UNIT 3: SYNTAX ANALYSIS-2

[]Parser:

A parser is program for Grammar G that takes a string ‘w’ as input and generates the Parse tree if the string ‘w’ is valid otherwise it generates error message indicating ‘w’ is invalid.

[]Bottom-up Parser :

A bottom-up parser 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).

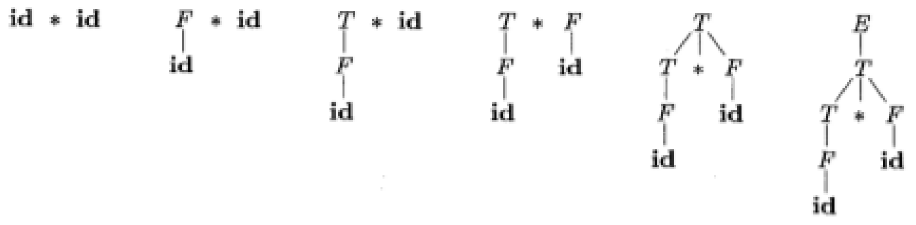


Figure 4.25: A bottom-up parse for **id** * **id**

[]Reductions:

Bottom-up parsing is 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.

Fig. above illustrates a sequence of reductions; the grammar is the expression grammar (4.1). The reductions will be discussed in terms of the sequence of strings id * id, F * id, T * id, T * F, T, E

The sequence starts with the input string id*id. The first reduction produces F * id by reducing the leftmost id to F, using the production F -+ id. The second reduction produces T * id by reducing F to T. Now, we have a choice between reducing the string T, which is the body of E -+ T, and the string consisting of the second id, which is the body of F -+ id. Rather than reduce T to E, the second id is reduced to T, resulting in the string T * F. This string then reduces to T. The parse completes with the reduction of T to the start symbol E.

The goal of bottom-up parsing is therefore to construct a derivation in reverse. The following derivation corresponds to the parse in Fig. 4.25:

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

This derivation is in fact a rightmost derivation.

[]Handle:

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.

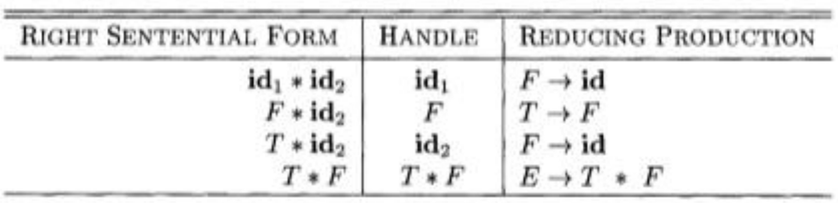
Informally, A handle is a substring that matches with the right hand side of some production, replacing its left hand side of the symbol that produces the previous right sentential form in the reverse process of right most derivation.

i.e., A handle of a right sentential form γ is a production A->β and a position of γ where the string β may be found and replaced by A to produce previous right sentential form in a right most derivation of a γ.

[]Handle pruning :

A handle pruning is a mechanism to obtain a rightmost derivation in reverse in the working of shift reduce parser

Here we start with of terminals ‘w’ to be parsed , if ‘w’ is the sentence of the grammar at hand, then let w=γn where γn is the nth right sentential form of some unknown rightmost derivation as mentioned below



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

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

An example of Bottom up parser is shift reduce parser and this can be implemented on large class of grammars and these grammars are called LR grammar. The Shift reduce Parser for LR grammar is called LR parser

Shift-reduce parsing is a form of bottom-up parsing in which a stack holds grammar symbols and an input buffer holds the rest of the string to be parsed. The handle always appears at the top of the stack just before it is identified as the handle. We use $ to mark the bottom of the stack and also the right end of the input

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 P of grammar symbols on top of the stack. It then reduces ,O 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:

[]Implementation of Shift Reduce parser:

1. Initialize the stack to empty and input buffer holds the string ‘w’ to be parsed. Also input ointer is pointing to the first character of W.

Stack	Input
$	W$

During the left to right scan of the input, parser shifts zero or more input symbols onto the stack until ***Handle* β is** onto the Stack.

2. Perform reduction action using the production A->β. i.e β is popped from the stack nd A is pushed.

3. Repeat the steps 2 and 3 until error is detected or until the stack contains the start symbol and the input is empty.
4. i. e.

Stack	Input
$S	$

5. Upon entering the above configuration Parser halts and announces successful completion of parsing.

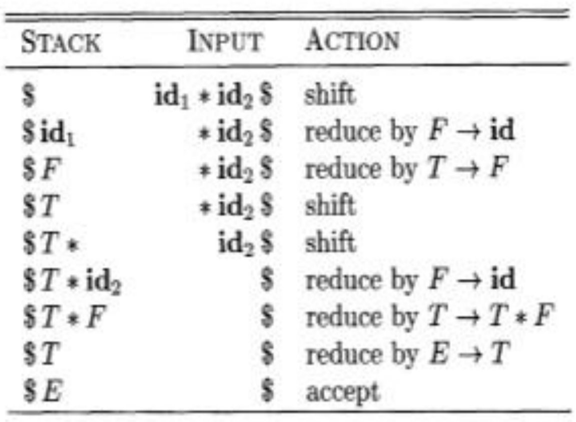
[]Trace of Shift Reduce Parser :

Ex. 1) reducing id+id*id

//to distinguish identifiers we have used id1, id2 and id3.

Stack	Input	Action
$	Id1+id2*id3 $	Shift id1
$id1	+id2*id3 $	Reduce by E->id
$E	+id2*id3 $	Shift +
$E+	id2*id3 $	Shift id2
$E + id2	*id3 $	Reduce by E->id
$E + E	*id3$	Shift *
$E + E *	id3$	Shift id3
$E + E * id3	$	Reduce by E->id
$E + E * E	$	Reduce by E->E * E
$E + E	$	Reduce by E-> E + E
$E	$	Accept

Ex. 2) reduce id*id



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.

[]Conflicts During Shift-Reduce Parsing :

There are context-free grammars for which shift-reduce parsing cannot be used. Every shift-reduce parser for such a grammar can reach a configuration in which the parser, knowing the entire stack contents and the next input symbol, cannot decide whether to shift or to reduce (a shift/reduce conflict), or cannot decide which of several reductions to make (a reduce/reduce conflict).

These grammars are not in the LR(k) class of grammars . we refer to them as non-LR grammars. The k in LR(k) refers to the number of symbols of look ahead on the input. Grammars used in compiling usually fall in the LR(1) class, with one symbol of look ahead at most.

Example: An ambiguous grammar can never be LR. For example, consider the dangling-else grammar

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

If we have a shift-reduce parser in configuration

STACK	INPUT
··· **if** *expr* **then** *stmt*	**else** ··· $

we cannot tell whether if expr then stmt is the handle, no matter what appears below it on the stack. Here there is a shiftlreduce conflict. Depending on what follows the else on the input, it might be correct to reduce if expr then stint to stmt, or it might be correct to shift else and then to look for another stmt to complete the alternative if expr then stmt else stmt.

Note that shift-reduce parsing can be adapted to parse certain ambigu- ous grammars, such as the if-then-else grammar above. If we resolve the shiftlreduce conflict on else in favor of shifting, the parser will behave as we expect, associating each else with the previous unmatched then.

[]LR Parser :

- This is one of the best method for syntactic recognition of programming language constructs.

- It uses Shift-Reduce technique discussed earlier and hence LR(k) parser is an example of Shift Reduce Parser.
- Here L stands for Left-to- right scanning of the input , R stands for construction of right most derivation in reverse and k stands for number of look ahead input symbols used in making parsing decisions. The value of k is either 0 or 1. if (k) omitted then k is assumed to be 1

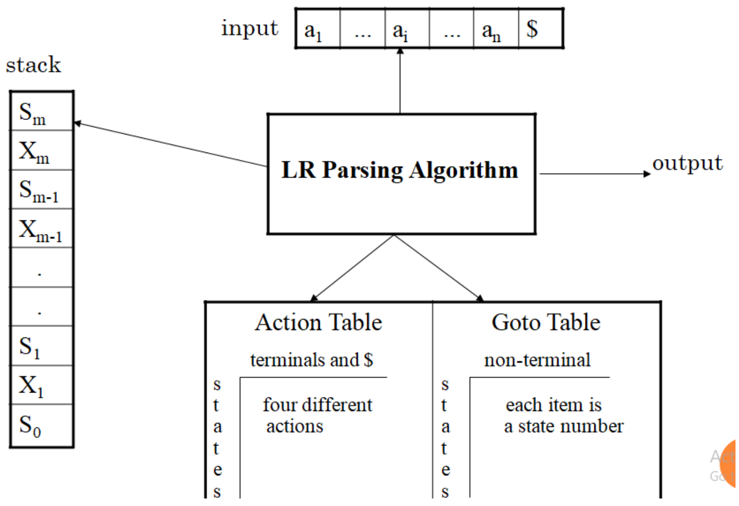
[]Attractive features of LR parser :

1. An LR parser can recognize virtually all programming language constructs written with context free grammars
2. It is most general non- backtracking technique known.
3. The class of grammars that can parsed using LR methods is proper superset of the class of grammars that can parsed with predictive parser.
4. It can detect syntax errors quickly.

[]Drawback of LR Parser :

It is too much work to construct an LR parser by band for typical programming language constructs. However tools exists to automatically generate LR parser from a given grammar.

[]Model of LR parser:



LR parser consists of five components namely,

an Input, an Output, a stack, a Parsing table, a driver/parsing Program.

1. Input :
   This holds the string to be processed and is divided into n number of cells, each capable of holding a single character. There is reading mechanism that reads the single character at a time from left to right.

2. Output :
   This executes the semantic actions associated with each productions whenever a particular production is used for reduction. Here we assume printing of production, reporting error messages and reporting acceptance of Input string.

3. Stack :
   It is used by the parsing program that holds the string of the form **s0X1s1X2s2………Xmsm** where **sm** is on top of stack.

   Here each **Xi** is grammar symbol and **si** is a state symbol which summarizes information contained in the stack and indicates where we are in a parse.

   In the implementation , the grammar symbol **Xi** need not appear on the stack. However for convenience we include them on the stack.

4. Parsing Table :

   It is a two dimensional array consists of two parts **Action** and **Goto**.

   The **action** part of the table consists of actions**(Shift, reduce, error and accept)** that the parser has to take. It is indexed by state **'Si'** on the top of stack and current input symbols '**ai**' i.e. **action[Si, ai].**

   The **Goto** part decides the next state whenever the parser performs reduce action. It is indexed by current state **'Si'** and head of the production used for reduction **A**. i. e **goto[Si, A].**

5. Driver/Parser program:

   It is software program that controls the entire parsing process. It reads the character from the input buffer one at a time. It determines the **'Sm'** the state currently on the top stack and **'ai'** the current input symbol. It then consults the parsing action table entry **action[Sm,ai],** which can have the following values:

   Shift **S** where **S** is state.

   Reduce by a production **A->β**

   Error

   Accept

   whenever reduce action is performed, it also consults the parsing **goto** table entry **goto[S, A]** to determine the next state

[]LR parsing Algorithm:

```
Set ip to point o the first symbol of w$
Repeat forever
Begin
    let 's' be the state on top of the stack and 'a'
    be the symbol pointed by the ip;
    if action[s,a] = shift s' then
    begin
        push 'a' and 's' on top of the stack;
        advance ip to the next input symbol
    end

    else if action[s, a] = reduce A→β then
      Begin
        pop 2*|β | symbols off the stack;
        let 's' 'be the state now on top of the stack
        push A then goto[s', A] on top of the stack;
        output the production A→β
      end
      else if action[s, a] = accept then
      begin
            print(" successful completion of parsing")
            return;
      end
      else error();
end
```

(SLR) Parsing Tables for Expression Grammar

1) E → E+T
2) E → T
3) T → T*F
4) T → F
5) F → (E)
6) F → id

	Action Table						Goto Table		
state	**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			

[]Actions of A (S)LR-Parser – Example:

stack	input	action	output
0	id*id+id$	shift 5	
0id5	*id+id$	reduce by F→id	F→id
0F3	*id+id$	reduce by T→F	T→F
0T2	*id+id$	shift 7	
0T2*7	id+id$	shift 5	
0T2*7id5	+id$	reduce by F→id	F→id
0T2*7F10	+id$	reduce by T→T*F	T→T*F
0T2	+id$	reduce by E→T	E→T
0E1	+id$	shift 6	
0E1+6	id$	shift 5	
0E1+6id5	$	reduce by F→id	F→id
0E1+6F3	$	reduce by T→F	T→F
0E1+6T9	$	reduce by E→E+T	E→E+T
0E1	$	accept	

[]Construction of Parsing table for LR Parser :

1. Collection of canonical sets of LR(0) items
   i. augment grammar:
      If **G** is a grammar with **start symbol S**, then **G'** is an augment grammar for **G** with a new start symbol **S'** and new production rule **S'→S**.
   ii. Collect LR(0) items:
      An LR(0) item of a grammar **G** is a production of **G** with a **dot** at the some position of the right side.
      Ex: A → aBb
      Possible LR(0) *Items*: A → .aBb
      (four different possibility) A → a.Bb
      A → aB.b
      A → aBb.
   iii. Take Closure:

[]Closure operation:

If ***I*** is a set of **LR(0) items** for a grammar G, then ***closure(I)*** is the set of LR(0) items constructed from ***I*** by the two rules:

Initially, every **LR(0) item** in ***I*** is added to **closure(I).**

If **A → α.Bβ** is in **closure(I)** and **B→γ** is a production rule of G; then **B→ •γ** will be in the **closure(I).** We will apply this rule until no more new **LR(0) items** can be added to **closure(I).**

[]Computation of closure:

```
function closure ( I )
begin
        J := I;
        repeat
        for each item A → α.Bβ in J and each production
                B→γ of G such that B→.γ is not in J do
                        add B→.γ to J
        until no more items can be added to J
          return J

end
```

Example:

```
E' → E              closure({E' → .E}) =

E → E+T          { E' → .E   ----→  kernel items
E → T               E → .E+T
T → T*F             E → .T
T → F               T → .T*F
F → (E)             T → .F
F → id              F → .(E)
                    F → .id  }
```

[]Goto Operation:

- If I is a set of LR(0) items and X is a grammar symbol (terminal or non-terminal), then goto(I,X) is defined as follows:
  - If $A \to \alpha \bullet X\beta$ in I then every item in **closure({$A \to \alpha X \bullet \beta$})** will be in goto(I,X).

Example:

$I = \{\ E' \to \bullet E,\ \ E \to \bullet E{+}T,\ \ E \to \bullet T,$
$\quad T \to \bullet T{*}F,\ \ T \to \bullet F,$
$\quad F \to \bullet (E),\ \ F \to \bullet id\ \}$

$goto(I,E) = \{\ E' \to E \bullet,\ E \to E \bullet {+}T\ \}$

$goto(I,T) = \{\ E \to T \bullet,\ T \to T \bullet {*}F\ \}$

$goto(I,F) = \{ T \to F \bullet\ \}$

$goto(I,() = \{\ F \to (\bullet E),\ E \to \bullet E{+}T,\ E \to \bullet T,\ T \to \bullet T{*}F,\ T \to \bullet F,$
$\quad F \to \bullet (E),\ F \to \bullet id\ \}$

$goto(I,id) = \{\ F \to id \bullet\ \}$

[]Algorithm to construct sets of LR(0) items :

```
Procedure items(G')
Begin
   C is { closure({S'→.S}) }
   repeat .
      for each I in C and each grammar
        symbol X
        if goto(I,X) is not empty and not in C
           add goto(I,X) to C
   Until no more sets of LR(0) items
   can be added to C
end
```

The Canonical LR(0) Collection – Example:

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

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

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

I_3: $T \rightarrow F.$

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

I_5: $F \rightarrow id.$

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

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

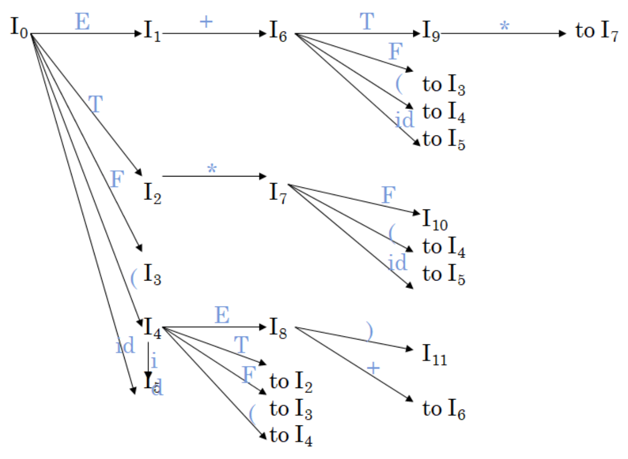
I_8: $E \rightarrow E.+T$
$F \rightarrow (E.)$

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

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

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

Transition Diagram (DFA) of Goto Function:



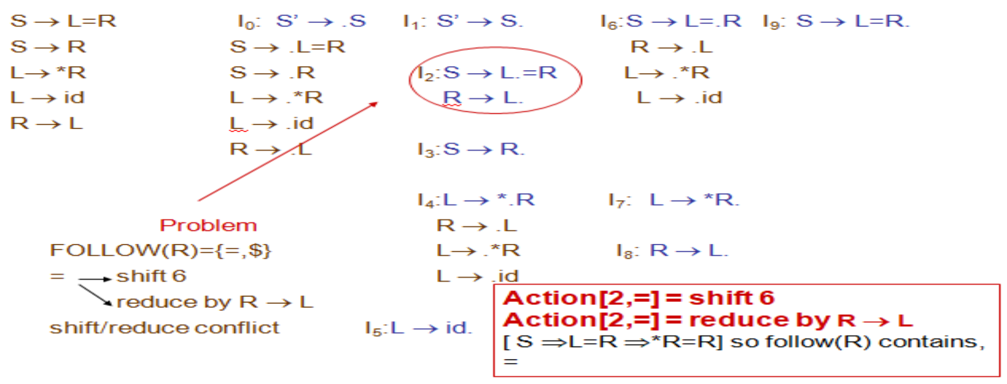
[]Algorithm to Construct SLR Parsing Table from an augmented grammar G’:

1. Construct the canonical collection of sets of LR(0) items for G’.
   $C \leftarrow \{I_0, \ldots, I_n\}$
2. State I is constructed from Ii. The parsing actions for state I are determined as follows
   - If a is a terminal, $A \rightarrow \alpha.a\beta$ in I_i and goto(I_i,a)=I_j then action[i,a] is ***shift j.***
   - If $A \rightarrow \alpha.$ is in I_i, then **action[i,a]** is ***reduce*** $A \rightarrow \alpha$ for all a in **FOLLOW(A)** where $A \neq S'$.
   - If $S' \rightarrow S.$ is in I_i, then action[i,$] is ***accept***.
   - If any conflicting actions generated by these rules, the grammar is not SLR(1).
3. Create the parsing goto table
   - for all non-terminals A, if goto(I_i,A)=I_j then goto[i,A]=j
4. All entries not defined by (2) and (3) are errors.
5. Initial state of the parser is the one constructed from the set of itemes containing $S' \rightarrow .S$

[]SLR(1) Grammar:

- An LR parser using SLR(1) parsing tables for a grammar G is called as the SLR(1) parser for G.
- If a grammar G has an SLR(1) parsing table, it is called SLR(1) grammar (or SLR grammar in short).
- Every SLR grammar is unambiguous, but every unambiguous grammar is not a SLR grammar.
- shift/reduce and reduce/reduce conflicts
- If a state does not know whether it will make a shift operation or reduction for a terminal, we say that there is a **shift/reduce conflict**.
- If a state does not know whether it will make a reduction operation using the production rule i or j for a terminal, we say that there is a **reduce/reduce conflict**.
- If the SLR parsing table of a grammar G has a conflict, we say that that grammar is not SLR grammar.

Conflict Example 1:



$S \rightarrow L=R$
$S \rightarrow R$
$L \rightarrow {*}R$
$L \rightarrow id$
$R \rightarrow L$

I_0: $S' \rightarrow .S$
$S \rightarrow .L=R$
$S \rightarrow .R$
$L \rightarrow .{*}R$
$L \rightarrow .id$
$R \rightarrow .L$

I_1: $S' \rightarrow S.$

I_2: $S \rightarrow L.=R$
$R \rightarrow L.$

I_3: $S \rightarrow R.$

I_4: $L \rightarrow {*}.R$
$R \rightarrow .L$
$L \rightarrow .{*}R$
$L \rightarrow .id$

I_5: $L \rightarrow id.$

I_6: $S \rightarrow L=.R$
$R \rightarrow .L$
$L \rightarrow .{*}R$
$L \rightarrow .id$

I_7: $L \rightarrow {*}R.$

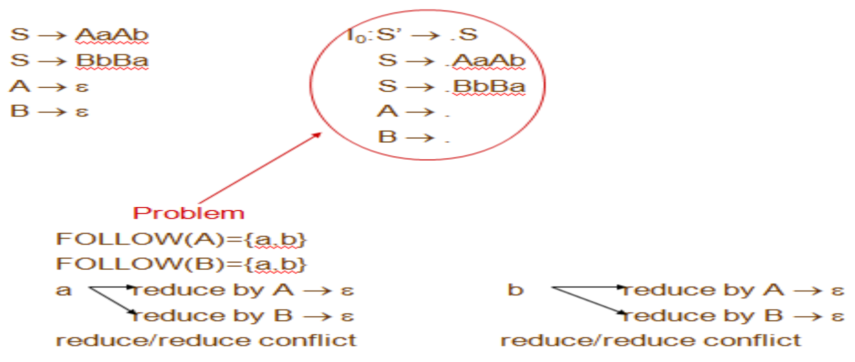
I_8: $R \rightarrow L.$

I_9: $S \rightarrow L=R.$

Problem
FOLLOW(R)={=,$}
= → shift 6
↘ reduce by $R \rightarrow L$
shift/reduce conflict

Action[2,=] = shift 6
Action[2,=] = reduce by $R \rightarrow L$
[$S \Rightarrow L=R \Rightarrow {*}R=R$] so follow(R) contains, =

Conflict Example 2:



[]Constructing Canonical LR(1) Parsing Tables:

In SLR method, the state i makes a reduction by A→α when the current token is **a**:

if the A→α. in the I_i and **a** is FOLLOW(A)

In some situations, βA cannot be followed by the terminal **a** in a right-sentential form when βα and the state i are on the top of stack. This means that making reduction in this case is not correct.

[]LR(1) Item:

To avoid some of invalid reductions, the states need to carry more information.

Extra information is put into a state by including a terminal symbol as a second component in an item.

A LR(1) item is:

A → α.β,a where **a** is the look-head of the LR(1) item

(**a** is a terminal or end-marker.)

Such an object is called LR(1) item.

1 refers to the length of the second component

The look ahead has no effect in an item of the form [A → α.β,a], where β is not ∊.

But an item of the form [A → α.,a] calls for a reduction by A → α only if the next input symbol is a.

The set of such a's will be a subset of FOLLOW(A), but it could be a proper subset.

When β (in the LR(1) item A → α.β,a) is not empty, the look-head does not have any affect.

When β is empty (A → α.,a), we do the reduction by A→α only if the next input symbol is **a** (not for any terminal in FOLLOW(A)).

[]Canonical Collection of Sets of LR(1) Items:

The construction of the canonical collection of the sets of LR(1) items are similar to the construction of the canonical collection of the sets of LR(0) items, except that *closure* and *goto* operations work a little bit different.

closure(I) is: (where I is a set of LR(1) items)

every LR(1) item in I is in closure(I)

if $A\rightarrow\alpha.B\beta,a$ in closure(I) and $B\rightarrow\gamma$ is a production rule of G; then $B\rightarrow.\gamma,b$ will be in the closure(I) for each terminal b in FIRST(βa) .

goto operation:

If I is a set of LR(1) items and X is a grammar symbol (terminal or non-terminal), then goto(I,X) is defined as follows:

If $A\rightarrow \alpha.X\beta,a$ in I then every item in **closure({$A\rightarrow \alpha X.\beta,a$})** will be in goto(I,X).

[]Construction of The Canonical LR(1) Collection:

Algorithm:

```
Procedure items(G')
Begin
  C = { closure({S'→.S,$}) }
  repeat
    for each I in C and each grammar symbol X
      if goto(I,X) is not empty and not in C
        add goto(I,X) to C
   Until until no more set of LR(I) items can be added to C.
 End.
```

Note: A Short Notation for The Sets of LR(1) Items-A set of LR(1) items containing the following items $A\rightarrow \alpha.\beta,a_1$

...

$A\rightarrow \alpha.\beta,a_n$

can be written as $A\rightarrow \alpha.\beta,a_1/a_2/.../a_n$

Example:

1. $S' \rightarrow S$
2. $S \rightarrow C\ C$
3. $C \rightarrow e\ C$
4. $C \rightarrow d$

I_0: closure({(S' → • S, $)}) =
(S' → • S, $)
(S → • C C, $)
(C → • e C, e/d)
(C → • d, e/d)

I_1: (S' → S • , $)

I_2: (S → C • C, $)
(C → • e C, $)
(C → • d, $)

I_3: (C → e • C, e/d)
(C → • e C, e/d)
(C → • d, e/d)

I_4: (C → d •, e/d)

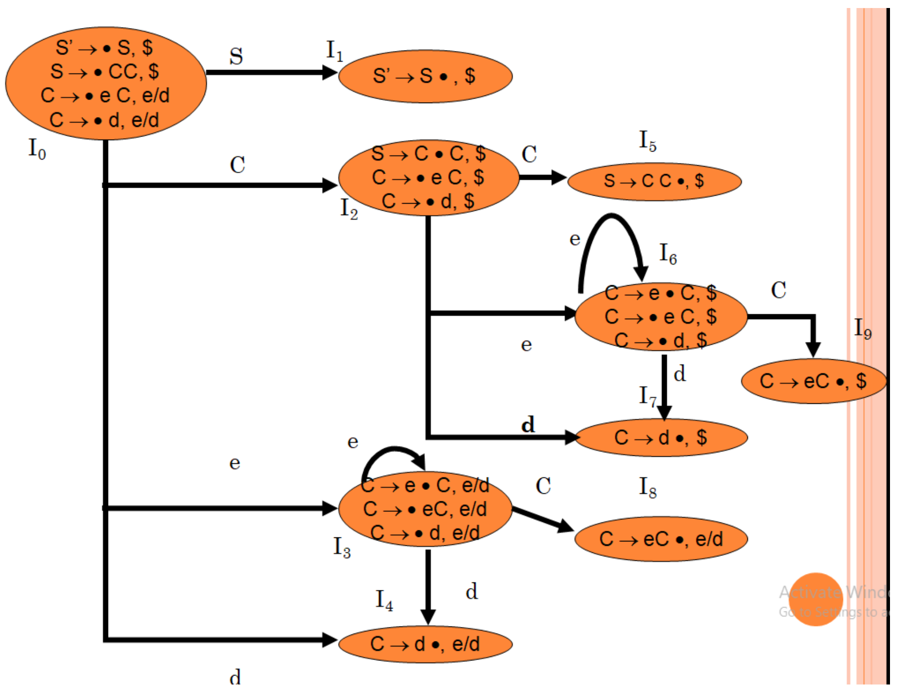
I_5: (S → C C •, $)

I_6: (C → e • C, $)
(C → • e C, $)
(C → • d, $)

I_7: goto(I_3, d) =
(C → d •, $)

I_8: (C → e C •, e/d)

I_9: goto(I_7, e) =
(C → e C •, $)



[]Construction of LR(1) Parsing Tables:

1. Construct the canonical collection of sets of LR(1) items for G'.
   $C \leftarrow \{I_0,\ldots,I_n\}$
2. Create the parsing action table as follows
   - If a is a terminal, $[A \to \alpha \bullet a\beta, b]$ in I_i and $\text{goto}(I_i,a)=I_j$ then action[i,a] is ***shift j.***
   - If $[A \to \alpha \bullet, a]$ is in I_i, then action[i,a] is ***reduce*** $A \to \alpha$ where $A \neq S'$.
   - If $[S' \to S \bullet, \$]$ is in I_i, then action[i,\$] is ***accept***.
   - If any conflicting actions generated by these rules, the grammar is not LR(1).
3. Create the parsing goto table
   - for all non-terminals A, if $\text{goto}(I_i,A)=I_j$ then goto[i,A]=j
4. All entries not defined by (2) and (3) are errors.
5. Initial state of the parser contains $[S' \to .S, \$]$

LR prsing tble for above problem:

	c	d	$	S	C
0	s36	s47		1	2
1			acc		
2	s36	s47			5
36	s36	s47			89
47	r3	r3	r3		
5			r1		
89	r2	r2	r2		

[]LALR Parsing Tables:

1. LALR stands for **Lookahead LR.**

2. LALR parsers are often used in practice because LALR parsing tables are smaller than LR(1) parsing tables.
3. The number of states in SLR and LALR parsing tables for a grammar G are equal.
4. But LALR parsers recognize more grammars than SLR parsers.
5. ***yacc*** creates a LALR parser for the given grammar.
6. A state of LALR parser will be again a set of LR(1) items.

[]Creating LALR Parsing Tables:

Canonical LR(1) Parser ➔ LALR Parser (shrink # of states)

This shrink process may introduce a **reduce/reduce** conflict in the resulting LALR parser (so the grammar is NOT LALR)

But, this shrink process does not produce a **shift/reduce** conflict.

The Core of A Set of LR(1) Items:

- The core of a set of LR(1) items is the set of its first component.

Ex: $S \rightarrow L\bullet{=}R,\$$ ➔ $S \rightarrow L\bullet{=}R$ ← Core

$R \rightarrow L\bullet,\$$ $R \rightarrow L\bullet$

- We will find the states (sets of LR(1) items) in a canonical LR(1) parser with same cores. Then we will merge them as a single state.

$I_1{:}L \rightarrow id\bullet,=$ A new state: $I_{12}{:}\ L \rightarrow id\bullet,=$

➔ $L \rightarrow id\bullet,\$$

$I_2{:}L \rightarrow id\bullet,\$$ have same core, merge them

- We will do this for all states of a canonical LR(1) parser to get the states of the LALR parser.
- In fact, the number of the states of the LALR parser for a grammar will be equal to the number of states of the SLR parser for that grammar.

The core of a set of LR(1) items is the set of its first component:

- The core of a set of LR(1) Items is the set of their first components (i.e., LR(0) items)
- The core of the set of LR(1) items

$\{ (C \rightarrow c \bullet C,\ c/d),$

$(C \rightarrow \bullet c\ C,\ c/d),$

$(C \rightarrow \bullet d,\ c/d) \}$

is $\{ C \rightarrow c \bullet C,$

$C \rightarrow \bullet c\ C,$

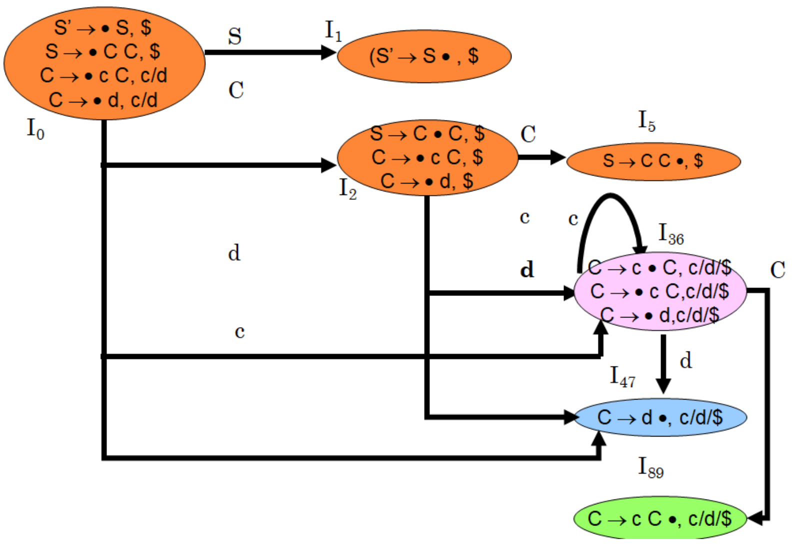
$C \rightarrow \bullet d \}$

[]Creation of LALR Parsing Tables:

1. Create the canonical LR(1) collection of the sets of LR(1) items for the given grammar.
2. For each core present; find all sets having that same core; replace those sets having same cores with a single set which is their union.

   $C=\{I_0,\ldots,I_n\}$ ➔ $C'=\{J_1,\ldots,J_m\}$ where $m \leq n$
3. Create the parsing tables (action and goto tables) same as the construction of the parsing tables of LR(1) parser.
   1. Note that: If $J=I_1 \cup \ldots \cup I_k$ since $I_1,\ldots,I_k$ have same cores
      ➔ cores of $goto(I_1,X),\ldots,goto(I_2,X)$ must be same.
   2. So, $goto(J,X)=K$ where K is the union of all sets of items having same cores as $goto(I_1,X)$.
4. If no conflict is introduced, the grammar is LALR(1) grammar. (We may only introduce reduce/reduce conflicts; we cannot introduce a shift/reduce conflict)

The above transition diagram noe becomes:



LALR Parse Table:

	c	d	$	S	C
0	s36	s47		1	2
1			acc		
2	s36	s47			5
36	s36	s47			89
47	r3	r3	r3		
5			r1		
89	r2	r2	r2		

[]Shift/Reduce Conflict:

- We say that we cannot introduce a shift/reduce conflict during the shrink process for the creation of the states of a LALR parser.
- Assume that we can introduce a shift/reduce conflict. In this case, a state of LALR parser must have:

  $A \to \alpha \bullet ,a$ and $B \to \beta \bullet a\gamma,b$

- This means that a state of the canonical LR(1) parser must have:

  $A \to \alpha \bullet ,a$ and $B \to \beta \bullet a\gamma,c$

  But, this state has also a shift/reduce conflict. i.e. The original canonical LR(1) parser has a conflict.

  (Reason for this, the shift operation does not depend on lookaheads)

[]Reduce/Reduce Conflict:

But, we may introduce a reduce/reduce conflict during the shrink process for the creation of the states of a LALR parser.

$$I_1 : A \to \alpha \bullet ,a \qquad\qquad I_2 : A \to \alpha \bullet ,b$$
$$\quad B \to \beta \bullet ,b \qquad\qquad\quad B \to \beta \bullet ,c$$

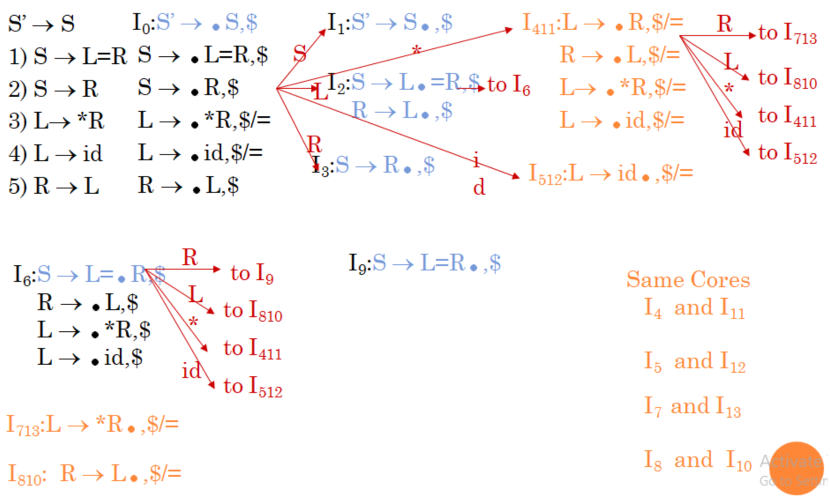
$$\Downarrow$$

$$I_{12} : A \to \alpha \bullet ,a/b \quad \Rightarrow$$

reduce/reduce conflict

$$B \to \beta \bullet ,b/c$$

Canonical LALR(1) Collection – Example2:



LALR(1) Parsing Tables – (for Example2):

	id	*	=	$	S	L	R
0	s512	S411			1	2	3
1				acc			
2			s6	r5			
3				r2			
411	s512	s411				810	713
512			r4	r4			
6	s512	s411				810	9
713			r3	r3			
810			r5	r5			
9				r1			

no shift/reduce or
no reduce/reduce conflict

$$\Downarrow$$

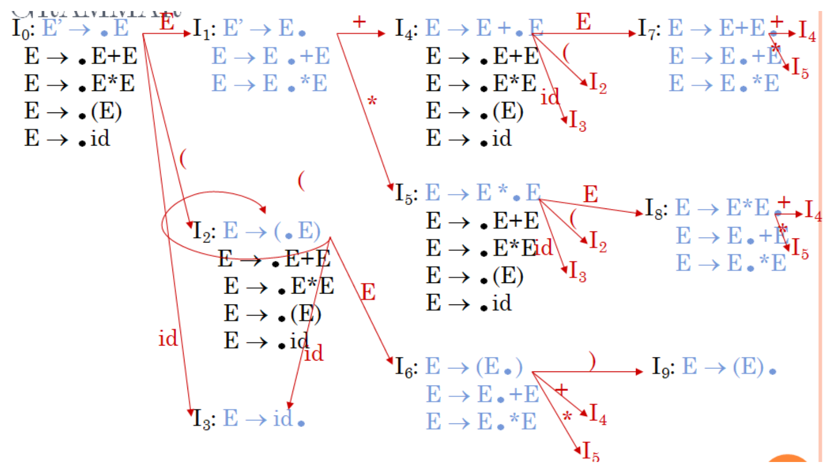
so, it is a LALR(1) grammar

[]Using Ambiguous Grammars:

- All grammars used in the construction of LR-parsing tables must be un-ambiguous.
- Can we create LR-parsing tables for ambiguous grammars ?
  - Yes, but they will have conflicts.
  - We can resolve these conflicts in favor of one of them to disambiguate the grammar.
  - At the end, we will have again an unambiguous grammar.
- Why we want to use an ambiguous grammar?
  - Some of the ambiguous grammars are **much natural**, and a corresponding unambiguous grammar can be very complex.
  - Usage of an ambiguous grammar may **eliminate unnecessary reductions**.
- Ex.

$$E \rightarrow E{+}E \mid E{*}E \mid (E) \mid id \quad \Rightarrow \quad \begin{array}{l} E \rightarrow E{+}T \mid T \\ T \rightarrow T{*}F \mid F \\ F \rightarrow (E) \mid id \end{array}$$

Sets of LR(0) Items for Ambiguous Grammar:



SLR-Parsing Tables for above problem:

	Action						Goto
	id	+	*	(	)	$	E
0	s3			s2			1
1		s4	s5			acc	
2	s3			s2			6
3		r4	r4		r4	r4	
4	s3			s2			7
5	s3			s2			8
6		s4	s5		s9		
7		r1	s5		r1	r1	
8		r2	r2		r2	r2	
9		r3	r3		r3	r3	

[]Error Recovery in LR Parsing:

- An LR parser will detect an error when it consults the parsing action table and finds an error entry. All empty entries in the action table are error entries.
- Errors are never detected by consulting the goto table.
- An LR parser will announce error as soon as there is no valid continuation for the scanned portion of the input.
- A canonical LR parser (LR(1) parser) will never make even a single reduction before announcing an error.
- The SLR and LALR parsers may make several reductions before announcing an error.
- But, all LR parsers (LR(1), LALR and SLR parsers) will never shift an erroneous input symbol onto the stack.

Panic Mode Error Recovery in LR Parsing:

- Scan down the stack until a state **s** with a goto on a particular nonterminal **A** is found. (Get rid of everything from the stack before this state s).
- Discard zero or more input symbols until a symbol **a** is found that can legitimately follow A.
- The symbol a is simply in FOLLOW(A), but this may not work for all situations.
- The parser stacks the nonterminal **A** and the state **goto[s,A]**, and it resumes the normal parsing.
- This nonterminal A is normally is a basic programming block (there can be more than one choice for A). stmt, expr, block, ...

Phrase-Level Error Recovery in LR Parsing:

- Each empty entry in the action table is marked with a specific error routine.
- An error routine reflects the error that the user most likely will make in that case.
- An error routine inserts the symbols into the stack or the input (or it deletes the symbols from the stack and the input, or it can do both insertion and deletion).

  * missing operand

  *unbalanced right parenthesis