Bottom-up parsing

LR grammars

4.5 Bottom-Up Parsing

A bottom-up parse corresponds to the construction of a parse tree for an input string beginning at the leaves (the bottom) and working up towards the root (the top).

$$
\begin{array}{lcl}
E & \rightarrow & E + T \mid T \\
T & \rightarrow & T * F \mid F \\
F & \rightarrow & (E) \mid \mathbf{id}
\end{array}
\qquad (4.1)
$$

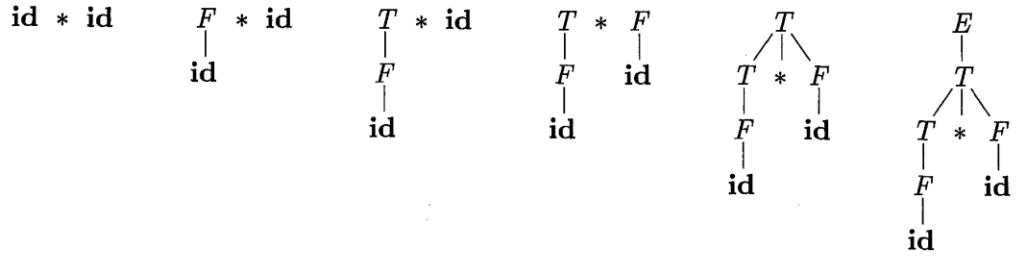


Figure 4.25: A bottom-up parse for **id** * **id**

id * **id**, F * **id**, T * **id**, $T * F$, T, E

Usually, bottom-up parsing is a sequence of rightmost derivations, but in the reverse order

$$E \underset{rm}{\Rightarrow} T \underset{rm}{\Rightarrow} T * F \underset{rm}{\Rightarrow} T * \mathbf{id} \underset{rm}{\Rightarrow} F * \mathbf{id} \underset{rm}{\Rightarrow} \mathbf{id} * \mathbf{id}$$

- Bottom-up parsers don't need left-factored grammars

  - Revert to the "natural" grammar for our example:

    $E \rightarrow T + E \mid T$

    $T \rightarrow \text{int} * T \mid \text{int} \mid (E)$

  - Consider the string: int * int + int

$E \rightarrow T + E \mid T$

$T \rightarrow \text{int} * T \mid \text{int} \mid (E)$

Bottom-up parsing *reduces* a string to the start symbol

int * int + int	$T \rightarrow \text{int}$
int * T + int	$T \rightarrow \text{int} * T$
T + int	$T \rightarrow \text{int}$
T + T	$E \rightarrow T$
T + E	$E \rightarrow T + E$
E	

Important difference between top-down and bottom-up

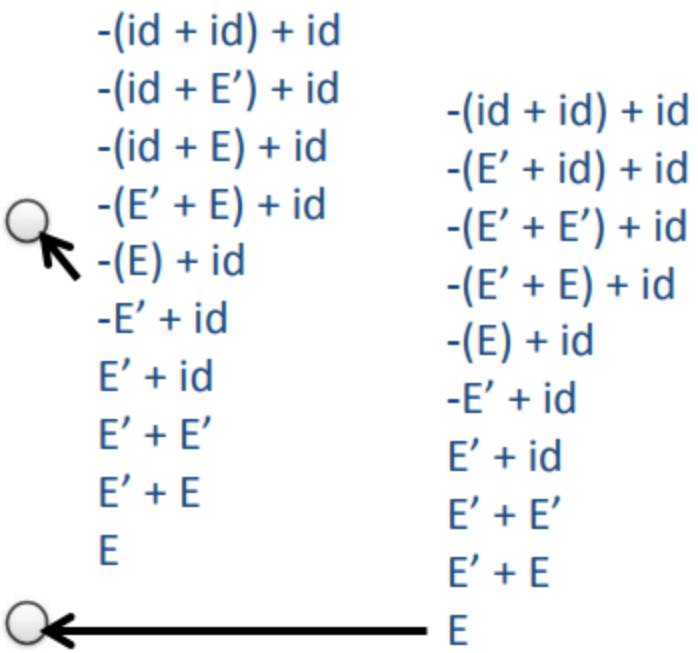
- Fact #1
- Bottom-up traces the right-most derivation, but in the reverse order.
  - Recall, Top-down traces the left-most derivation, in the right order.

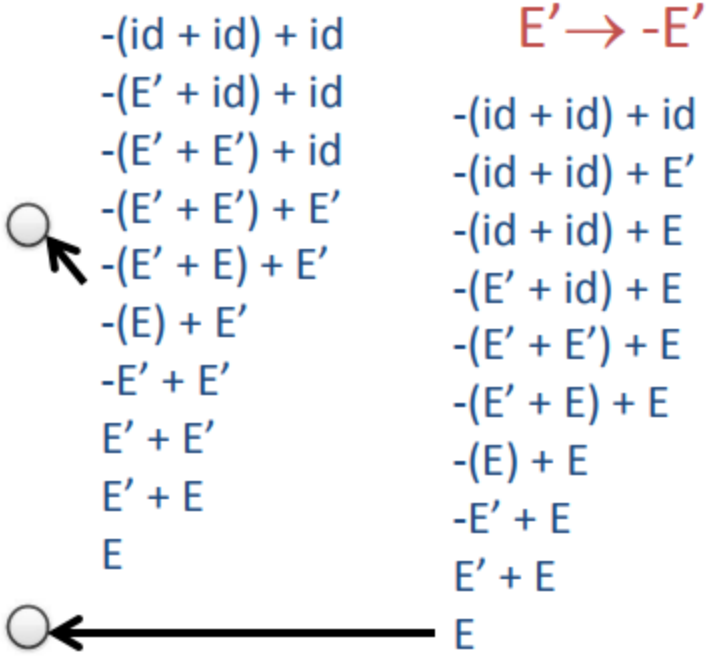
Bottom-Up Parsing

For the given grammar, what is the correct series of reductions for the string: -(id + id) + id

$E \rightarrow E' \mid E' + E$

$E' \rightarrow -E' \mid id \mid (E)$





4.5.1 Reductions

We can think of bottom-up parsing as the process of "reducing" a string w to the start symbol of the grammar.

At each *reduction* step, a specific substring matching the body of a production is replaced by the nonterminal at the head of that production.

The key decisions during bottom-up parsing are about when to reduce and about what production to apply, as the parse proceeds.

Handle

- Informally, a “handle” is a substring that matches the body of a production, and whose reduction represents one step along the reverse of a rightmost derivation.

For example, for $\mathbf{id} * \mathbf{id}$ subscripts subscripts are added for clarity, the handles during parsing of $\mathbf{id}_1 * \mathbf{id}_2$ according to grammar (4.1).

RIGHT SENTENTIAL FORM	HANDLE	REDUCING PRODUCTION
$\mathbf{id}_1 * \mathbf{id}_2$	$\mathbf{id}_1$	$F \to \mathbf{id}$
$F * \mathbf{id}_2$	F	$T \to F$
$T * \mathbf{id}_2$	$\mathbf{id}_2$	$F \to \mathbf{id}$
$T * F$	$T * F$	$E \to T * F$

Figure 4.26: Handles during a parse of $\mathbf{id}_1 * \mathbf{id}_2$

Important Fact #2 about bottom-up parsing:

In shift-reduce parsing, handles appear only at the top of the stack, never inside

- In shift-reduce parsing, handles always appear at the top of the stack
- Handles are never to the left of the rightmost non-terminal
  - Therefore, shift-reduce moves are sufficient;
- Bottom-up parsing algorithms are based on recognizing handles

Let $\gamma = \alpha\beta w$ be a right-sentential form.

We say β is a handle (along with the position in the γ), if

(1) $S \underset{rm}{\overset{*}{\Rightarrow}} \alpha A w \underset{rm}{\Rightarrow} \alpha\beta w$

(2) $A \to \beta$ is a production

(3) w consists of only terminals.

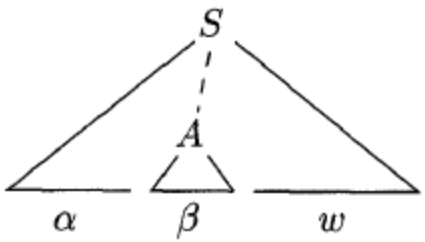


Figure 4.27: A handle $A \to \beta$ in the parse tree for $\alpha\beta w$

•If a grammar is unambiguous , then every right-sentential form of the grammar will have exactly one handle.

SEQUENCE OF RIGHT SENTENTIAL FORMS
IN DERIVING THE STRING *w*

$$S = \gamma_0 \underset{rm}{\Rightarrow} \gamma_1 \underset{rm}{\Rightarrow} \gamma_2 \underset{rm}{\Rightarrow} \cdots \underset{rm}{\Rightarrow} \gamma_{n-1} \underset{rm}{\Rightarrow} \gamma_n = w$$

Given Input String

we locate the handle β_n

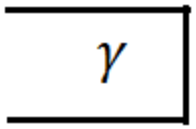
$$A_n \rightarrow \beta_n$$

replace β_n by A_n to get γ_{n-1}

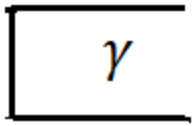
Same process is continued up to S

How stack is represented

- In top-down parsing stack is represented like:



  - Top of the stack is towards the left.
- In bottom-up parsing stack is represented like:



  - Top of the stack is towards the right.

Shift-reduce parsing

STACK	INPUT
$	w $

During a left-to-right scan of the input string, the parser shifts zero or more input symbols onto the stack, until it is ready to reduce a string β of grammar symbols on top of the stack.

It then reduces β to the head of the appropriate production.

The parser repeats this cycle until it has detected an error or until the stack contains the start symbol and the input is empty:

STACK	INPUT	ACTION
$	$\mathbf{id}_1 * \mathbf{id}_2$ $	shift
$ $\mathbf{id}_1$	$* \mathbf{id}_2$ $	reduce by $F \to \mathbf{id}$
$ F	$* \mathbf{id}_2$ $	reduce by $T \to F$
$ T	$* \mathbf{id}_2$ $	shift
$ $T *$	$\mathbf{id}_2$ $	shift
$ $T * \mathbf{id}_2$	$	reduce by $F \to \mathbf{id}$
$ $T * F$	$	reduce by $T \to T * F$
$ T	$	reduce by $E \to T$
$ E	$	accept

Figure 4.28: Configurations of a shift-reduce parser on input $\mathbf{id}_1 * \mathbf{id}_2$

While the primary operations are shift and reduce, there are actually four possible actions a shift-reduce parser can make: (1) shift, (2) reduce, (3) accept, and (4) error.

1. *Shift.* Shift the next input symbol onto the top of the stack.
2. *Reduce.* The right end of the string to be reduced must be at the top of the stack. Locate the left end of the string within the stack and decide with what nonterminal to replace the string.
3. *Accept.* Announce successful completion of parsing.
4. *Error.* Discover a syntax error and call an error recovery routine.

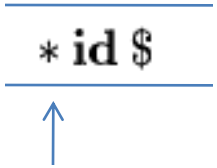
- Shift pushes a terminal on the stack

- Reduce
  - pops symbols off of the stack (production rhs)
  - pushes a non-terminal on the stack (production lhs)

- In a given state, more than one action (shift or reduce) may lead to a valid parse
- If it is legal to shift or reduce, there is a *shift-reduce* conflict
- If it is legal to reduce by two different productions, there is a *reduce-reduce* conflict

Shift Reduce Conflict

$$
\begin{array}{lcl}
E & \rightarrow & E + T \mid T \\
T & \rightarrow & T * F \mid F \\
F & \rightarrow & (E) \mid \mathbf{id}
\end{array}
$$



- We do not know whether to shift * on to stack
- Or, to reduce T to E

An ambiguous grammar will have this conflict

$$
\begin{array}{lcl}
stmt & \rightarrow & \textbf{if } expr \textbf{ then } stmt \\
 & | & \textbf{if } expr \textbf{ then } stmt \textbf{ else } stmt \\
 & | & \textbf{other}
\end{array}
$$

STACK	INPUT
$\cdots$ **if** *expr* **then** *stmt*	**else** $\cdots$ \$

- if *expr* then *stmt* can be reduced to stmt
- Or, *else* can be shifted on to the stack

- Normally this conflict is resolved in favor of shifting *else*

Reduce-reduce conflict

STACK	INPUT BUFFER
$...Ba	*w$*

$$Grammar$$

$$\ldots$$

$$A \rightarrow a$$

$$B \rightarrow Ba$$

Which one is the handle?
Both are at top of the stack !

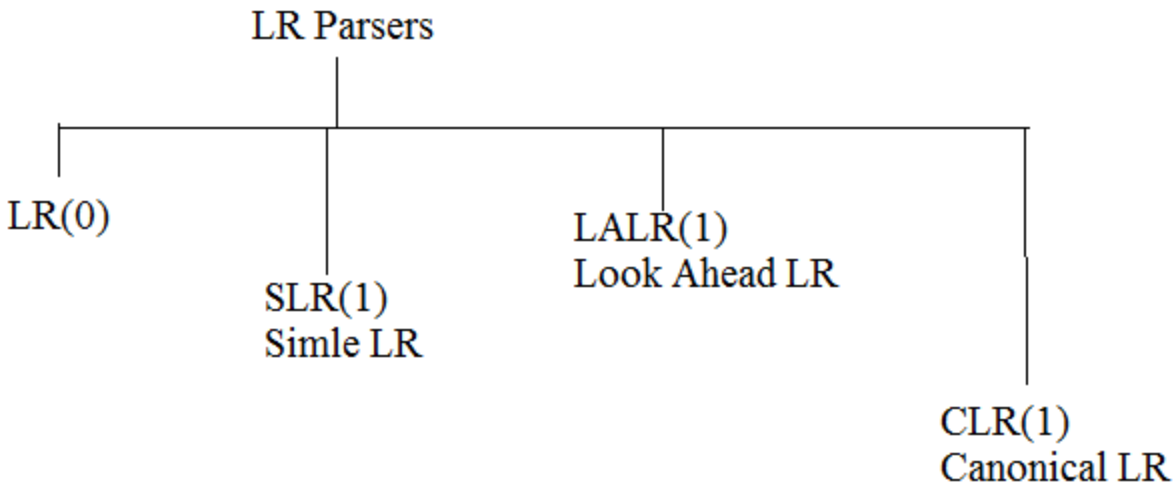
***By proper analysis of the CFG, one can resolve conflicts**
***thus, resulting in various parsing methods.**
***Theory says, there will be utmost one handle at the top of the stack. Since, there is a unique right-most derivation (CFG being unambiguous). This, to be applied, demands a longer look-ahead.**

LR Parsers

- L stands for left to right reading of input
- R stands for the right-most derivation (in reverse order).

- If shift reduce parsing is possible to build the parse tree, we call such a grammar LR grammar.
- Automatic parsing tools like Yacc are using this type of parsing.

A categorization of LR parsers



- LR Parsing Table
  - LR(0), SLR(1) depends on LR(0) items
  - LALR(1) and CLR(1) depends on LR(1) items

LR(0) item

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

$$A \to \cdot XYZ$$
$$A \to X \cdot YZ$$
$$A \to XY \cdot Z$$
$$A \to XYZ \cdot$$

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

What does item means…

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.

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.

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
will serve as the running example for discussing the canonical LR(0) collection for a grammar.

$$
\begin{array}{lcl}
E & \rightarrow & E+T \mid T \\
T & \rightarrow & T*F \mid F \\
F & \rightarrow & (E) \mid \mathbf{id}
\end{array}
$$

Augmented 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 $S' \rightarrow S$.

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 $S' \rightarrow S$.

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

Closure of Item Sets

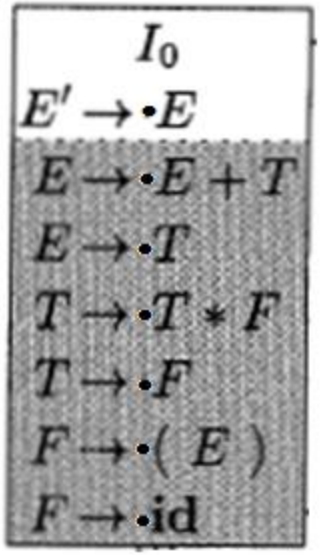
Closure of Item Sets

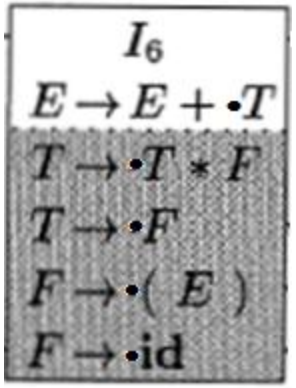
If I is a set of items for a grammar G, then CLOSURE(I) is the set of items constructed from I by the two rules:

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).

Intuitively, $A \to \alpha \cdot B\beta$ in CLOSURE(I) indicates that, at some point in the parsing process, we think we might next see a substring derivable from $B\beta$ as input. The substring derivable from $B\beta$ will have a prefix derivable from B by applying one of the B-productions. We therefore add items for all the B-productions; that is, if $B \to \gamma$ is a production, we also include $B \to \cdot\gamma$ in CLOSURE(I).

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





1. *Kernel items*: the initial item, $S' \rightarrow \cdot S$, and all items whose dots are not at the left end.

2. *Nonkernel items*: all items with their dots at the left end, except for $S' \rightarrow \cdot S$.

In Fig. 4.31, nonkernel items are in the shaded part of the box for a state.

Moreover, each set of items of interest is formed by taking the closure of a set of kernel items; the items added in the closure can never be kernel items, of course.

Thus, we can represent the sets of items we are really interested in with very little storage if we throw away all nonkernel items, knowing that they could be regenerated by the closure process. In Fig. 4.31, nonkernel items are in the shaded part of the box for a state.

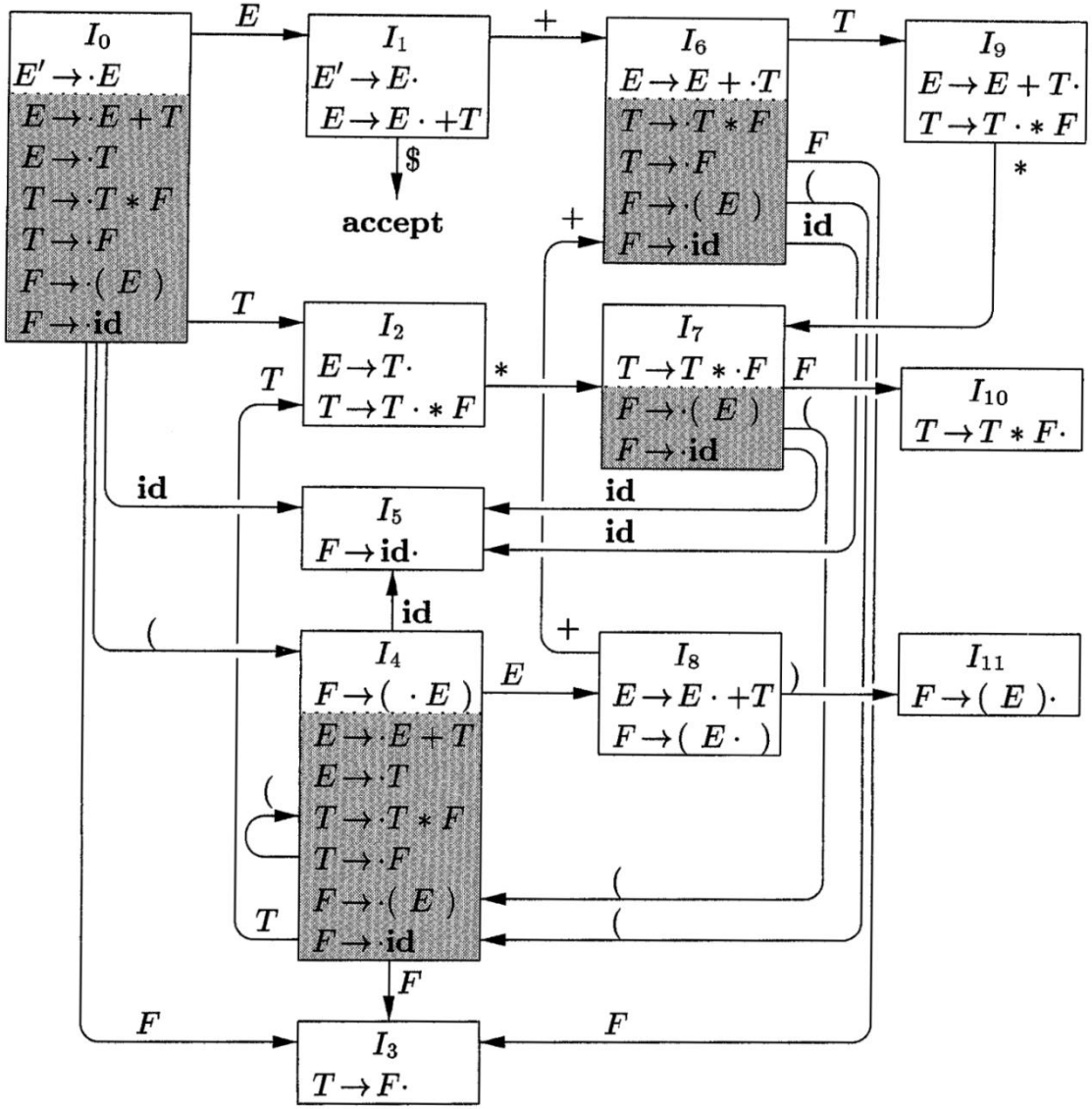


Figure 4.31: LR(0) automaton for the expression grammar (4.1)

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.

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

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

$$
\begin{array}{l}
E \rightarrow E + \cdot T \\
T \rightarrow \cdot T * F \\
T \rightarrow \cdot F \\
F \rightarrow \cdot (E) \\
F \rightarrow \cdot \mathbf{id}
\end{array}
$$

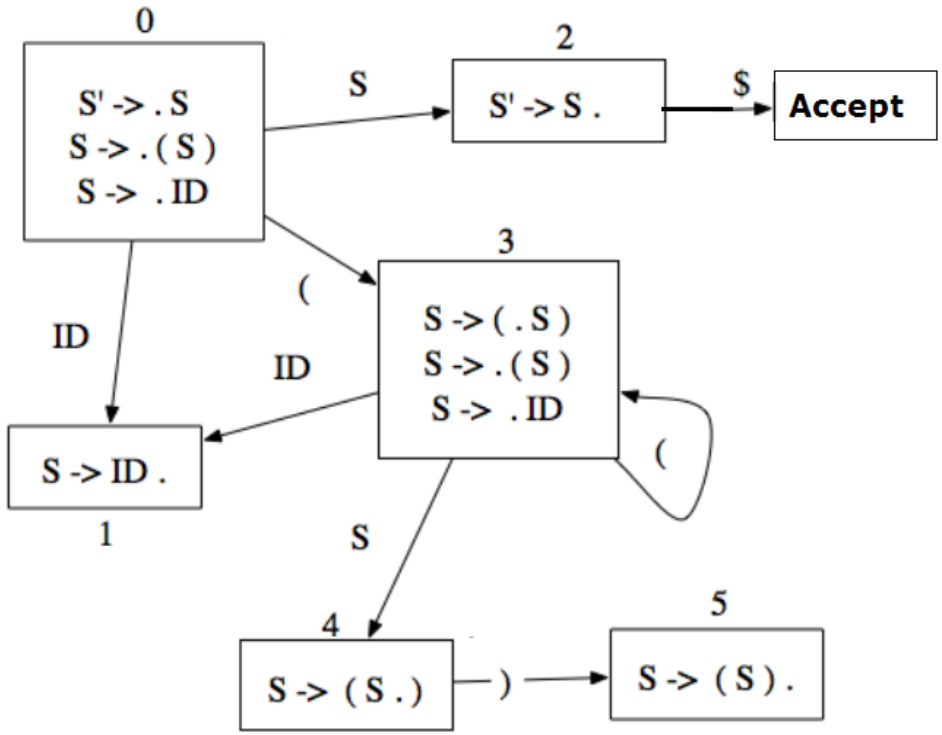
We computed GOTO$(I, +)$ by examining I for items with $+$ immediately to the right of the dot. $E' \rightarrow E\cdot$ is not such an item, but $E \rightarrow E\cdot + T$ is. We moved the dot over the $+$ to get $E \rightarrow E + \cdot T$ and then took the closure of this singleton set. $\square$

LR(0) Parsing

- Upon reaching an item of form $X \rightarrow \alpha \cdot \beta$ where β is not empty string, you have to shift the next input onto stack. **Must shift.**
- Upon reaching an item of form $X \rightarrow \alpha \cdot$ you have to reduce. **Must reduce.**
- If no conflicts then the CFG is LR(0).
- Otherwise, rewrite the grammar or use stronger parsing techniques.

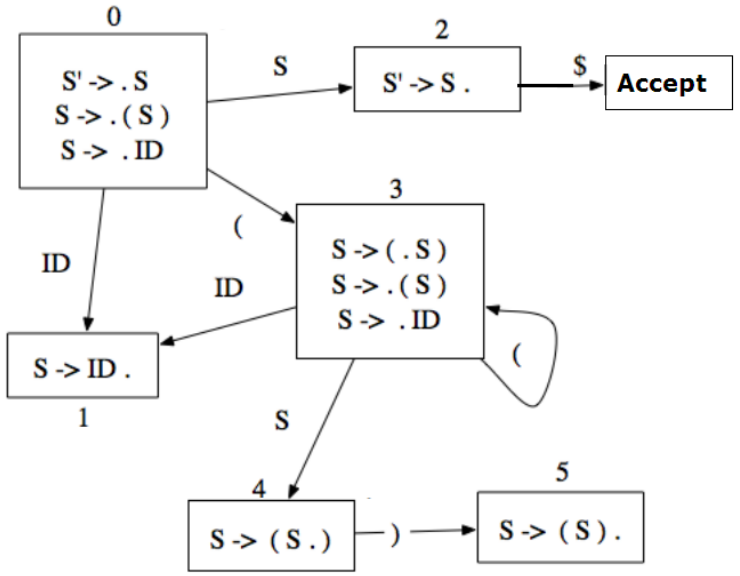
Example

(0) $S \rightarrow (S)$
(1) $S' \rightarrow S$
(2) $S \rightarrow id$



(0) $S \rightarrow (S)$
(1) $S' \rightarrow S$
(2) $S \rightarrow id$

	Action				Goto
State	(	)	$	ID	S
0	s3			s1	2
1	r2	r2	r2	r2	
2			accept		
3	s3			s1	4
4		s5			
5	r0	r0	r0	r0	



Stack of States	Stack of Symbols	Input	Action
0	$	(id)$	Shift (
0 3	$(	id)$	Shift id
0 3 1	$(id	)$	Reduce S->id
0 3 4	$(S	)$	Shift)
0 3 4 5	$(S)	$	Reduce S->(S)
0 2	$S	$	Accept

The grammar is LR(0)

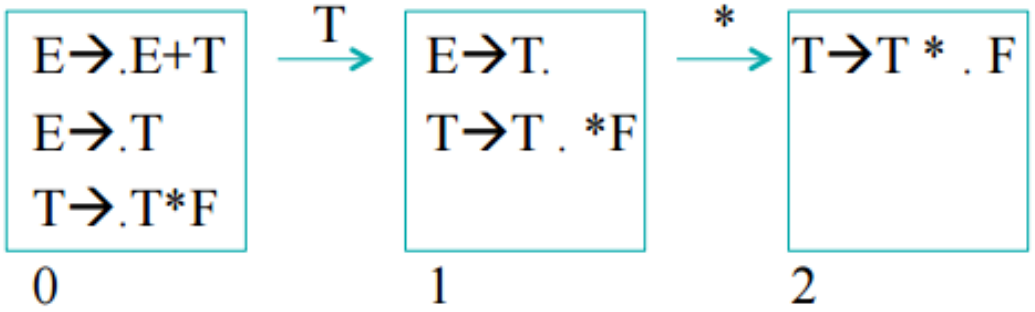
For input : **(id)**

Problem with LR(0): shift reduce conflict

If there is an item A→α . item in I , we reduce for all terminals

This can cause CONFLICTS:

E→ E+T
E → T
T → T*F



In state 1:

we reduce (E→T.) AND we shift (T→T . *F)

What should we do?

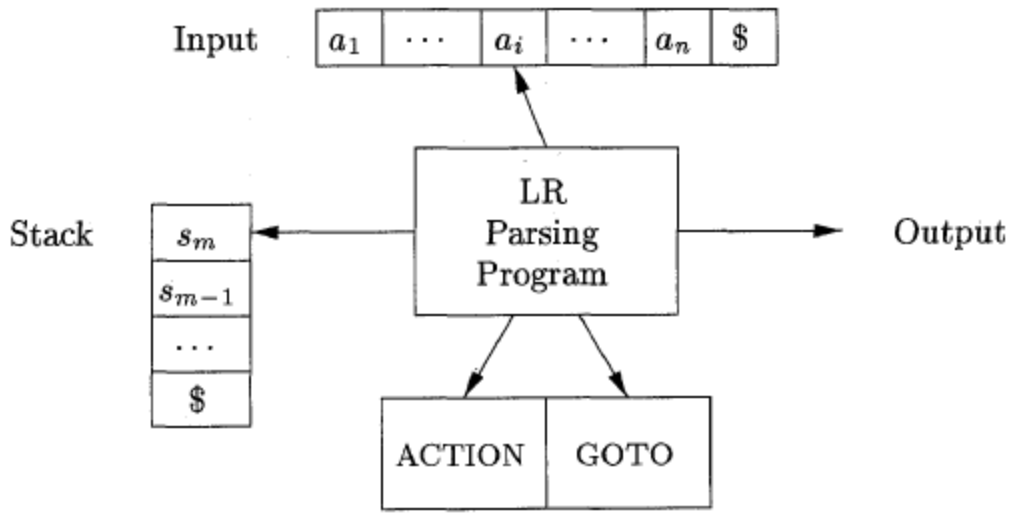


Figure 4.35: Model of an LR parser

In LR parsing ACTION, GOTO are given in the parsing table.
Any LR parsing looks like the above figure. Only the parsing table differs from parser to parser.

LR(0) Parse Table

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

STATE	ACTION						GOTO		
	id	+	*	(	)	$	E	T	F
0	s5			s4			1	2	3
1		s6				acc			
2	r2	r2	r2/s7	r2	r2	r2			
3	r4	r4	r4	r4	r4	r4			
4	s5			s4			8	2	3
5	r6	r6	r6	r6	r6	r6			
6	s5			s4				9	3
7	s5			s4					10
8		s6			s11				
9	r1	r1	r1/s7	r1	r1	r1			
10	r3	r3	r3	r3	r3	r3			
11	r5	r5	r5	r5	r5	r5			

The grammar is not LR(0)

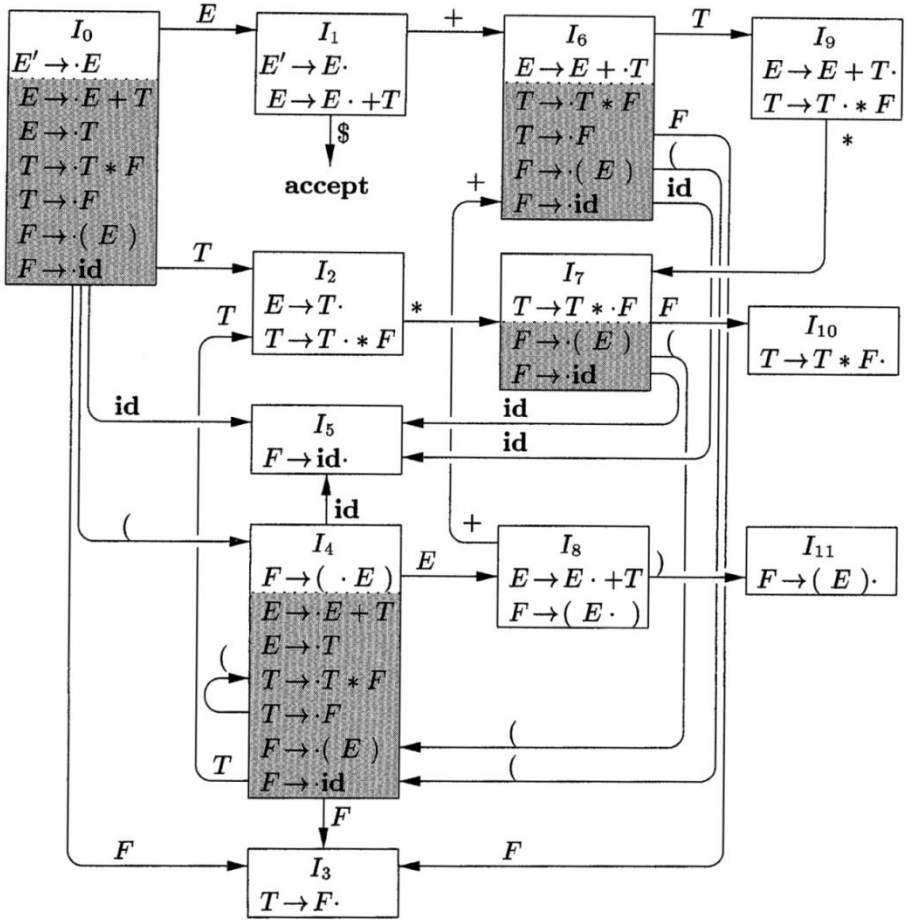


Figure 4.31: LR(0) automaton for the expression grammar (4.1)

SLR parsing: how it is better than LR(0) parsing?

- Conflicts can be reduced by

> If a is the next input, and we are in state I_i, then
>
> if $[A \to \alpha\cdot]$ is in I_i, then "reduce $A \to \alpha$" for all a in FOLLOW(A).

- But, if there is a transition in LR(0) automaton on the input *a* then ?
  - We conclude there is a conflict and the grammar is not SLR.

STACK	INPUT BUFFER
$\$\ldots\ \alpha$	$a\ldots\$$

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

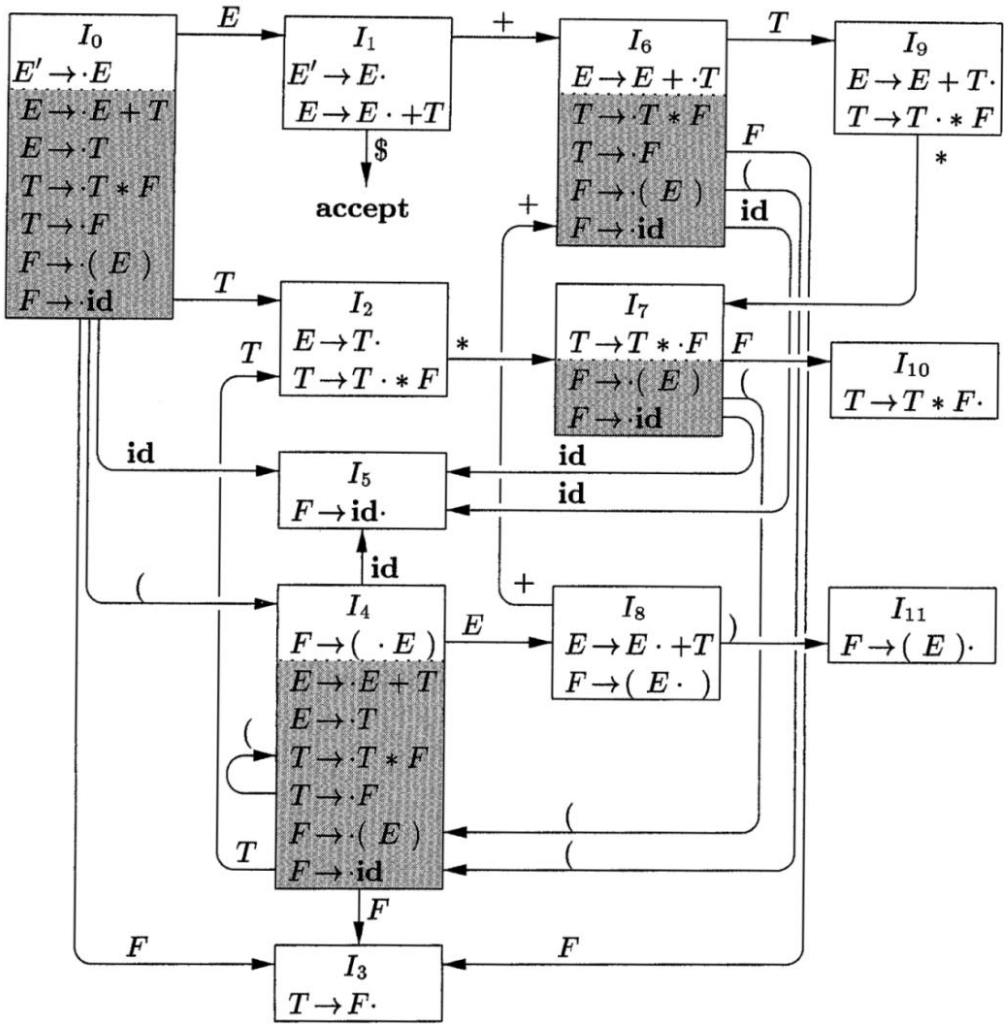


Figure 4.31: LR(0) automaton for the expression grammar (4.1)

	FOLLOW
E	\$, +,)
T	\$, +, *,)
F	\$, +, *,)

Note: * does not FOLLOW E.
Hence, while in state 2, on input *, we can not reduce T to E.
So, we must shift * on to stack.

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

SLR parsing example

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

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

LINE	STACK	SYMBOLS	INPUT	ACTION
(1)	0	\$	**id** * **id** \$	shift to 5
(2)	0 5	\$ **id**	* **id** \$	reduce by $F \to \mathbf{id}$
(3)	0 3	\$ F	* **id** \$	reduce by $T \to F$
(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 $F \to \mathbf{id}$
(7)	0 2 7 10	\$ T * F	\$	reduce by $T \to T * F$
(8)	0 2	\$ T	\$	reduce by $E \to T$
(9)	0 1	\$ E	\$	accept

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

Figure 4.37: Parsing table for expression grammar

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

	STACK	SYMBOLS	INPUT	ACTION
(1)	0	\$	**id** $*$ **id** $+$ **id** \$	shift
(2)	0 5	\$ **id**	$*$ **id** $+$ **id** \$	reduce by $F \to \mathbf{id}$
(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 $F \to \mathbf{id}$
(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 $F \to \mathbf{id}$
(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

Moves of an LR parser on **id** $*$ **id** $+$ **id**

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. This configuration represents the right-sentential form

$$X_1 X_2 \cdots X_m a_i a_{i+1} \cdots a_n$$

X_i is the grammar symbol represented by state s_i.

- All in edges on a state will have the same grammar symbol.
- This grammar symbol corresponds/represents the state.

Behavior of the LR Parser

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.

1. If ACTION$[s_m, a_i]$ = 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 ACTION$[s_m, a_i]$ = reduce $A \to \beta$, 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 β, and s = 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 GOTO$[s_{m-r}, A]$, onto the stack.

The current input symbol is not changed in a reduce move.

3. If ACTION$[s_m, a_i]$ = accept, parsing is completed.

4. If ACTION$[s_m, a_i]$ = 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. □

```
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

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 $\text{GOTO}(I_i, a) = I_j$, then set $\text{ACTION}[i, a]$ to "shift j." Here a must be a terminal.

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

   (c) If $[S' \to S \cdot]$ 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 $[S' \to \cdot S]$.

Look at this: if a is in FOLLOW(A) and there is a transition on a in the LR(0) automaton then shift/reduce conflict results, and the grammar is not a SLR(1) grammar.

- SLR is same as SLR(1)
- A grammar which can be parsed (which has SLR parsing table) by SLR parser is said to be SLR grammar.

- How detect whether a given grammar is SLR or not?

SLR or not?

$A \to \alpha\cdot$ is called a final item.

If we reached a state having a final item $A \to \alpha\cdot$

Then we are applying the reduction using $A \to \alpha$

This is not a mistake only if the current input terminal is in FOLLOW(A), and we do not have yet another item which causes either shift/reduce conflict or reduce/reduce conflict.

This characterizes the SLR grammars.

What LR(0) parser does:
Lookahead is not used as described above. As soon as a final item is reached it applies reduction (doesnot care about what is the lookahead).

Example: non-SLR grammar

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

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: $S \rightarrow L = R\cdot$

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

FOLLOW (R) contains $=$ (since, $S \Rightarrow L = R \Rightarrow *R = R$)

Assume that we are in state 2 and the next input is $=$

There is a shift/reduce conflict.