

BOTTOM-UP PARSING

Principles Of Bottom Up Parsing

In bottom-up parsing,

1. a parse tree is created from leaves upwards. The symbols in the input, after completion of successful parse, are placed at the leaf nodes of the tree.
2. Starting from the leaves, the parser fills in the internal nodes of the unknown parse tree gradually, eventually ending at the root.
3. The creation of an internal node involves replacing symbols from $(N \cup T)^*$ by a single nonterminal repeatedly. The task is to identify the rhs of a production rule in the tree constructed so far and replace it by the corresponding lhs of the rule. This kind of use of a production rule is called a **reduction**.
4. The crucial task, therefore, is to find the production rules that have to be used for reduction of a sentence to the start symbol of the grammar.

Q. Is there a derivation that corresponds to a bottom-up construction of the parse tree ?

The construction process indicates that it has to trace out some derivation in the reverse order (provided one such exists). The following example provides the answer.

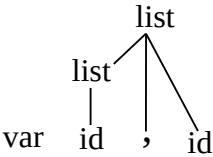
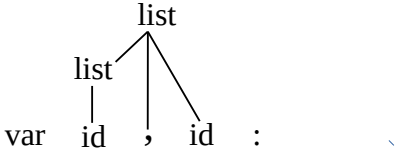
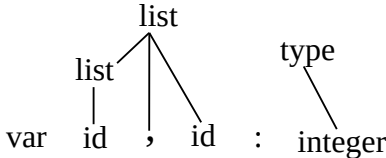
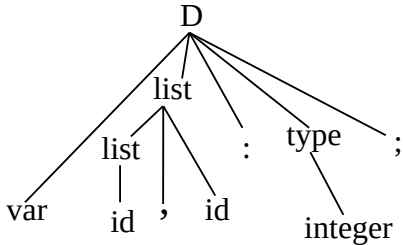
Example : Consider the following declaration grammar from Pascal language, which has 3 nonterminals and the terminals are marked in bold.

$D \rightarrow \text{var list} : \text{type} ;$
 $\text{type} \rightarrow \text{integer} \mid \text{real}$
 $\text{list} \rightarrow \text{list} , \text{id} \mid \text{id}$

The input string to be parsed is `var id, id : integer ;`

The reductions used and the parse tree construction from the leaves as the input string is processed is shown in Figure below.

Input	Production Rule used	Parse Tree
var id, id : integer ;	None	NULL
id, id : integer ;	None	var
, id : integer ;	list \rightarrow id	<pre>list var id</pre>
id : integer ;	None	<pre>list var id ,</pre>

Input	Production Rule used	Parse Tree
: integer ;	$\text{list} \rightarrow \text{list} , \text{id}$	 <pre> graph TD list1[list] --- list2[list] list1 --- comma1['] list1 --- id1[id] list2 --- var[var] list2 --- id2[id] </pre>
integer ;		 <pre> graph TD list1[list] --- list2[list] list1 --- comma1['] list1 --- id1[id] list1 --- colon1[:] list2 --- var[var] list2 --- id2[id] </pre>
;	$\text{type} \rightarrow \text{integer}$	 <pre> graph TD list1[list] --- list2[list] list1 --- comma1['] list1 --- id1[id] list1 --- colon1[:] list1 --- type1[type] list2 --- var[var] list2 --- id2[id] type1 --- integer1[integer] </pre>
empty	$D \rightarrow \text{var list : type ;}$	 <pre> graph TD D[D] --- var1[var] D --- list1[list] D --- colon2[:] D --- type1[type] D --- semicolon1['] list1 --- list2[list] list1 --- comma2['] list1 --- id1[id] list2 --- var2[var] list2 --- id2[id] type1 --- integer1[integer] </pre>

The sentential forms that were produced during the parse are as given below. In each sentential form the rhs of the rule that is used for reduction has been highlighted.

```

var id , id : integer ;
var list , id : integer ;
var list : integer ;
var list : type ;
D

```

The sentential forms, as you can easily verify, happen to be a right most derivation in the reverse order.

Bottom-Up parsing and rightmost derivation

Working out a rightmost derivation in reverse turns out to be a natural choice for these parsers.

1. Right most sentential forms have the property that all symbols beyond the rightmost nonterminal are terminal symbols.
2. If for a string of terminals w , there exists a n -step rm-derivation from S , that is,
 $S = \alpha_1 \Rightarrow \alpha_2 \Rightarrow \dots \Rightarrow \alpha_n = w$; then given α_n it should be possible to construct α_{n-1} .
3. The $(n-1)^{\text{th}}$ right sentential form, denoted by α_{n-1} above, contains exactly one nonterminal, say A , but the position of A in this sentential form is not known. However, examining the tokens of w (or α_n) from left to right, one at a time, till the rhs of A , say β , is found, and then performing a reduction by $A \rightarrow \beta$ should give us α_{n-1} .
4. In general, the construction of a previous right most sentential form from a given right most sentential form is similar in spirit. The essence of bottom up parsing is to identify the rhs of a rule in the sentential forms.
5. It might be instructive to examine why it is not easy for a bottom-up parser to use a leftmost derivation in reverse.

Example : Consider the expression grammar whose rules are given by

$$\begin{aligned} E &\rightarrow E+T \mid T \\ T &\rightarrow T*F \mid F \\ F &\rightarrow \text{id} \mid (E) \end{aligned}$$

along with the input to be parsed : $\text{id} + \text{id} * \text{id}$

Leftmost derivation of this sentence is :

$$\begin{aligned} E &\Rightarrow_{\text{lm}} E + T \Rightarrow_{\text{lm}} T + T \Rightarrow_{\text{lm}} F + T \Rightarrow_{\text{lm}} \text{id} + T \Rightarrow_{\text{lm}} \text{id} + T * F \Rightarrow_{\text{lm}} \text{id} + F * F \\ &\Rightarrow_{\text{lm}} \text{id} + \text{id} * F \Rightarrow_{\text{lm}} \text{id} + \text{id} * \text{id} \end{aligned}$$

Observe the following from the sequence of derivation above.

- Consider the input sentence $\text{id} + \text{id} * \text{id}$. There are three instances of **id** in the given input, that matches with the rhs of the rule, $F \rightarrow \text{id}$
- Reducing the first **id** in $\text{id} + \text{id} * \text{id}$ produces $F + \text{id} * \text{id}$ which is not the previous leftmost sentential form, $\text{id} + \text{id} * F$
- Reducing the second **id** in, $\text{id} + \text{id} * \text{id}$, produces $\text{id} + F * \text{id}$ which is also not the previous leftmost sentential form, $\text{id} + \text{id} * F$
- In order to produce the previous sentential, only the third **id** in, $\text{id} + \text{id} * \text{id}$, should be used in the reduction which yield the previous leftmost sentential form, $\text{id} + \text{id} * F$
- In summary, the parser would have several candidates for reduction (only one correct) and it is not easy to identify the right one.

Recall that the basic steps of a bottom-up parser are

1. Reduce the input string to the start symbol by performing a series of reductions.
2. The sentential forms produced while parsing must trace out a rm-derivation in reverse.

There are two main considerations to carry out a reduction,

- i. to identify a substring within a rm-sentential form which matches the rhs of a rule (there may be several such)
- ii. when this substring is replaced by the lhs of the matching rule, it must produce the previous rm-sentential form. Such substrings, being central to bottom up parsing, have been given the name **handle**. Bottom up parsing is essentially the process of detecting handles and using them in reductions. Different bottom-up parsers use different methods for handle detection.

Definition of a Handle

- A **handle** of a **right sentential form**, say γ , is a production rule $A \rightarrow \beta$, and a **position of β** in γ such that when β is replaced by A in γ , the resulting string is the **previous right sentential form** in a rightmost derivation of γ from S , the start nonterminal.
- Formally, if $S \Rightarrow_{rm}^* \alpha A w \Rightarrow_{rm} \alpha \beta w$, then rule $A \rightarrow \beta$, in the position following α in γ , is a handle of γ (that is $\alpha\beta w$).
- It should be clear that only terminal symbols can appear to the right of a handle in a rightmost sentential form.

Shift-Reduce Parser

A parsing method that directly uses the principle of bottom up parsing outlined above is known as a shift reduce parser. It is a generic name for a family of bottom- up parsers that employ this strategy.

A shift reduce parser requires the following data structures,

- a buffer for holding the input string to be parsed,
- a data structure for detecting handles (a stack happens to be adequate), and
- a data structure for storing and accessing the lhs and rhs of rules.

The parser operates by applying the steps 1 and 2 repeatedly, till a situation given by either of steps 3 or 4 is found.

Step 1. Moving symbols from the input buffer onto the stack, one symbol at a time. This move is called a **shift** action.

Step 2. Checking whether a handle occurs in the stack; if a handle is detected then a reduction by an appropriate rule is performed (which includes popping the rhs of a rule from the stack and then pushing the lhs of the rule), this move is called a **reduce** action.

Step 3. If the stack contains the start symbol only and input buffer is empty, then we announce a successful parse. This move is known as **accept** action.

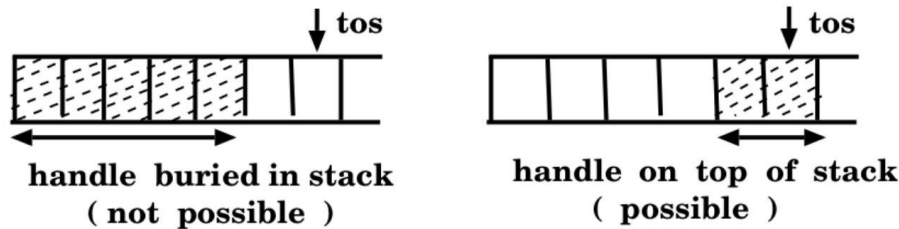
Step 4. A situation when the parser can neither shift nor reduce nor accept, it declares an error, which is known as **error** action and halts.

Remarks On Shift Reduce Parsing

1. Appending the stack contents with the unexpended input produces a rightmost sentential form.
2. Shift reduce parser as discussed above, is difficult to implement, since the details of when to shift and how to detect handles are not specified clearly.

3. A property that relates to the occurrence of handles in the parser stack is exploited by shift reduce parsers. If a handle can lie anywhere within the stack then handle detection is going to be expensive since all possible contiguous substrings in the stack are potential candidates for a handle.

4. It can be proved that a handle in a shift reduce parser always occurs on top of the stack (that is, the right end of the handle is the top of stack) and is never buried inside the stack.



Example : If the stack contents happens to be αAz , with z on tos , then possible handles are :

$z \quad Az \quad \alpha Az$

while substrings that are not handles are: $\alpha \quad \alpha A$

Handle Detection In Shift Reduce Parsing

Result : Handle in a shift reduce parser is never buried inside the stack.

Proof Outline : The result is easily verified for the last sentential form. Let $a_1 a_2 \dots a_n$ be the sentence to be parsed, and

$$S \Rightarrow_{\text{rm}}^* a_1 a_2 \dots a_i A a_{i+k+1} \dots a_n \Rightarrow_{\text{rm}} a_1 a_2 \dots a_n$$

The last two sentential forms (say m and $m-1$) are shown above, and $A \rightarrow a_{i+1} \dots a_{i+k}$ is the rule used. In this case shifting symbols upto a_{i+k} in the sentence, $a_1 a_2 \dots a_n$, would expose the handle on top of the stack. Similar arguments hold for the other possibilities of placement of A .

In the general case, consider the stack contents at some intermediate stage of parsing, say j^{th} sentential form, and assume the handle is on top of stack (tos) as shown below.

Stack	Input
αB	$x y z$

B is currently on tos (obtained after handle pruning from the $(j+1)^{\text{th}}$ sentential form). In order to construct the $(j-1)^{\text{th}}$ sentential form, the handle in the j^{th} sentential has to be detected. There are two possible forms for the $(j-1)^{\text{th}}$ sentential.

$(j-1)^{\text{th}}$ sentential	j^{th} sentential	Rule used
1. $\alpha B x A z$	$\alpha B x y z$	$A \rightarrow y$
2. $\delta A y z$	$\alpha B x y z$	$A \rightarrow \beta B x$

In case 1 above, after shifting x and y to the stack, the stack content will be $\alpha B x y$, and the handle y would be found on tos as claimed.

In case 2, after shifting x , the stack content will be $\alpha B x$, and if $\alpha = \delta \beta$, then stack contents are $\delta \beta B x$, and in this situation also the handle is on the top of stack, reiterating the fact that shifting of zero or more symbols allows the handle to occur on tos and get detected. This completes the proof outline.

The major implications of the above theoretical result are the following.

- Right end of a handle is guaranteed to occur at the top of the stack
- Left end of the handle would have to be found by examining as many symbols in the stack as present in the rhs of the rule.
- Search space for a handle is reduced considerably, at most linear in the size of the stack.
- Justifies the use of stack for detecting handles.

Illustrative Example : Consider the following extended grammar for expressions.

1, 2, 3) $E \rightarrow E + T \mid E - T \mid T$

4, 5, 6) $T \rightarrow T * F \mid T / F \mid F$

7, 8) $F \rightarrow P \uparrow F \mid P$

9, 10) $P \rightarrow -P \mid Q$

11, 12) $Q \rightarrow (E) \mid \text{id}$

The grammar rules are written by taking note of the operator properties. The symbol \uparrow denotes the exponentiation operator which is given to be right associative. There are two minus operators, the unary minus and the binary minus. Except for \uparrow , all the other operators are left associative. The operators, in the order from higher to lower precedence are : {unary minus ($-$); \uparrow ; $[\ast, /]$; $[\ast, /]$; $[\ast, /]$ }, where operators within square braces have the same precedence. The grammar given above is unambiguous and uses the empirical rules mentioned earlier – a) one nonterminal for every class of operators of same precedence, b) left or right recursive grammar for an operator that is respectively left or right associative.

Let us try an intuitive parsing of the string : $- a - b \ast c \uparrow d \uparrow e$ using a shift reduce parser. The parser shifts a token to the stack if the stack contents are part of a handle that has been partially seen. It reduces when a handle appears on tos. The moves of such a parser are illustrated as parsing proceeds. We show the stack contents with the tos marker at its right end. The input just before the parser move is shown. The parser action on the given configuration (stack, nexttoken) is mentioned. The execution of this action may either change the stack or the nexttoken or both. The new configuration after the action is shown in the next row in the following table.

The initial configuration comprises an empty stack (the symbol $\$$ is placed on the stack to indicate its bottom) and the input stream is appended with the symbol $\$$ to detect the end of input. The tos element is highlighted for ease in readability. The nexttoken is highlighted for the same reason.

Move	Stack contents	Input stream	Parser Action	Remarks
1	\$	$- a - b \ast c \uparrow d \uparrow e \$$	Shift $-$	Both input and stack change. The token $-$ is part of rhs of rule 9

Move	Stack contents	Input stream	Parser Action	Remarks
2	\$ -	a - b * c ↑ d ↑ e \$	Shift a	Both input and stack change.
3	\$ - id	- b * c ↑ d ↑ e \$	Reduce by rule 12	Stack changes
4	\$ - Q	- b * c ↑ d ↑ e \$	Reduce by rule 10	Stack changes
5	\$ - P	- b * c ↑ d ↑ e \$	Reduce by rule 9	Stack changes
6	\$ P	- b * c ↑ d ↑ e \$	Reduce by rule 8	Stack changes
7	\$ F	- b * c ↑ d ↑ e \$	Reduce by rule 6	Stack changes
8	\$ T	- b * c ↑ d ↑ e \$	Reduce by rule 3	Stack changes
9	\$ E	- b * c ↑ d ↑ e \$	Shift -	Both change
10	\$ E -	b * c ↑ d ↑ e \$	Shift b	Both change
11	\$ E - id	* c ↑ d ↑ e \$	Reduce by rule 12	Stack changes
12	\$ E - Q	* c ↑ d ↑ e \$	Reduce by rule 10	Stack changes
13	\$ E - P	* c ↑ d ↑ e \$	Reduce by rule 8	Stack changes
14	\$ E - F	* c ↑ d ↑ e \$	Reduce by rule 6	Stack changes
15	\$ E - T	* c ↑ d ↑ e \$	Shift *	Both change
16	\$ E - T *	c ↑ d ↑ e \$	Shift c	Both change
17	\$ E - T * id	↑ d ↑ e \$	Reduce by rule 12	Stack changes
18	\$ E - T * Q	↑ d ↑ e \$	Reduce by rule 10	Stack changes
19	\$ E - T * P	↑ d ↑ e \$	Shift ↑	Both change
20	\$ E - T * P ↑	d ↑ e \$	Shift d	Both change
21	\$ E - T * P ↑ id	↑ e \$	Reduce by rule 12	Stack changes
22	\$ E - T * P ↑ Q	↑ e \$	Reduce by rule 10	Stack changes
23	\$ E - T * P ↑ P	↑ e \$	Shift ↑	Both change
24	\$ E - T * P ↑ P ↑	e \$	Shift e	Both change
25	\$ E - T * P ↑ P ↑ id	\$	Reduce by rule 12	Stack changes
26	\$ E - T * P ↑ P ↑ Q	\$	Reduce by rule 10	Stack changes
27	\$ E - T * P ↑ P ↑ P	\$	Reduce by rule 8	Stack changes
28	\$ E - T * P ↑ P ↑ F	\$	Reduce by rule 7	Stack changes
29	\$ E - T * P ↑ F	\$	Reduce by rule 7	Stack changes
30	\$ E - T * F	\$	Reduce by rule 4	Stack changes
31	\$ E - T	\$	Reduce by rule 2	Stack changes
32	\$ E	\$	Accept	Successful Parse

Explanation of a few selected moves of the shift reduce parser in the Example. Many of the parser moves may appear to be like black magic because there were alternative moves that were not taken. The fact is that the moves are not all adhoc, instead they are carefully guided based on analyses of the grammar rules. Insights with reasons about a few selected moves, chosen by the parser, from the above table are presented below.

1. Why did the parser in move 3 choose reduce by rule 12 instead of shifting the lookahead symbol ‘-’ to the stack ? The stack content at that point was “- id”. The only rule whose rhs matches partially with the stack content is “ $P \rightarrow -P$ ”. The parser therefore knows that to get to this handle, id has to be reduced to P, if such is possible. It therefore chooses the rule 12 to reduce “id” to “Q” resulting in the string “- Q” in the stack. However since “- Q” is not a handle, parser reduces this string “-id” by a series of reductions, using the rules, $Q \rightarrow id$ and $P \rightarrow Q$. As the handle “- P” gets exposed at tos in move 5 the parser uses the rule “ $P \rightarrow -P$ ” for reduction.
2. In move 6, the parser has the configuration (P, -) with P on tos and “-” as the lookahead. The parser does not shift “-”, since there is no handle with “P -” as a substring. However the parser is aware that “E - T” is a handle because of rule 2. Therefore it initiates a series of reductions, $F \rightarrow P$, $T \rightarrow F$, $E \rightarrow T$, via the moves 6 to 8 to ensure that the part “E”, of the handle “E -”, has been placed on the stack.

Reasoning for the other parser moves are left as an exercise for the reader.

Parse Tree Construction

A parse tree can be easily constructed by a shift reduce parser as it proceeds with parsing.

- Let the stack be augmented to include an extra field for each entry, a pointer to a tree associated with the symbol, on the stack. The code for tree construction can then be easily added to the basic moves of the parser as outlined below.
- shift a ; create a leaf labeled a; the root of this tree is associated with the stack symbol, a
- reduce by the rule : $A \rightarrow X_1 X_2 \dots X_n$; A new node labeled A is created ; sub-trees rooted at X_1, \dots, X_n are made its children in the left to right order. The pointer for this tree rooted at A is stored along with A in the stack.
- reduction by a rule, $A \rightarrow \epsilon$, is handled by creating a tree with root labeled A which has a leaf labeled with ϵ , as its only child. The pointer to the root of this tree is saved along with A in the stack.

Limitations Of Shift Reduce Parser

Q. What kind of grammars (languages) would not admit a shift reduce parser?

Consider the following situations.

1. a handle , say β , occurs at tos; the nexttoken is a, and βa happens to be another handle. The parser has two options for proceeding further.
 - Reduce the handle using $A \rightarrow \beta$. In such a case, the other handle , $B \rightarrow \beta a$, may not get exposed for reduction as β is not present in the stack anymore, and hence this longer handle may be missed for good.
 - Ignore the handle β present on the stack, shift a on to the stack and continue parsing. In this case, reduction using $B \rightarrow \beta a$ would be possible, because this handle may appear on the stack soon in the future. The price to pay is that of missing the handle β .

The situation above is described in the parser as a **shift reduce** conflict.

- The handle β , which has just appeared on the tos is such that it matches with the rhs of two or more rules, say $A \rightarrow \beta$ and $B \rightarrow \beta$. The the parser in this case has two or more reduce possibilities. This situation is known as a **reduce reduce** conflict.

Q. How do shift reduce parsers handle shift reduce and reduce reduce conflicts ?

The nexttoken is used by the parser to prefer one move over the other. The common wisdom is to choose shift (over reduce) in the case of a shift reduce conflict. The rationale given is that a longer handle is more desirable than the shorter one, since it is a substring of the longer handle. In the event of a reduce reduce conflict, parser designer specifies a tie breaking method to select one among the competing rules.

With this background information about a shift reduce parser, we begin the design of a family of deterministic shift reduce parsers, known as LR parsers.

LR Parsers : Some Facts

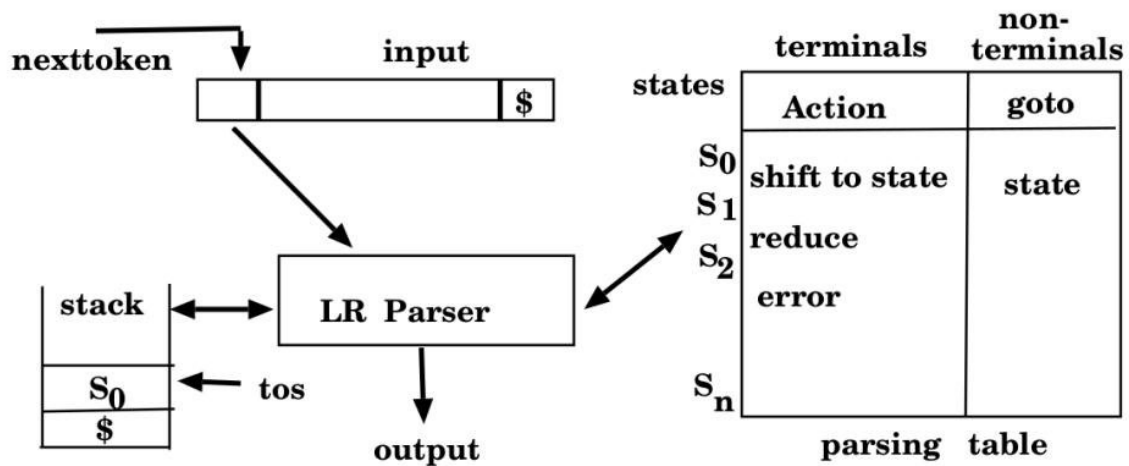
1. All the parsers in the family are shift reduce parsers, the basic parsing actions are shift, reduce, accept and error.
2. These are the most powerful of all deterministic parsers in practice; they subsume the class of grammars that can be parsed by predictive parsers.
3. LR parsers can be written to recognize most of the PL constructs which can be expressed through a cfg.
4. The overall structure of all the parsers is the same. All are table driven parsers, the information content in the respective tables distinguish the members of the family,

$$SLR(1) \leq LALR(1) \leq LR(1)$$

where $p \leq q$ means that q parses a larger class of grammars as compared to that parsed by p .

Structure of a LR parser

The components of a LR parser and their interaction is given in the figure. The parser, apart from grammar symbols, uses extra symbols (called states). While both grammar and state symbols can appear on the stack, the parsers are so designed that tos , is always a state.



Top of Stack (tos)	nexttoken	Parser Action	Realization of Action
State j (S_j)	a	Shift to state i (si)	push a; push S_i ; continue
	a	Reduce by rule j (rj)	$r_j : A \rightarrow \alpha$; $ \alpha = r$ (rhs has r symbols) pop 2r symbols from stack; tos = S_k (state on tos after popping) let goto [S_k, a] be S_m ; push A; push S_m ; continue
	\$	Accept (acc)	Successful parse; Halt
	a	error	Issue error message and handle error

Working of a LR Parser

Explanation Of Terms Used In Figure

The parser uses two data structures prominently.

1. A stack which contains strings of the form : $s_0X_1s_1X_2 \dots X_ms_m$, where X_i is a grammar symbol and s_i is a special symbol called a state.
2. A parsing table which comprises two parts, usually named as, Action and Goto.
 - The Action part is a table of size $n \times m$ where n is the total number of states of the parser and $m = |T|$ (including \$). The possible entries are

- a) s_i which means shift to state i
- b) r_j which stands for reduce by the j^{th} rule,
- c) accept
- d) error

- The Goto part of the parsing table is of size $n \times p$, where $p = |N|$. The only interesting entries are state symbols.

The LR parser is a driver routine which

- i) initializes the stack with the start state and calls scanner to get a token (nexttoken).
- ii) For any configuration, as indicated by (tos , nexttoken), it consults the parsing table and performs the action specified there.
- iii) The goto part of the table is used only after a reduction.
- iv) The parsing continues till either an error or accept entry is encountered.

Simple LR(1) Parser

We start by examining the parser of the family which is known as Simple LR(1) or SLR(1) parser. SLR(1) Parsing Table for an expression grammar is given below. The working of this parser for a sample input is shown after the parsing table. Consider the simple expression Grammar again.

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

SLR(1) Parsing table for this grammar follows.

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			

Few key observations on the table structure and its contents for the given example. The table is of size 11 x 9, where the 11 rows record the details of each of the 11 states of the SLR(1) automaton, numbered from 0 to 10. The Action part of the table has 6 columns, one for each of 5 terminals of the grammar and an additional column for \$ (end-of-input-marker). The Goto part of the table has 3 columns, one for each of the nonterminals of G. To summarize, the size of SLR(1) table of size $n \times m$, where n is the number of states, and $m = |T| + |N| + 1$.

We shall explain later how the SLR(1) parsing table was constructed for this grammar. The use of table in parsing is illustrated with an example.

To illustrate the working of the parsing table on an input, consider the string : $a + b * (c + d) * e$

The 1st column counts the moves made by a SLR(1) parser. The 2nd column gives the stack contents with the tos at the right end, the 3rd column gives the unexpended input with the leftmost symbol as the current token, the 4th and 5th columns give the Action and Goto table entries respectively. The table entries are to be read in the following manner.

- Given the stack contents and the current token in the input, we consult the table with the configuration, [tos, nexttoken]. When tos is a state symbol, we consult the Action table.
- If the Action table entry is a shift move, the nexttoken is shifted to the stack and the new state is placed on tos.
- When the Action table entry is reduce by a rule, say $A \rightarrow \alpha$, twice the number of symbols present in the rhs α , that is $2*|\alpha|$ symbols are popped off the stack, $|\alpha|$ symbols of the rule and $|\alpha|$ state symbols that follow the grammar rules. After popping $2*|\alpha|$ symbols from the stack, a state symbol say t is exposed on the tos. The lhs nonterminal A is pushed on the stack after the state symbol t so that A now becomes the tos. A reduce results in a nonterminal getting placed on tos. This is the only situation when tos is not a state symbol and this needs to be rectified which is done by the next move.
- The Goto table is consulted for the entry $\text{Goto}[t, A]$ which gives a state say u . Then u is pushed on the stack with u becoming the tos, restoring a state on tos.

The details are presented in the following table.

Moves	Stack	Input	Action Table Entry	Goto Table Entry
1 (initial)	\$ 0	$a + b * (c + d) * e \$$	Shift to state 5	
2	\$ 0 id 5	$+ b * (c + d) * e \$$	Reduce by rule 6	
3	\$ 0 F	$+ b * (c + d) * e \$$		$\text{Goto}[0, F] = 3$
4	\$ 0 F 3	$+ b * (c + d) * e \$$	Reduce by rule 4	
5	\$ 0 T	$+ b * (c + d) * e \$$		$\text{Goto}[0, T] = 2$
6	\$ 0 T 2	$+ b * (c + d) * e \$$	Reduce by rule 2	
7	\$ 0 E	$+ b * (c + d) * e \$$		$\text{Goto}[0, E] = 1$
8	\$ 0 E 1	$+ b * (c + d) * e \$$	Shift to state 6	
9	\$ 0 E 1 + 6	$b * (c + d) * e \$$	Shift to state 5	
10	\$ 0 E 1 + 6 id 5	$* (c + d) * e \$$	Reduce by rule 6	
11	\$ 0 E 1 + 6 F	$* (c + d) * e \$$		$\text{Goto}[6, F] = 3$
12	\$ 0 E 1 + 6 F 3	$* (c + d) * e \$$	Reduce by rule 4	

Moves	Stack	Input	Action Table Entry	Goto Table Entry
13	\$ 0 E 1 + 6 T	* (c + d) * e \$		Goto[6, T] = 9
14	\$ 0 E 1 + 6 T 9	* (c + d) * e \$	Shift to state 7	
15	\$ 0 E 1 + 6 T 9 * 7	(c + d) * e \$	Shift to state 4	
16	\$ 0 E 1 + 6 T 9 * 7 (4	c + d) * e \$	Shift to state 5	
17	\$ 0 E 1 + 6 T 9 * 7 (4 id 5	+ d) * e \$	Reduce by rule 6	
18	\$ 0 E 1 + 6 T 9 * 7 (4 F	+ d) * e \$		Goto[4, F] = 3
19	\$ 0 E 1 + 6 T 9 * 7 (4 F 3	+ d) * e \$	Reduce by rule 4	
20	\$ 0 E 1 + 6 T 9 * 7 (4 T	+ d) * e \$		Goto[4, T] = 2
21	\$ 0 E 1 + 6 T 9 * 7 (4 T 2	+ d) * e \$	Reduce by rule 2	
22	\$ 0 E 1 + 6 T 9 * 7 (4 E	+ d) * e \$		Goto[4, E] = 8
23	\$ 0 E 1 + 6 T 9 * 7 (4 E 8	+ d) * e \$	Shift to state 6	
24	\$ 0 E 1 + 6 T 9 * 7 (4 E 8 + 6	d) * e \$	Shift to state 5	
25	\$ 0 E 1 + 6 T 9 * 7 (4 E 8 + 6 id 5) * e \$	Reduce by rule 6	
26	\$ 0 E 1 + 6 T 9 * 7 (4 E 8 + 6 F) * e \$		Goto[6, F] = 3
27	\$ 0 E 1 + 6 T 9 * 7 (4 E 8 + 6 F 3) * e \$	Reduce by rule 4	
28	\$ 0 E 1 + 6 T 9 * 7 (4 E 8 + 6 T) * e \$		Goto[6, T] = 9
29	\$ 0 E 1 + 6 T 9 * 7 (4 E 8 + 6 T 9) * e \$	Reduce by rule 1	
30	\$ 0 E 1 + 6 T 9 * 7 (4 E) * e \$		Goto[4, E] = 8
31	\$ 0 E 1 + 6 T 9 * 7 (4 E 8) * e \$	Shift to state 11	
32	\$ 0 E 1 + 6 T 9 * 7 (4 E 8) 11	* e \$	Reduce by rule 5	
33	\$ 0 E 1 + 6 T 9 * 7 F	* e \$		Goto[7, F] = 10
34	\$ 0 E 1 + 6 T 9 * 7 F 10	* e \$	Reduce by rule 3	
35	\$ 0 E 1 + 6 T	* e \$		Goto[6, T] = 9
36	\$ 0 E 1 + 6 T 9	* e	Shift to state 7	
37	\$ 0 E 1 + 6 T 9 * 7	e \$	Shift to state 5	
38	\$ 0 E 1 + 6 T 9 * 7 id 5	\$	Reduce by rule 6	
39	\$ 0 E 1 + 6 T 9 * 7 F	\$		Goto[7, F] = 10
40	\$ 0 E 1 + 6 T 9 * 7 F 10	\$	Reduce by rule 3	
41	\$ 0 E 1 + 6 T	\$		Goto[6, T] = 9
42	\$ 0 E 1 + 6 T 9	\$	Reduce by rule 1	
43	\$ 0 E	\$		Goto[0, E] = 1
44	\$ 0 E 1	\$	Accept	
Parsing terminates with success				

The parser recognizes the input as a valid string of the grammar after 44 moves.

Configuration of a LR Parser

A configuration of a LR parser is defined by a tuple, (stack contents , unexpended part of input).

Initial configuration is shown by $(s_0, a_1a_2 \dots a_n\$)$, where s_0 is the designated start state and the second component is the entire sentence to be parsed.

Let an intermediate configuration be given by $(s_0X_1s_1 \dots X_is_i, a_{j+1} \dots a_n\$)$, which shows that the state s_i is on top, and the unexpended input is, $a_{j+1} \dots a_n$, then resulting configuration

i) after a shift action is given by $(s_0X_1s_1 \dots X_is_ia_js_k, a_{j+1} \dots a_n\$)$ provided $Action[s_i, a_j] = s_k$; both stack and nexttoken change after the shift, and

ii) after a reduce action is given by $(s_0X_1s_1 \dots X_{i-r}s_{i-r}As, a_j \dots a_n\$)$, where $Action[s_i, a_j] = r_m$; and the rule $r_m : Action[s_i, a_j] : A \rightarrow \beta$, β has r grammar symbols and $goto(s_{i-r}, A) = s$. Only the stack changes here.

SLR(1) Parser Construction

We now study SLR(1) parsing table construction.

1. The relevant definitions are introduced first.
2. The construction process is then explained in full detail.
3. Theory of LR parsing which provides the basis for the construction process and also shows its correctness is addressed last.

Definition of LR(0) Item and Related Terms

- **LR(0) item** : An LR(0) item for a grammar G is a production rule of G with the symbol \bullet (read as dot or bullet) inserted at some position in the rhs of the rule.
- Example of LR(0) items : Consider the rule given below.

$decls \rightarrow decls decl$

the possible LR(0) items are :

I1 : $decls \rightarrow \bullet decls decl$

I2 : $decls \rightarrow decls \bullet decl$

I3 : $decls \rightarrow decls decl \bullet$

The rule $decls \rightarrow \epsilon$ has only one LR(0) item,

I4 : $decls \rightarrow \bullet$

- **Incomplete and Complete LR(0) items** : An LR(0) item is complete if the \bullet is the last symbol in the rhs, else it is an incomplete item.

For every rule, $A \rightarrow \alpha$, $\alpha \neq \epsilon$ there is only one complete item, $A \rightarrow \alpha\bullet$ but as many incomplete items as there are grammar symbols in the rhs.

Example : I3 and I4 are complete items and I1 and I2 are incomplete items.

- **Augmented Grammar** : From the grammar, $G = (N, T, P, S)$, we create a new grammar $G' = (N', T, P, S')$, where $N' = N \cup \{S'\}$, S' is the start symbol of G' and $P' = P \cup \{S' \rightarrow S\}$. Note that G' is equivalent to G , as we can formally prove that $L(G) = L(G')$.
- **Kernel and Nonkernel items** : An LR(0) item is called a kernel item, if the dot is not at the left end. However the item $S' \rightarrow \bullet S$, which is introduced by the new rule, $S' \rightarrow S$, is an exception and is defined to be a kernel item.

An LR(0) item which has dot at the left end is called a nonkernel item.

Example :

I1 : $\text{decls} \rightarrow \bullet \text{decls decl}$ is a nonkernel item

I2 : $\text{decls} \rightarrow \text{decls} \bullet \text{decl}$ is a kernel item

I3 : $\text{decls} \rightarrow \text{decls decl} \bullet$ is a kernel item

I4 : $\text{decls} \rightarrow \bullet$ is a nonkernel item

Note that if we augment the grammar above with a new start symbol, say D , and then add the rule $\{D \rightarrow \text{decls}\}$, then this rule produces the LR(0) item, $\{D \rightarrow \bullet \text{decls}\}$, which is a special kernel item, as stated above.

Canonical Collection of LR(0) Items

- Functions closure and goto : Two functions closure and goto are defined which are used to create a set of items from a given item.

Let U be the collection of all LR(0) items of a cfg G . The closure function, f , is of the form $f : U \rightarrow 2^U$ and is constructed as follows.

- i. $\text{closure}(I) = \{I\}$, for $I \in U$
- ii. If $A \rightarrow \alpha \bullet B\beta \in \text{closure}(I)$, then for every rule of the form $B \rightarrow \eta$, the item $B \rightarrow \bullet \eta$ is added (if it is not already there) to $\text{closure}(I)$.
- iii. Apply step (ii) above repeatedly till no more new items can be added to $\text{closure}(I)$.

It can be seen that $\text{closure}(I) \neq \{I\}$ only when I is an item in which a nonterminal immediately follows the dot.

- The goto function, g , has the form $g : U \times X \rightarrow 2^U$, where X is a grammar symbol. The function goto is defined using closure as follows.

$$\text{goto}(A \rightarrow \alpha \bullet X\beta, X) = \text{closure}(A \rightarrow \alpha X \bullet \beta).$$

Clearly $\text{goto}(I, X)$ for a complete item I is Φ , because there are no symbols to the right of the bullet (\bullet) for any X .

• Example of closure and goto sets construction

Consider the rule : $A \rightarrow Aa \mid b$ which gives 2 LR(0) items, $A \rightarrow \bullet Aa$ and $A \rightarrow \bullet b$

- Now $\text{closure}(A \rightarrow \bullet Aa) = \{ A \rightarrow \bullet Aa \}$, by rule (i). The item $A \rightarrow \bullet b$ is added to the closure by rule (ii) giving $\text{closure}(A \rightarrow \bullet Aa) = \{ A \rightarrow \bullet Aa, A \rightarrow \bullet b \}$, and since no other LR(0) can be added, after step (iii), the final content is $\text{closure}(A \rightarrow \bullet Aa) = \{ A \rightarrow \bullet Aa, A \rightarrow \bullet b \}$.
- Using the closure set constructed above, we get $\text{goto}(A \rightarrow \bullet Aa, A) = \text{closure}(A \rightarrow A \bullet a) = \{ A \rightarrow A \bullet a \}$

The functions $\text{closure}()$ and $\text{goto}()$ were constructed for a single LR(0) item. These sets can be constructed to a set S of LR(0) items by appropriate generalizations, as shown below.

$$\text{closure}(S) = \bigcup_{I \in S} \{ \text{closure}(I) \}; \text{ and } \text{goto}(S, X) = \bigcup_{I \in S} \{ \text{goto}(I, X) \}$$

Algorithm given below give pseudo code for computing both of these functions.

Given a cfg, the collection U of all LR(0) items is well defined.

A particular collection C of sets of items i. e., $C \in 2^U$, is of specific interest and is called the canonical collection of LR(0) items. This collection is constructed using $\text{closure}()$ and $\text{goto}()$ as given in the following Algorithm.

Canonical Collection of LR(0) Items

Algorithm for construction of sets of items

procedure items(G', C)

begin

$i = 0$; $I_0 = \{ \text{closure}(S' \rightarrow \bullet S) \}$; $C = I_i$;

repeat

for each set of items I_i in C and each grammar symbol X ,

such that $\text{goto}(I_i, X) \neq \Phi$ and $\text{goto}(I_i, X) \notin C$, **do**

$i++$; $I_i := \text{goto}(I_i, X)$;

$C := C \cup I_i$

until no more sets of items can be added to C

end

```

procedure closure(I); { I is a set of items }
begin
  repeat
    for each item  $A \rightarrow \alpha \bullet B\beta \in I$ 
      and each rule  $B \rightarrow \gamma$  such that  $B \rightarrow \bullet\gamma$  is not in I
    do add  $B \rightarrow \bullet\gamma$  to I;
  until no more items can be added to I ;
  return I ;
end

function goto(I, X)
begin
  goto items :=  $\Phi$  ;
  for each item  $A \rightarrow \alpha \bullet X\beta \in I$ ,
    such that  $A \rightarrow \alpha X \bullet \beta$  is not in goto items
  do add  $A \rightarrow \alpha X \bullet \beta$  to goto items
  goto := closure(goto items);
  return goto ;
end

```

Illustration of Construction of Canonical Collection

Consider the expression grammar augmented with $E' \rightarrow E$.

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

I_0 is closure ($E' \rightarrow \bullet E$). The items $I_1 = \text{goto}(I_0, E)$ are added to I_0 . The last item in turn causes the addition of $T \rightarrow \bullet T * F$ and $T \rightarrow \bullet F$ to I_0 . The item $T \rightarrow \bullet F$ leads to the addition of $F \rightarrow \bullet (E)$ and $F \rightarrow \bullet \text{id}$. No more items can now be added and the collection I_0 is therefore complete and is shown in the figure that follows.

I_0 has several items having $\bullet\alpha$ combination for $\alpha = \{ E, T, F, (, \text{id} \}$. The set $\text{goto}(I_0, \alpha)$ is defined for all these values of α which in turn give rise to different collections as shown. For example, we choose the following names of the new collections arising out of I_0 . $I_1 = \text{goto}(I_0, E)$; $I_2 = \text{goto}(I_0, T)$; $I_3 = \text{goto}(I_0, F)$; $I_4 = \text{goto}(I_0, ()$; $I_5 = \text{goto}(I_0, \text{id})$, and so on. The rest of the collections are given below.

<div> $E' \rightarrow \bullet E$ $E \rightarrow \bullet E + T$ $E \rightarrow \bullet T$ $T \rightarrow \bullet T * F$ $T \rightarrow \bullet F$ $F \rightarrow \bullet (E)$ $F \rightarrow \bullet \text{id}$ </div> <div>I_0</div>	<div> $E' \rightarrow E \bullet$ $E \rightarrow E \bullet + T$ </div> <div>$I_1 = \text{goto}(I_0, E)$</div>	<div> $E \rightarrow T \bullet$ $T \rightarrow T \bullet * F$ </div> <div>$I_2 = \text{goto}(I_0, T)$</div>	<div> $T \rightarrow F \bullet$ </div> <div>$I_3 = \text{goto}(I_0, F)$</div>	<div> $F \rightarrow (\bullet E)$ $E \rightarrow \bullet E + T$ $E \rightarrow \bullet T$ $T \rightarrow \bullet T * F$ $T \rightarrow \bullet F$ $F \rightarrow \bullet (E)$ $F \rightarrow \bullet \text{id}$ </div> <div>$I_4 = \text{goto}(I_0, ()$</div>
	<div> $F \rightarrow \text{id} \bullet$ </div> <div>$I_5 = \text{goto}(I_0, \text{id})$</div>			

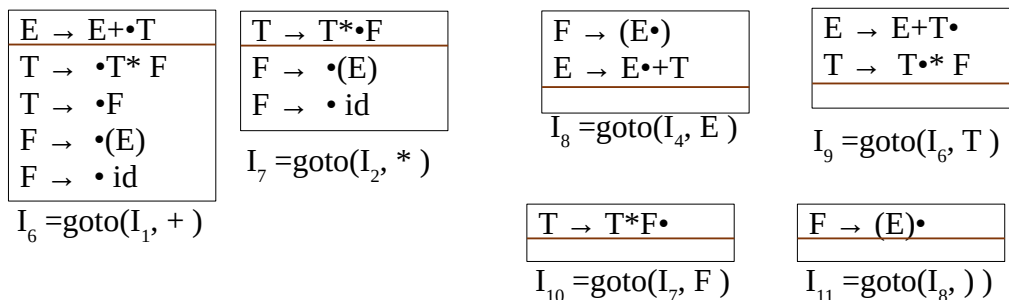


Figure : Canonical Collection of LR(0) items organized into states

Construction of SLR(1) Parsing Table

From the collection C constructed using the Algorithm, one can directly construct the SLR(1) parsing table. The procedure is described in the form of an algorithm.

Algorithm for SLR(1) Parsing Table

Input : Grammar G

Output : Action and goto part of the parsing table

1. From the input grammar G, construct an equivalent augmented grammar G'.
2. Use the algorithm for constructing FOLLOW sets to compute FOLLOW(A), $\forall A \in N'$.
3. Call procedure items(G', C) to get the desired canonical collection $C = \{I_0, I_1, \dots, I_n\}$.
4. Choose as many state symbols as the cardinality of C. We use numbers 0 through n to represent states.
5. The set I_i and its constituent items define the entries for state i of the Action table and the Goto table.

For each I_i , $0 \leq i \leq n$ do steps 5.1 through 5.6 given below.

5.1 If $A \rightarrow \alpha \bullet a \beta \in I_i$ and $\text{goto}(I_i, a) = I_j$, then $\text{Action}[i, a] = s_j$, i. e., shift to state j

5.2 If $A \rightarrow \alpha \bullet \in I_i$, then $\text{Action}[i, a] = r_j$, $\forall a \in \text{FOLLOW}(A)$ which means reduce by the j^{th} rule

$$A \rightarrow \alpha$$

5.3 If I_i happens to contain the item $S' \rightarrow S\bullet$, then $\text{Action}[i, \$] = \text{accept}$

5.4 All remaining entries of state i in the Action table are marked as error

5.5 For nonterminals A, such that $\text{goto}(I_i, A) = I_j$ create $\text{Goto}[i, A] = j$

5.6 The remaining entries in the Goto table are marked as error.

Note : The Goto entries of the state i of the parsing table are decided by the $\text{goto}(I_i, A)$ states, where A is a nonterminal.

The SLR(1) table for the canonical collection is partially constructed in the following.

Remarks : 1. initial or start state of the parser is the state corresponding to the set I_0 which contains the kernel item $S' \rightarrow \bullet S$

2. The parsing table is conflict free since there is no multiple entry (shift-reduce and/or reduce-reduce) in it.

Illustration of Table Construction Algorithm

The steps of the algorithm are traced out to fill up the first 3 states of the parsing table. Let us examine the LR(0) items in state 0.

$E' \rightarrow \bullet E$ $E \rightarrow \bullet E + T$ $E \rightarrow \bullet T$ $T \rightarrow \bullet T * F$ $T \rightarrow \bullet F$ $F \rightarrow \bullet (E)$ $F \rightarrow \bullet id$	$E' \rightarrow E \bullet$ $E \rightarrow E \bullet + T$ $I_1 = \text{goto}(I_0, E)$	$E \rightarrow T \bullet$ $T \rightarrow T \bullet * F$ $I_2 = \text{goto}(I_0, T)$	$T \rightarrow F \bullet$ $I_3 = \text{goto}(I_0, F)$	$F \rightarrow (\bullet E)$ $E \rightarrow \bullet E + T$ $E \rightarrow \bullet T$ $T \rightarrow \bullet T * F$ $T \rightarrow \bullet F$ $F \rightarrow \bullet (E)$ $F \rightarrow \bullet id$
I_0	$I_5 = \text{goto}(I_0, id)$			$I_4 = \text{goto}(I_0, ()$

- There are 2 items with \bullet followed by a terminal, namely $F \rightarrow \bullet (E)$ and $F \rightarrow \bullet id$. These 2 items result in 2 shift moves in the Action part of table. The target states are dictated by $\text{goto}(I_0, id) = I_5$ and $\text{goto}(I_0, () = I_4$. The table at this point is drawn below.

State	Action						Goto		
	id	+	*	()	\$	E	T	F
0	s5			s4					

- There are 5 items where \bullet is followed by a nonterminal, namely $E' \rightarrow \bullet E$; $E \rightarrow \bullet E + T$; $E \rightarrow \bullet T$; $T \rightarrow \bullet T * F$; and $T \rightarrow \bullet F$. These items result in 3 entries in the Goto table determined by the states obtained using $\text{goto}(I_0, E) = I_1$; $\text{goto}(I_0, T) = I_2$ and $\text{goto}(I_0, F) = I_3$ which causes 3 entries in the Goto part of the table as shown below.

State	Action						Goto		
	id	+	*	()	\$	E	T	F
0	s5			s4			1	2	3

- Since there are no complete LR(0) items (items with \bullet at the end) in I_0 , no reduce actions are possible in this state. The row for state 0 is complete at this point.

Let us consider filling up the entries of the table for state 1. This state has 2 LR(0) items, $E' \rightarrow E \bullet$ and $E \rightarrow E \bullet + T$. The item, $E \rightarrow E \bullet + T$, is a case of a terminal following the \bullet and hence causes an entry in the Action part determined by $\text{goto}(I_1, +) = I_6$, that is shift to state 6 on +.

State	Action						Goto		
	id	+	*	()	\$	E	T	F
0	s5			s4			1	2	3
1		s6							

The other LR(0) item, $E' \rightarrow E\bullet$, is a complete item and calls for reduce by this rule. This being a special reduce to the start symbol of the augmented grammar is the accept action which is always entered against \$. The table changes to the following at this point.

State	Action						Goto		
	id	+	*	()	\$	E	T	F
0	s5			s4			1	2	3
1		s6				acc			

This completes the processing for state 1. Note that there are no entries in Goto part of this table as there are no LR(0) items in state 1 where the \bullet is followed by a nonterminal.

As a final illustration, we consider state 2 and its contents to lay out the row for state 2. The LR(0) items in state 2 are $\{E \rightarrow T\bullet, T \rightarrow T\bullet *F\}$. The item, $T \rightarrow T\bullet *F$, causes a shift to the state goto($I_2, *$) which is I_7 . This entry for state 2 is placed into the table below.

State	Action						Goto		
	id	+	*	()	\$	E	T	F
0	s5			s4			1	2	3
1		s6				acc			
2			s7						

The other item, $E \rightarrow T\bullet$, is a complete item and calls for a reduce by the corresponding rule, which is rule 2. SLR(1) uses the FOLLOW() to enter the reduce actions. FOLLOW(E) is the set of 3 terminals $\{+,), \$\}$. Hence we place reduce by rule 2 in the Action part of the table for these 3 terminals. Since state 2 has no items with a nonterminal following the \bullet , the Goto part has no entries for this state. The table at this stage of construction has the following entries.

We can check that the table given earlier has exactly the same entries for the first 3 rows as shown below.

State	Action						Goto		
	id	+	*	()	\$	E	T	F
0	s5			s4			1	2	3
1		s6				acc			
2		r2	s7		r2	r2			

SLR(1) Grammar and Parser

A grammar for which there is a conflict free SLR(1) parsing table is called a SLR(1) grammar and a parser which uses such a table is known as SLR(1) parser.

Q. How to detect shift-reduce conflicts and reduce-reduce conflicts in SLR(1) parser ?

- Step 5 of Algorithm 3.3 contains all the information. A shift- reduce conflict is detected when a state has

(a) a complete item of the form $A \rightarrow \alpha \bullet$ with $a \in \text{FOLLOW}(A)$, and also

(b) an incomplete item of the form $B \rightarrow \beta \bullet a \gamma$

- A reduce-reduce conflict is noticed when a state has two or more complete items of the form

(a) $A \rightarrow \alpha \bullet$

(b) $B \rightarrow \beta \bullet$, and

(c) $\text{FOLLOW}(A) \cap \text{FOLLOW}(B) \neq \Phi$

Having seen how to construct SLR(1) parsing table manually, we need to address the conceptual aspects of this method in order to answer questions of the following kind.

Exercise : Draw the SLR(1) automaton given in the Canonical Collection of LR(0) items given earlier in the figure, and also in the SLR(1) parsing table in the form of a directed graph. Then answer the following questions.

Q. What are the state symbols used by the parser and what information do they contain ?

Q. What is the significance of a path from start state to a given state ?

Q. How to produce a rightmost sentential while the parsing is in progress ?

End of Document