

بسم الله الرحمن الرحيم

دانشگاه صنعتی اصفهان – دانشکده مهندسی برق و کامپیوتر  
(نیم سال تحصیلی ۴۰۱۲)

کامپایلر

حسین فلسفین

## The Function GOTO

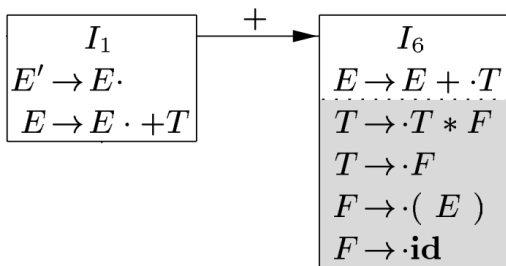
*The second useful function is  $\text{GOTO}(I, X)$  where  $I$  is a set of items and  $X$  is a grammar symbol (terminal or nonterminal).*

*$\text{GOTO}(I, X)$  is defined to be the **closure** of the set of all items  $[A \rightarrow \alpha X \bullet \beta]$  such that  $[A \rightarrow \alpha \bullet X \beta]$  is in  $I$ . (GOTO moves the dot past the symbol  $X$  in all items.)*

*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  $\text{GOTO}(I, X)$  specifies the transition from the state for  $I$  under input  $X$ .*

**Example:** If  $I$  is the set of two items  $\{[E' \rightarrow E\bullet], [E \rightarrow E\bullet + T]\}$ , then  $\text{GOTO}(I, +)$  contains the set of items  $I_6$  in Fig. 4.31.

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



**Goto**( $I, X$ ) =

set  $J$  to the empty set

**for** any item  $A \rightarrow \alpha.X\beta$  in  $I$

add  $A \rightarrow \alpha X.\beta$  to  $J$

**return** **Closure**( $J$ )

*We are now ready for the algorithm to construct  $C$ , the canonical collection of sets of LR(0) items for an augmented grammar  $G'$ .*

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

Figure 4.33: Computation of the canonical collection of sets of LR(0) items

## شبه‌کد کتاب اپل

Now here is the algorithm for LR(0) parser construction. First, augment the grammar with an auxiliary start production  $S' \rightarrow S\$$ . Let  $T$  be the set of states seen so far, and  $E$  the set of (shift or goto) edges found so far.

Initialize  $T$  to  $\{\mathbf{Closure}(\{S' \rightarrow .S\})\}$

Initialize  $E$  to empty.

**repeat**

**for** each state  $I$  in  $T$

**for** each item  $A \rightarrow \alpha.X\beta$  in  $I$

**let**  $J$  be  $\mathbf{Goto}(I, X)$

$T \leftarrow T \cup \{J\}$

$E \leftarrow E \cup \{I \xrightarrow{X} J\}$

**until**  $E$  and  $T$  did not change in this iteration

However, for the symbol  $\$$  we do not compute  $\mathbf{Goto}(I, \$)$ ; instead we will make an **accept** action.

# اتوماتون مربوط به گرامر (۴.۱) در کتاب آهو

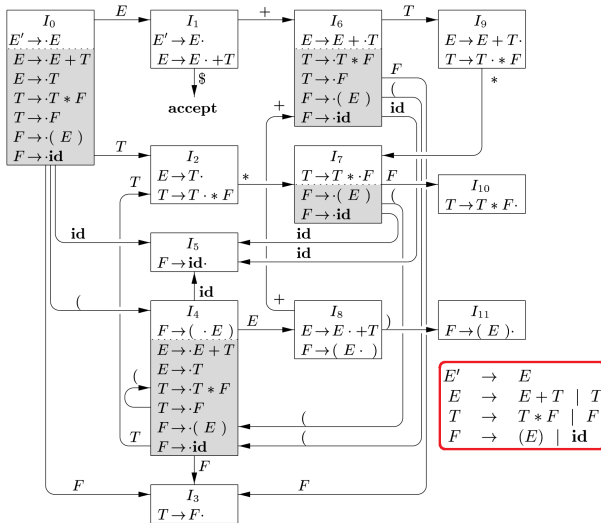
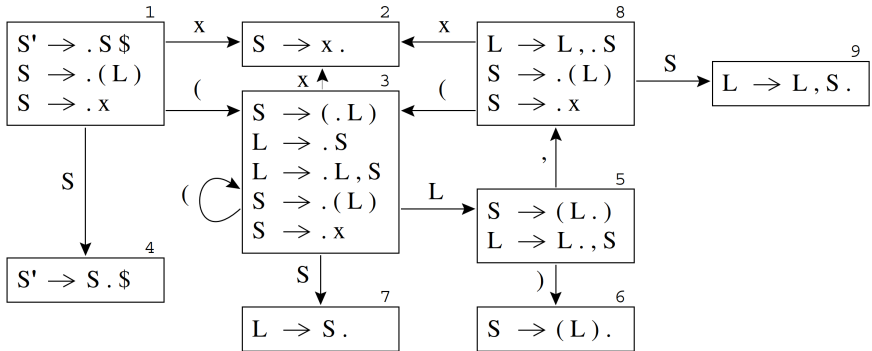


Figure 4.31: LR(0) automaton for the expression grammar (4.1)

# اتوماتون مربوط به گرامر (۳.۲۰) در کتاب اپل



0  $S' \rightarrow S \$$

3  $L \rightarrow S$

1  $S \rightarrow ( L )$

4  $L \rightarrow L , S$

2  $S \rightarrow x$

GRAMMAR 3.20.



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

Shift-reduce decisions can be made as follows. Suppose that the string  $\gamma$  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.

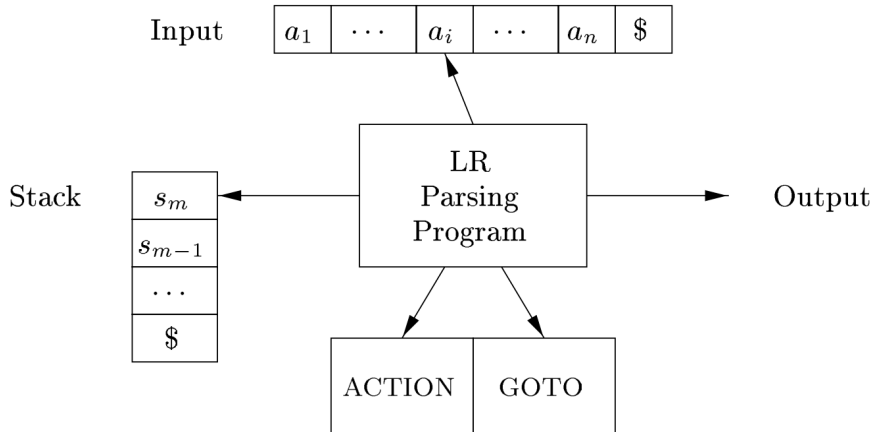
**Example 4.43:**

LINE	STACK	SYMBOLS	INPUT	ACTION
(1)	0	\$	<b>id * id</b> \$	shift to 5
(2)	0 5	\$ <b>id</b>	* <b>id</b> \$	reduce by $F \rightarrow \mathbf{id}$
(3)	0 3	\$ $F$	* <b>id</b> \$	reduce by $T \rightarrow F$
(4)	0 2	\$ $T$	* <b>id</b> \$	shift to 7
(5)	0 2 7	\$ $T$ *	<b>id</b> \$	shift to 5
(6)	0 2 7 5	\$ $T$ * <b>id</b>	\$	reduce by $F \rightarrow \mathbf{id}$
(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

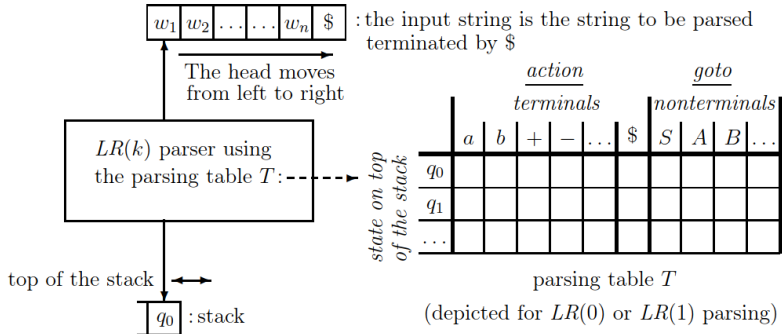
Figure 4.34: The parse of **id \* id**

## The LR-Parsing Algorithm

A schematic of an LR parser is shown in Fig. 4.35:



# یک ترسیم دیگر



A deterministic pushdown automaton for  $LR(k)$  parsing, with  $k \geq 0$ . The string to be parsed is  $w_1w_2 \dots w_n$ . Initially, the stack has one symbol only: the initial state  $q_0$  at the bottom of the stack. The input string is the string to be parsed with the extra right-most symbol  $\$$ . We have depicted the parsing table  $T$  for  $LR(0)$  or  $LR(1)$  parsers. For the  $LR(k)$  parsers, with  $k \geq 2$ , different parsing tables should be used.

## ذکر چند نکته حائز اهمیت

☞ 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**.

☞ By construction, **each state has a corresponding grammar symbol**. Recall that states correspond to sets of items, and that there is a transition from state  $i$  to state  $j$  if  $\text{GOTO}(I_i, X) = I_j$ . **All transitions to state  $j$  must be for the same grammar symbol  $X$** . Thus, each state, except the start state 0, has a unique grammar symbol associated with it. (The converse need not hold; that is, more than one state may have the same grammar symbol.)

## 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  $\beta$  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-Parser Configurations

To describe the behavior of an LR parser, it helps to have a notation representing the complete state of the parser: its stack and the remaining input.

A configuration of an LR parser is a pair:

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

where the first component is the stack contents (top on the right), and the second component is the remaining input.



## Behavior of the LR Parser

*The next move of the parser from the configuration*

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

*is determined by reading  $a_i$ , the current input symbol, and  $s_m$ , the state on top of the stack, and then consulting the entry  $\text{ACTION}[s_m, a_i]$  in the parsing action table. The configurations resulting after each of the four types of move are as follows*

1. If  $\text{ACTION}[s_m, a_i] = \text{shift } s$ , the parser executes a shift move; it shifts the next state  $s$  onto the stack, entering the configuration

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

The symbol  $a_i$  need not be held on the stack, since it can be recovered from  $s$ , if needed (which in practice it never is). The current input symbol is now  $a_{i+1}$ .

2. If  $\text{ACTION}[s_m, a_i] = \text{reduce } A \rightarrow \beta$ , then the parser executes a reduce move, entering the configuration

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

where  $r$  is the length of  $\beta$ , and  $s = \text{GOTO}[s_{m-r}, A]$ . Here the parser first popped  $r$  state symbols off the stack, exposing state  $s_{m-r}$ . The parser then pushed  $s$ , the entry for  $\text{GOTO}[s_{m-r}, A]$ , onto the stack. The current input symbol is **not changed** in a reduce move. For the LR parsers we shall construct,  $X_{m-r+1} \cdots X_m$ , the sequence of grammar symbols corresponding to the states popped off the stack, will always match  $\beta$ , **the right side** of the reducing production.

The output of an LR parser is generated after a reduce move by executing the semantic action associated with the reducing production. For the time being, we shall assume the output consists of just printing the reducing production.

3. If  $\text{ACTION}[s_m, a_i] = \text{accept}$ , parsing is completed.
4. If  $\text{ACTION}[s_m, a_i] = \text{error}$ , the parser has discovered an error and calls an error recovery routine.

**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.

□

## LR-parsing program

```
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;  
}
```