

Shift-Reduce Parsing:

Shift-reduce parsing attempts to construct a parse tree for an input string beginning at the leaves (the bottom) and working up towards the root (the top). We can think of this process as one of "**reducing**" a string w to the start symbol of a grammar. At each **reduction** step a particular substring matching the right side of a production is replaced by the symbol on the left of that production, and if the substring is chosen correctly at each step, a rightmost derivation is traced out in reverse.

Example: Consider the grammar:

$S \longrightarrow aABe$

$A \longrightarrow Abc / b$

$B \longrightarrow d$

The sentence *abbcd*e can be reduced to S by the following steps:

<i>abbcd</i> e
<i>aAbcd</i> e
<i>aAde</i>

We scan *abbcd*e looking for a substring that matches the right side of some production. These reductions, in fact, trace out the following **rightmost** derivation in reverse:

$$abbcd\epsilon \xrightarrow{rm} aAbcd\epsilon \xrightarrow{rm} aAde \xrightarrow{rm} aABe \xrightarrow{rm} S$$

Handles

Informally, a "**handle**" of a string is a substring that matches the right side of a production, and whose reduction to the nonterminal on the left side of the production represents one step along the reverse of a rightmost derivation. In many cases the leftmost substring β that matches the right side of some production $A \rightarrow \beta$ is not a **handle**, because a reduction by the production $A \rightarrow \beta$ yields a string that cannot be reduced to the **start symbol**.

In previous example, if we replaced **b** by **A** in the second string $aAbcd\epsilon$ we would obtain the string $aAAcd\epsilon$ that cannot be subsequently reduced to S . For this reason, we must give a more precise definition of a handle.

Formally, a **handle** of a right-sentential form γ is a production $A \rightarrow \beta$ and a position of γ where the string β may be found and replaced by A to produce the previous right-sentential form in a rightmost derivation of γ .

Stack Implementation of Shift-Reduce Parsing

There are two problems that must be solved if we are to parse by handle. The first is to locate the substring to be reduced in a right-sentential form, and the second is to determine what production to choose in case there is more than one production with that substring on the right side. Before we get to these questions, let us first consider the type of data structures to use in a shift-reduce parser.

A convenient way to implement a **Shift-Reduce Parsing** is to use **stack** to hold grammar symbols and an **input buffer** to hold the string (**W**) to be parsed. Use \$ to mark the bottom of the stack and also the right end of the input.

Initially, the stack is empty, and the string (**W**) is on the input as follows:

Stack

\$

Input

W\$

The parser operates by shifting zero or more input symbols onto the stack until a handle β is on top of the stack. The parser then reduces β to left side of the appropriate production. The parser repeat this cycle until it has detected an error or until the stack contains the start symbol S and the input is empty:

Stack

\$S

Input

\$

After entering this configuration, the parser halts and announces successful completion of parsing.

Note: the operations of the parser (**Shift**, **Reduce**, **Accept**, and **Error "not accept"**).

Example: Parse the input **id1+id2*id3** for this grammar:

$E \longrightarrow E+E / E * E / (E) / id$

Stack	Input	Action
\$	id1+id2*id3\$	Shift
\$id1	+id2*id3\$	Reduce $E \longrightarrow id$
\$E	+id2*id3\$	Shift
\$E+	id2*id3\$	Shift
\$E+id2	*id3\$	Reduce $E \longrightarrow id$
\$E+E	*id3\$	Shift
\$E+E*	id3\$	Shift
\$E+E*id3	\$	Reduce $E \longrightarrow id$
\$E+E*E	\$	Reduce $E \longrightarrow E*E$
\$E+E	\$	Reduce $E \longrightarrow E+E$
\$E	\$	Accept

Note: There is another sequence of steps a shift-reduce parser could take because the grammar is **ambiguous**.

Example: parse the input **id +*id** for same grammar above:

Stack	Input	Action
\$	id +*id	Shift
\$id	+ *id\$	Reduce $E \longrightarrow id$
\$E+	*id\$	Shift
\$E+*	id\$	Shift
\$E+*id	\$	Shift
\$E+*E	\$	Reduce $E \longrightarrow id$
\$E+*E	\$	Error

Operator-Precedence Parsing (OPP)

The largest class of grammars for which shift-reduce parsers can be built successfully. However, for a small but important class of grammars we can easily construct efficient shift-reduce parsers by hand. These grammars have the **property** (among other essential requirements) that **no production right side is ϵ or has two adjacent nonterminals**. A grammar with the latter property is called an **operator grammar**.

Example: The following grammar for expressions

$$E \longrightarrow EAE / (E) / -E / id$$

$$A \longrightarrow + / - / * / \div / \uparrow$$

Is not an **operator grammar**, because the right side **EAE** has two (in fact three) consecutive nonterminals. However, if we substitute for **A** each of its alternatives, we obtain the following operator grammar:

$$E \longrightarrow E+E / E-E / E * E / E \div E / E \uparrow E / (E) / -E / id$$

We now describe an easy-to-implement parsing technique called operator-precedence parsing.

In operator-precedence parsing, we define three disjoint **precedence relations**, **$<\bullet$** , **$=$** , and **$\bullet>$** , between certain pairs of **terminals**. These precedence relations guide the selection of **handles** and have the following meanings:

RELATION	MEANING
$a <\bullet b$	a "yields precedence to" b
$a = b$	a "has the same precedence as" b
$a \bullet > b$	a "takes precedence over" b

Example: The following grammar for expressions

$E \longrightarrow E + E / E * E / id$

	id	+	*	\$
Id		$\bullet >$	$\bullet >$	$\bullet >$
+	$< \bullet$	$\bullet >$	$< \bullet$	$\bullet >$
*	$< \bullet$	$\bullet >$	$\bullet >$	$\bullet >$
\$	$< \bullet$	$< \bullet$	$< \bullet$	

Operator-precedence relations

For example, suppose we initially have the right-sentential form **id + id ***
id and the precedence relations are shown in the table below.

Then the string with the precedence relations inserted is:

\$ <• id •> + <• id •> * <• id •> \$

\$ E + <• id •> * <• id •> \$

\$ <• E + <• id •> * <• id •> \$

\$ <• E + E * <• id •> \$

\$ <• E + <• E * <• id •> \$

\$ <• E + <• E * E \$

\$ <• E + <• E * E •> \$

\$ <• E + E \$

\$ <• E + E •> \$

\$ E\$ **Accept**

Table Construction of Operator-Precedence Relations

The table of Operator-precedence relations can be created according to the following steps:

- 1- Compute the **LEADING** and **TRAILING** for each nonterminal.
- 2- Determine the relation for each two terminal symbols *a* and *b*.

LEADING & TRAILING

LEADING(A) = {a | A $\xrightarrow{+}$ γaδ, where γ is ε or single nonterminal }

TRAILING(A) = { $a \mid A \xrightarrow{+} \gamma a \delta$, where δ is ϵ or single nonterminal }

Example: The following grammar for expressions

$$E \longrightarrow E+T \mid T$$

$$T \longrightarrow T * F \mid F$$

$$F \longrightarrow (E) \mid \text{id}$$

Nonterminals	LEADING	TRAILING
E	$+, *, (, \text{id}$	$+, *,), \text{id}$
T	$*, (, \text{id}$	$*,), \text{id}$
F	$(, \text{id}$	$), \text{id}$

Relations of Operator-Precedence Table

For each two terminal symbols **a** and **b**, we say:

- 1) **a = b** if there is a right side of a production of the form $\alpha a \beta b \gamma$, where β is either ϵ or a single nonterminal. That is **a = b** if **a** appears immediately to the left of **b** in a right side, or if they appear separated by one nonterminal. For example, the production $S \longrightarrow i C t S e S$ implies that **i = t** and **t = e**.
- 2) **a <• b** if for some nonterminal A there is a right side of the form $\alpha a A \beta$, then **a <• LEADING (A)**

For Example, $S \longrightarrow i C t S$, and $C \xrightarrow{+} b$, so **i <• b**. and **t <• i**

Also, the **\$ <• LEADING (S)**, where **S** is start Symbol.

3) $a \bullet > b$ if for some nonterminal A there is a right side of the form $\alpha A b \beta$, then $\text{TRAILING}(A) \bullet > b$

For Example, $S \longrightarrow iCtS$, and $C \xrightarrow{+} b$, so $b \bullet > t$.

Also, the $\text{TRAILING}(A) \bullet > \$$, where S is start Symbol.

Example: The following grammar for expressions

$$E \longrightarrow E+T \mid T$$

$$T \longrightarrow T*F \mid F$$

$$F \longrightarrow (E) \mid \text{id}$$

By applying the **Relations** in above will be resulted the following table:

	+	*	()	id	\$
+	$\bullet >$	$< \bullet$	$< \bullet$	$\bullet >$	$< \bullet$	$\bullet >$
*	$\bullet >$	$\bullet >$	$< \bullet$	$\bullet >$	$< \bullet$	$\bullet >$
($< \bullet$	$< \bullet$	$< \bullet$	=	$< \bullet$	
)	$\bullet >$	$\bullet >$		$\bullet >$		$\bullet >$
id	$\bullet >$	$\bullet >$		$\bullet >$		$\bullet >$
\$	$< \bullet$	$< \bullet$	$< \bullet$		$< \bullet$	

1- $a = b$, $Aa\beta b\gamma (E) \Rightarrow (=)$

2- $a < \bullet b$, $\alpha a A \beta$

$$E+T \Rightarrow + < \bullet \text{LEADING}(T) \quad + < \bullet \{*, (, \text{id}\}$$

$$T*F \Rightarrow * < \bullet \text{LEADING}(F) \quad * < \bullet \{ (, \text{id}\}$$

$(E) \Rightarrow (\prec \bullet \text{LEADING}(E) \quad (\prec \bullet \{+, *, (, \text{id}\}$

$\$ \prec \bullet \text{LEADING}(E) \quad \$ \prec \bullet \{+, *, (, \text{id}\}$

3- $a \bullet > b, \alpha Ab \beta \quad E+T \Rightarrow \text{TRAILING}(E) \bullet > + \quad \{+, *,), \text{id}\} \bullet > +$

$T * F \Rightarrow \text{TRAILING}(T) \bullet > * \quad \{*,), \text{id}\} \bullet > *$

$(E) \Rightarrow \text{TRAILING}(E) \bullet >) \quad \{+, *,), \text{id}\} \bullet >)$

$\text{TRAILING}(E) \bullet > \$ \quad \{+, *,), \text{id}\} \bullet > \$$

LR parser

This section presents an efficient bottom-up syntax analysis technique that can be used to parse a large class of context-free grammars. This technique is called LR parsing; the L is for left-right scanning of the input, the R for constructing a rightmost derivation in reverse. This method presents three techniques for constructing an LR parsing table for grammar. The first method, called simple LR (SLR), is easiest to implement. But the least powerful. The second method, called canonical LR, is the most powerful and will work on a very large class of grammars and the most expensive. The third method, called look ahead LR (LALR), is intermediate in power and cost between the SLR and the canonical LR methods.

1.SLR Parser

This method of parsing is the weakest of three in terms of the number of grammars for which it succeeds, but it is easiest to implement. This parsing method has four basic steps:

1. Find first & follow.
2. Find set of I.
3. Find parsing table.
4. Check the sentence (parse the input).

The CLOSURE operation

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

1. Every item in I is in $CLOSURE(I)$.
2. If $A \alpha.B\beta$ is in $CLOSURE(I)$ and $B \rightarrow \gamma$ is a production, then add the item $B \rightarrow \gamma$ to I , if it is not already there.

Example: consider the grammar:

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

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

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

The function closure can be computed bellow:

function $CLOSURE(I)$;

begin

$J = I$;

repeat

for each item $A \rightarrow \alpha \cdot B \beta$ in I and each production $B \rightarrow \gamma$ in G such

that $B \rightarrow \cdot \gamma$ is not in I do

add $B \rightarrow \cdot \gamma$ to J .

until no more items can be added to J ;

return J

end.

GO TO operation

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 $A \longrightarrow \alpha X. \beta$ such that $A \longrightarrow \alpha X. \beta$ is in I.

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

$E \rightarrow E + \cdot T$

$T \rightarrow \cdot T * F$

$T \rightarrow \cdot F$

$F \rightarrow \cdot (E)$

$F \rightarrow \cdot id$

The Set of items Constructions:

procedure ITEMS(G');

begin

$c = \{\text{closure}(\{\dot{S} \longrightarrow .S\})\};$

repeat

for each set of items I in c and each grammar symbol x such that GOTO(I, x) is not empty and is not in c do add GOTO(I, x) to C

until no more sets of items can be added to C

end.

Example: consider the grammar:

$E \longrightarrow E + T \mid T$

$T \longrightarrow T * F \mid F$

$F \longrightarrow (E) \mid id$

1. Find first and follow

First

Follow

E (, id

\$,) , +

T (, id

\$,) , + , *

F (, id

\$,) , + , *

2. Find set of I.

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

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

$I_2:$ $E \rightarrow T \cdot$
 $T \rightarrow T \cdot * F$

$I_3:$ $T \rightarrow F \cdot$

$I_4:$ $F \rightarrow (\cdot E)$
 $E \rightarrow \cdot E + T$
 $E \rightarrow \cdot T$
 $T \rightarrow \cdot T * F$
 $T \rightarrow \cdot F$
 $F \rightarrow \cdot (E)$
 $F \rightarrow \cdot id$

$I_5:$ $F \rightarrow id \cdot$

$I_6:$ $E \rightarrow E + \cdot T$
 $T \rightarrow \cdot T * F$
 $T \rightarrow \cdot F$
 $F \rightarrow \cdot (E)$
 $F \rightarrow \cdot id$

$I_7:$ $T \rightarrow T * \cdot F$
 $F \rightarrow \cdot (E)$
 $F \rightarrow \cdot id$

$I_8:$ $F \rightarrow (E \cdot)$
 $E \rightarrow E \cdot + T$

$I_9:$ $E \rightarrow E + T \cdot$
 $T \rightarrow T \cdot * F$

$I_{10}: T \rightarrow T * F \cdot$

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

3. Find parsing table.

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			

4. Check the sentence (parse the input).

	Stack	Input
(1)	0	id * id + id \$
(2)	0 id 5	* id + id \$
(3)	0 F 3	* id + id \$
(4)	0 T 2	* id + id \$
(5)	0 T 2 * 7	id + id \$
(6)	0 T 2 * 7 id 5	+ id \$
(7)	0 T 2 * 7 F 10	+ id \$
(8)	0 T 2	+ id \$
(9)	0 E 1	+ id \$
(10)	0 E 1 + 6	id \$
(11)	0 E 1 + 6 id 5	\$
(12)	0 E 1 + 6 F 3	\$
(13)	0 E 1 + 6 T 9	\$
(14)	0 E 1	\$

SLR Parsing tables

INPUT: An augmented grammar \hat{G}

OUTPUT: The SLR parsing table functions action and GOTO for \hat{G}

Method. Let $C = \{I_0, I_1, \dots, I_n\}$. The states of the parser are $0, 1, \dots, n$, state i being constructed from I_i . The parsing actions for state i are determined as follows:

1. If $[A \rightarrow \alpha \cdot a \beta]$ is in I_i and $\text{GOTO}(I_i, a) = I_j$, then set $\text{ACTION}[i, a]$ to “shift j .” Here a is a terminal.
2. If $[A \rightarrow \alpha \cdot]$ is in I_i , then set $\text{ACTION}[i, a]$ to “reduce $A \rightarrow \alpha$ ” for all a in $\text{FOLLOW}(A)$.
3. If $[S' \rightarrow S \cdot]$ is in I_i , then set $\text{ACTION}[i, \$]$ to “accept.”

The goto transitions for state i are constructed using the rule:

4. If $\text{GOTO}(I_i, A) = I_j$, then $\text{GOTO}[i, A] = j$.
5. All entries not defined by rules (1) through (4) are made “error.”
6. The initial state of the parser is the one constructed from the set of items containing $[S' \rightarrow \cdot S]$. \square

Canonical LR Parser

It is possible to carry more information in the state by re- defining items, to include a terminal symbol as a second component. The general form of an item becomes $\{A \alpha \cdot B\beta, X\}$, where $A \alpha \cdot B$ is a production and X is a terminal or the right end marker $\$$, we call such an object an LR(1) item.

parsing method there are four basic steps:

- 1. Find first function**
- 2. Find set of I.**
- 3. Find canonical parsing table.**
- 4. Check the sentence (parse the input).**

Algorithm :Construction of the sets of LR(1) items for a grammar

Input : a grammar G

Output: The sets of LR(1) items

Method: Using the procedure Closure and GO TO and the main routine for constructing the sets of items.

```

procedure CLOSURE( $I$ );
begin
    repeat
        for each item  $[A \rightarrow \alpha \cdot B \beta, a]$  in  $I$ , each
            production  $B \rightarrow \gamma$ , and each terminal  $b$  in  $\text{FIRST}(\beta a)$ 
            such that  $[B \rightarrow \cdot \gamma, b]$  is not in  $I$  do
                add  $[B \rightarrow \cdot \gamma, b]$  to  $I$ ;
    until no more items can be added to  $I$ ;
    return  $I$ ;
end;

procedure GOTO( $I, X$ );
begin
    let  $J$  be the set of items  $[A \rightarrow \alpha X \cdot \beta, a]$ , such that
         $[A \rightarrow \alpha \cdot X \beta, a]$  is in  $I$ ;
    return CLOSURE( $J$ )
end;

begin
     $C := \{\text{CLOSURE}(\{S' \rightarrow \cdot S, \$\})\}$ ;
    repeat
        for each set of items  $I$  in  $C$  and each grammar
            symbol  $X$  such that GOTO( $I, X$ ) is not empty
            and not already in  $C$  do
                add GOTO( $I, X$ ) to  $C$ 
    until no more sets of items can be added to  $C$ 
end

```

Algorithm 6.3. Construction of a canonical LR parsing table.

Input. A grammar G augmented by production $S' \rightarrow S$.

Output. If possible, the canonical LR parsing action function ACTION and goto function GOTO.

Method.

1. Construct $C = \{I_0, I_1, \dots, I_n\}$, the collection of sets of LR(1) items for G .
2. State i of the parser is constructed from I_i . The parsing actions for state i are determined as follows:
 - a) If $[A \rightarrow \alpha \cdot a \beta, b]$ is in I_i and $\text{GOTO}(I_i, a) = I_j$, then set $\text{ACTION}[i, a]$ to "shift j ."
 - b) If $[A \rightarrow \alpha \cdot, a]$ is in I_i , then set $\text{ACTION}[i, a]$ to "reduce $A \rightarrow \alpha$."
 - c) If $[S' \rightarrow S \cdot, \$]$ is in I_i , then set $\text{ACTION}[i, \$]$ to "accept."

If a conflict results from the above rules, the grammar is said not to be LR(1), and the algorithm is said to fail.

3. The goto transitions for state i are determined as follows: If $\text{GOTO}(I_i, A) = I_j$, then $\text{GOTO}[i, A] = j$.
4. All entries not defined by rules (2) through (3) are made "error."
5. The initial state of the parser is the one constructed from the set containing item $[S' \rightarrow \cdot S, \$]$. \square

An LR parser using a previous table is called the canonical LR parser.

Notes:

1. $A \longrightarrow \alpha. B\beta, X$

α, β : anything

X : terminal or sign dolar

B : nonterminal

2. $\text{first}(\beta X)$

$S \sim \longrightarrow .S. \$$

$\alpha = \lambda, B = S, \beta = \lambda, X = \$$

$\text{first}(\beta X) = \text{first}(\$) = \$$

Example: consider the following grammar

$S \longrightarrow CC$

$C \longrightarrow cC \mid d$

1. find the first function

first

S c, d

C c, d

2. find set of I

$I_0: S' \rightarrow \cdot S, \$$
 $S \rightarrow \cdot CC, \$$
 $C \rightarrow \cdot cC, c/d$
 $C \rightarrow \cdot d, c/d$

$I_1: S' \rightarrow S \cdot, \$$

$I_2: S \rightarrow C \cdot C, \$$
 $C \rightarrow \cdot cC, \$$
 $C \rightarrow \cdot d, \$$

$I_3: C \rightarrow c \cdot C, c/d$
 $C \rightarrow \cdot cC, c/d$
 $C \rightarrow \cdot d, c/d$

$I_4: C \rightarrow d \cdot, c/d$

$I_5: S \rightarrow CC \cdot, \$$

$I_6: C \rightarrow c \cdot C, \$$
 $C \rightarrow \cdot cC, \$$
 $C \rightarrow \cdot d, \$$

$I_7: C \rightarrow d \cdot, \$$

$I_8: C \rightarrow cC\cdot, c/d$

$I_9: C \rightarrow cC\cdot, \$$

3. Find canonical LR parsing table

State	Action			Goto	
	c	d	\$	S	C
0	s3	s4		1	2
1			acc		
2	s6	s7			5
3	s3	s4			8
4	r3	r3			
5			r1		
6	s6	s7			9
7			r3		
8	r2	r2			
9			r2		

4. Check the input string cdc

Stack

Input

0	cdcd\$
0c3	dcd\$
0c3d4	cd\$
0c3C8	cd\$
0C2	cd\$
0C2c6	d\$
0C2c6d7	\$
0C2c6C9	\$
0C2C5	\$
0S1	\$

Accept

LALR Parser

We now introduce LALR { Look ahead LR } technique. For a comparison of parsing size, the SLR and LALR tables for a grammar always have the same number of states, and a smaller than canonical LR table. Thus, it is much easier and more economical to construct SLR or LALR tables than the canonical LR tables.

Algorithm 6.4. An easy, but space consuming LALR table construction.

Input. A grammar G augmented by production $S' \rightarrow S$.

Output. The LALR parsing tables ACTION and GOTO.

Method.

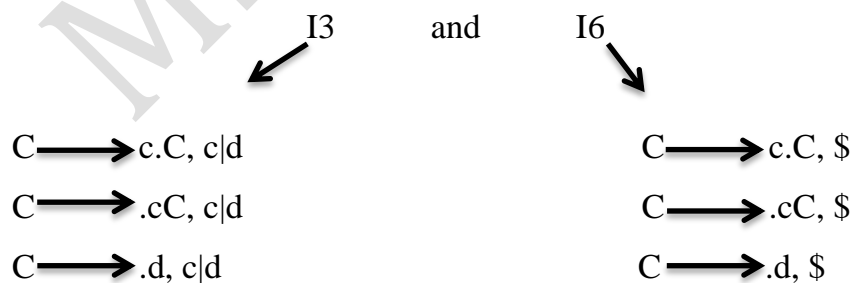
1. Construct $C = \{I_0, I_1, \dots, I_n\}$, the collection of sets of LR(1) items.
2. For each core present among the sets of LR(1) items, find all sets having that core, and replace these sets by their union.
3. Let $C' = \{J_0, J_1, \dots, J_m\}$ be the resulting sets of LR(1) items. The parsing actions for state i are constructed from J_i in the same manner as in Algorithm 6.3. If there is a parsing-action conflict, the algorithm fails to produce a parser, and the grammar is said not to be LALR(1).
4. The GOTO table is constructed as follows. If J is the union of one or more sets of LR(1) items, i.e., $J = I_1 \cup I_2 \cup \dots \cup I_m$, then the cores of $\text{GOTO}(I_1, X)$, $\text{GOTO}(I_2, X), \dots, \text{GOTO}(I_k, X)$ are the same, since I_1, I_2, \dots, I_k all have the same core.

Example: Consider the grammar

$S \longrightarrow CC$

$C \longrightarrow cC \mid d$

As we mentioned, there are three pairs of sets of items that can be merged.



These two sets are replaced by their union:

I36: $C \longrightarrow c.C, c|d \mid \$$

$C \longrightarrow .cC, c|d \mid \$$

$C \longrightarrow .d, c|d \mid \$$

Also

I4
 \swarrow
 $C \longrightarrow d . , c|d$

and

I7
 \swarrow
 $C \longrightarrow d . , \$$

Their union :

I47: $C \longrightarrow d . , c|d \mid \$$

And also

I8
 \swarrow
 $C \longrightarrow cC . , c|d$

and

I9
 \swarrow
 $C \longrightarrow cC . , \$$

Their union:

I89: $C \longrightarrow cC . , c|d \mid \$$

The LALR parsing table is:

State	Actions			Go to	
	c	d	\$	S	C
0	s36	s47		1	2
1			acc		
2	s36	s47			5
3 6	s36	s47			8 9
4 7	r3	r3	r3		
5			r1		
8 9	r2	r2	r2		

Conflict in shift-Reduce parsing

"Conflicts" occur when an ambiguity in the grammar creates a situation where the parser does not know which step to perform at a given point during parsing.

There are two kinds of conflicts that occur

1. shift-reduce: a shift reduce conflict occurs when the grammar indicates that different successful parses might occur with either a shift or a reduce at a given point during parsing. The vast majority of situations where this conflict occurs can be correctly resolved by shifting.

2- reduce-reduce : a reduce-reduce conflict occurs when the parser has two or more handles at the same time on the top of the stack. Whatever choice the parser makes is just as likely to be wrong as not. In this case it is usually best to rewrite the grammar to eliminate the conflict, possibly by factoring.

Example1: shift reduce conflict:

$S \rightarrow \text{if } E \text{ then } S$

$S \rightarrow \text{if } E \text{ then } S \text{ else } S$

In many languages two nested "if" statements produce a situation where an "else" clause could legally belong to either "if". The usual rule (to shift) attaches the else to the nearest (i.e. inner) if statement.

Example 2: reduce reduce conflict:

- (1) $\text{statement} \rightarrow \text{id}(\text{parameter-list})$
- (2) $\text{statement} \rightarrow \text{expression} := \text{expression}$
- (3) $\text{parameter-list} \rightarrow \text{parameter-list}, \text{parameter}$
- (4) $\text{parameter-list} \rightarrow \text{parameter}$
- (5) $\text{parameter} \rightarrow \text{id}$
- (6) $\text{expression} \rightarrow \text{id}(\text{expression-list})$
- (7) $\text{expression} \rightarrow \text{id}$
- (8) $\text{expression-list} \rightarrow \text{expression-list}, \text{expression}$
- (9) $\text{expression-list} \rightarrow \text{expression}$

A statement beginning with $A(i,*)$ would appear as the token stream $\text{id}(\text{id}, \text{id})$ to the parser. After shifting the first three tokens onto the stack, a shift-reduce parser would be in configuration

Stack	input
....id(id	,id)\$

The id on top of the stack must be reduced by :

- Production (5) if A is a procedure
- Production (7) if A is an array

Example 2: the following grammar

- (1) $E \rightarrow E + E$
- (2) $E \rightarrow E * E$
- (3) $E \rightarrow (E)$
- (4) $E \rightarrow \text{id}$

Is ambiguous because it does not specify the precedence of the operators + and *.
The sets of LR(0) items augmented by $E' \rightarrow E$:

$I_0:$ $E' \rightarrow \cdot E$
 $E \rightarrow \cdot E + E$
 $E \rightarrow \cdot E * E$
 $E \rightarrow \cdot (E)$
 $E \rightarrow \cdot \text{id}$

$I_3:$ $E \rightarrow E * \cdot E$
 $E \rightarrow \cdot E + E$
 $E \rightarrow \cdot E * E$
 $F \rightarrow \cdot (E)$
 $E \rightarrow \cdot \text{id}$

$I_1:$ $E' \rightarrow E \cdot$
 $E \rightarrow E \cdot + E$
 $E \rightarrow E \cdot * E$

$I_6:$ $E \rightarrow (E \cdot)$
 $E \rightarrow E \cdot + E$
 $E \rightarrow E \cdot * E$

$I_2:$ $E \rightarrow (\cdot E)$
 $E \rightarrow \cdot E + E$
 $E \rightarrow \cdot E * E$
 $E \rightarrow \cdot (E)$
 $E \rightarrow \cdot \text{id}$

$I_7:$ $E \rightarrow E + E \cdot$
 $E \rightarrow E \cdot + E$
 $E \rightarrow E \cdot * E$

$I_3:$ $E \rightarrow \text{id} \cdot$

$I_8:$ $E \rightarrow E * E \cdot$
 $E \rightarrow E \cdot + E$
 $E \rightarrow E \cdot * E$

$I_4:$ $E \rightarrow E + \cdot E$
 $E \rightarrow \cdot E + E$
 $E \rightarrow \cdot E * E$
 $E \rightarrow \cdot (E)$
 $E \rightarrow \cdot \text{id}$

$I_9:$ $E \rightarrow (E) \cdot$

STATE	action						goto
	M	+	*	()	\$	E
0	s3			s2			1
1		s4	s5			acc	
2	s3			s2			6
3		r4	r4		r4	r4	
4	s3			s2			8
5	s3			s2			8
6		s4	s5		s9		
7		r1	s5		r1	r1	
8		r2	r2		r2	r2	
9		r3	r3		r3	r3	

The relative precedence of + followed by * uniquely determines how the parsing action conflict between reducing $E \rightarrow E+E$ and shifting on * in state 7 should be resolved.

Error recovery in LR parsing

An LR parser will announce error as soon as there is no valid continuation for the portion of the input thus far scanned. Thus we may fill in each blank entry in the action field with a pointer to an error routine that will take an appropriate action selected by the compiler designer. The action may include insertion or deletion of symbols from the stack or input or both, or alteration and transposition of input symbols.

STATE	action						goto
	id	+	*	()	\$	E
0	s3	e1	e1	s2	e2	e1	1
1	e3	s4	s5	e3	e2	acc	
2	s3	e1	e1	s2	e2	e1	6
3	r4	r4	r4	r4	r4	r4	
4	s3	e1	e1	s2	e2	e1	7
5	s3	e1	e1	s2	e2	e1	8
6	e3	s4	s5	e3	s9	e4	
7	r1	r1	s5	r1	r1	r1	
8	r2	r2	r2	r2	r2	r2	
9	r3	r3	r3	r3	r3	r3	

- e1:** /* This routine is called from states 0, 2, 4 and 5, all of which expect the beginning of an operand, either an id or a left parenthesis. Instead, an operator, + or *, or the end of the input was found. */
 push an imaginary id onto the stack and cover it with state 3 (the goto of states 0, 2, 4 and 5 on id)⁵
 issue diagnostic "missing operand"
- e2:** /* This routine is called from states 0, 1, 2, 4 and 5 on finding a right parenthesis. */
 remove the right parenthesis from the input
 issue diagnostic "unbalanced right parenthesis"
- e3:** /* This routine is called from states 1 or 6 when expecting an operator, and an id or right parenthesis is found. */
 push + onto the stack and cover it with state 4.
 issue diagnostic "missing operator"
- e4:** /* This routine is called from state 6 when the end of the input is found.

State 6 expects an operator or a right parenthesis. */
push a right parenthesis onto the stack and cover it with state 9.
issue diagnostic "missing right parenthesis"

Semantic Analysis

The role of semantic analyzer

The role of semantic analyzer is to derive methods by which the structures constructed by the syntax analyzer may be evaluated or analyzed.

The semantic analysis phase checks the source program for semantic errors and gathers data type information for the subsequent code-generation phase. An important component of semantic analysis is **type checking**. Here the compiler checks that each operator has operands that are permitted by the source language specification. For example: Many programming languages definition require a compiler to report an error every time a **real number is used to index an array**.

Semantic Errors:

Semantic errors include type mismatches between operators and operands. For examples:

1. int x; x="book";
2. int a[10]; a[15]=2;
3. for (i=1; i>=10; i++)
4. a[1.2]=44;
5. and more.....

Intermediate code generation:

Generate an explicit intermediate representation of the source program. This representation should have two important properties, it should be easy to produce and easy to translate into the target program.

Some of the basic operations which in the so program, to change in the assembly language:

Operations	H.L.L	Assembly language
Math. operation	+, -, *, /	Add, sub, mult, div
Boolean operation	&, , ~	And, or, not
Assignment	:=	Mov, LD, Store
Jump	Go to	JP, JN, JC
Conditional	If, Case	CMP
Loop instruction	For, Do, Repeat, While	These must have I.C

The operation which change H.L.L to Assembly language, is called the Intermediate code generation and there is the division operation come it, which mean every statement have a sing operation.

Example:

$$X=A+B*C/D-Y*N$$

$$T1= B*C$$

$$T2=T1/D$$

$T3 = Y * N$

$T4 = A + T2$

$T5 = T4 - T3$

Example: $Y = \text{Cos}(A * B) + C / N - X * P$

$T1 = A * B$

$T2 = \text{Cos}(T1)$

$T3 = X * p$

$T4 = C / N$

$T5 = T2 + T4$

$T6 = T5 - T3$

If Condition Statement:

Example:

$X = 1;$

If ($X > Y$)

{ $A = A + 1;$

$B = B - A + 2;$

}

$P = P + 1;$

10 $X = 1$

20 If $X \leq Y$ go to 60

30 $A = A + 1$

40 $T1 = B - A$

50 $B = T1 + 2$

Example:

X=1

If ((X>Y) && (Y>=2))

{

A=A+1

B=B-A+2

}

Else X=X+1;

P=P+2+X;

10 X=1

20 If X>Y go to 50

30 X= X+1

40 go to 100

50 If Y>=2 go to 70

60 go to 30

70 A=A+1

80 T1=B-A

90 B=T1+2

100 T2=P+2

110 P=T2+X

For - Loop

Example:

For (i=1; i<=10;i++)

X = X+ (i*Y);

10 i= 1

20 If i> 10 go to 70

30 T1= i* Y

40 X= X+T1

50 i= i+1

60 go to 20

70 end

Example:

For (i=10; i>=0;i--)

$Y = Y + X * Z$

```
10 i= 10
20 If i<0 go to 70
30 T1= X* Z
40 Y= Y+T1
50 i= i-1
60 go to 20
70 end
```