	reduce by $T \rightarrow F$
(13)	0 1 6 9	$E+T$	\$	reduce by $E \rightarrow E+T$
(14)	0 1	E	\$	accept

Figure 4.38: Moves of an LR parser on **id** $*$ **id** $+$ **id**

Algorithm 4.46: Constructing an SLR-parsing table.

INPUT: An augmented grammar G'.

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

METHOD:

1. Construct $C = \{I_0, I_1, \ldots, I_n\}$, 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 GOTO$(I_i, a) = I_j$, then set ACTION$[i, a]$ to "shift j." Here a must be a terminal.

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

   (c) If $[S' \to S \cdot]$ is in I_i, then set 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 $[S' \to \cdot S]$.

Example 4.48: Every SLR(1) grammar is unambiguous, but there are many unambiguous grammars that are not SLR(1). Consider the grammar with productions

$$\begin{array}{lcl} S & \rightarrow & L = R \mid R \\ L & \rightarrow & *R \mid \mathbf{id} \\ R & \rightarrow & L \end{array} \tag{4.49}$$

Think of L and R as standing for l-value and r-value, respectively, and $*$ as an operator indicating "contents of."[5] The canonical collection of sets of LR(0) items for grammar (4.49) is shown in Fig. 4.39.

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

I_1: $S' \to S\cdot$

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

I_3: $S \to R\cdot$

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

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

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

I_7: $L \to *R\cdot$

I_8: $R \to L\cdot$

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

Figure 4.39: Canonical LR(0) collection for grammar (4.49)

Consider the set of items I_2. The first item in this set makes ACTION$[2, =]$ be "shift 6." Since FOLLOW(R) contains $=$ (to see why, consider the derivation $S \Rightarrow L = R \Rightarrow *R = R$), the second item sets ACTION$[2, =]$ to "reduce $R \rightarrow L$." Since there is both a shift and a reduce entry in ACTION$[2, =]$, state 2 has a shift/reduce conflict on input symbol $=$.

Grammar (4.49) is not ambiguous. This shift/reduce conflict arises from the fact that the SLR parser construction method is not powerful enough to remember enough left context to decide what action the parser should take on input $=$, having seen a string reducible to L. The canonical and LALR methods,

Example 1: Consider the following augmented grammar:

1. $E' \rightarrow E\$$
2. $E \rightarrow id$
3. $E \rightarrow (E)$
4. $E \rightarrow (id)E$

i) Draw the LR(0) automation for this grammar.

ii) Construct SLR parsing table

iii) Is the grammar LR(0)? (why / why not)

	FIRST	FOLLOW
E'	first $(E') = \{id, (\}$	~~first~~ follow $(E') = \{\$\}$
E	first $(E) = \{id, (\}$	follow $(E) = \{\$,)\}$

(i) LR(0) automation:-

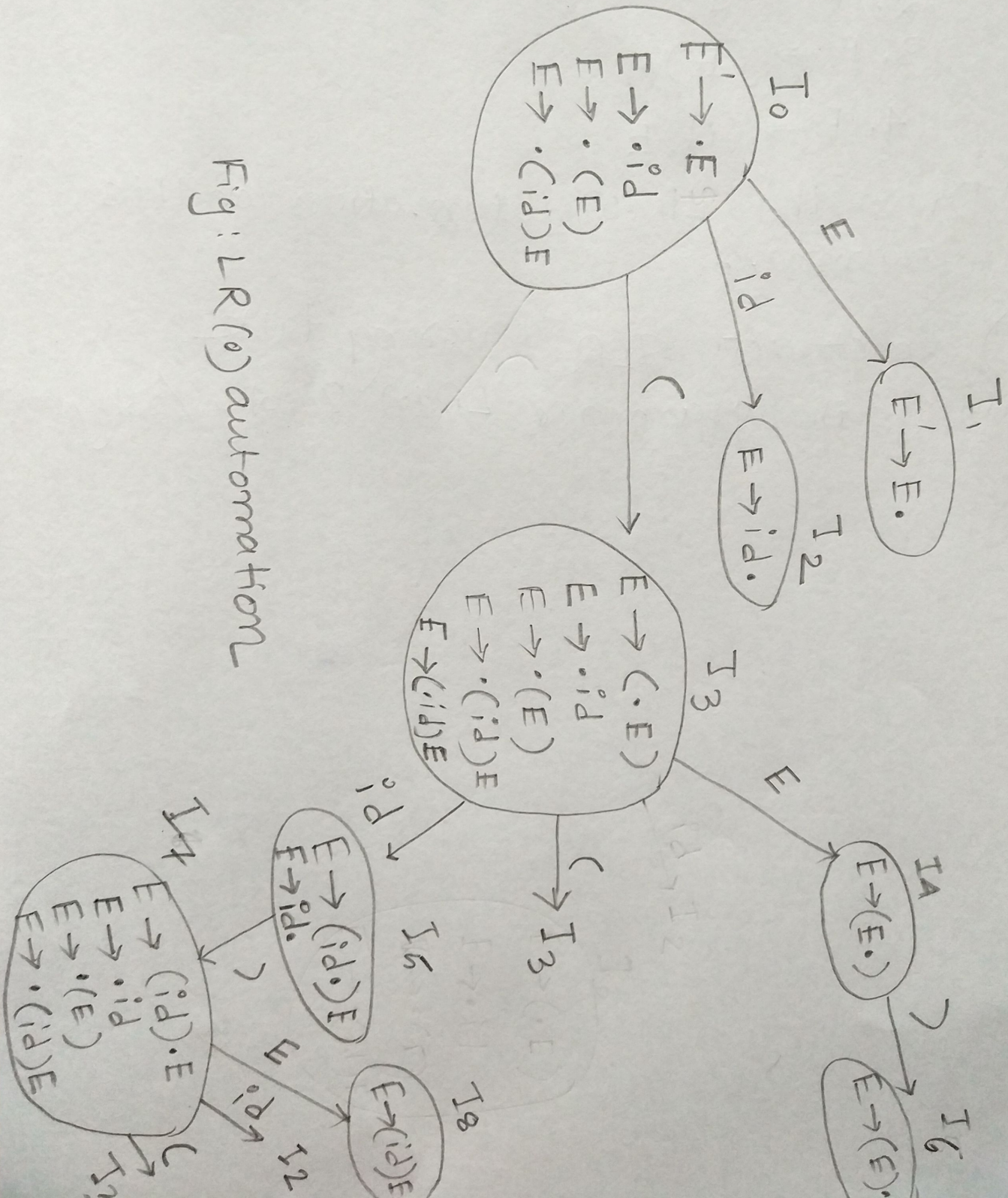


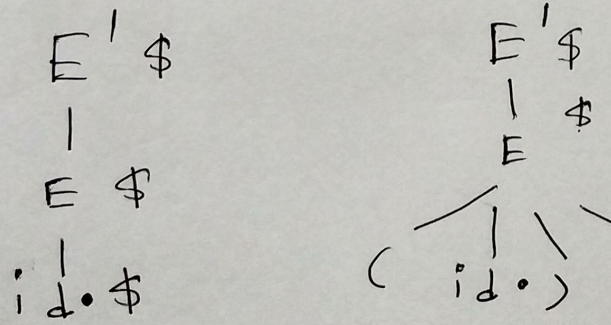
Fig: LR(0) automation

(ii)

	ACTION (i,a)				GOTO(i, A)
State	id	(	)	$	E
0	S2	S3			1
1				acc	
2			R2	R2	
3	S5	S3			4
4			S6		
5			S7 R2	R2	
6			R3	R3	
7	S2	S3			8
8			R4	R4	

(iii)

The grammar is not LR(0), because it has a shift-reduce conflict in state I_5



The appropriate action would be to shift & only reduce id to E when lookahead is $.