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 finding 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 productions that have to be used for reduction.**

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.

BOTTOM-UP PARSING

PRINCIPLES OF BOTTOM UP PARSING

Example of Bottom Up Parsing

Consider the following grammar

$D \rightarrow$ `var` *list* `:` *type* `;`

type $\rightarrow$ `integer` | `real`

list $\rightarrow$ *list*`,id` | `id`

`var id,id : integer;` **is the input string.**

1. **The parse tree construction from the leaves is shown in Figure 3.13.**
2. **The sentential forms that were produced during the parse are**

   `var id , id : integer ;`

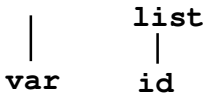
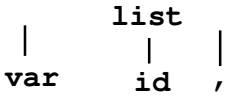
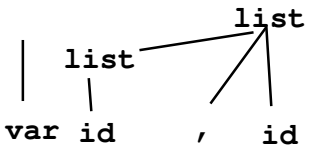
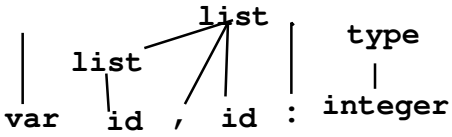
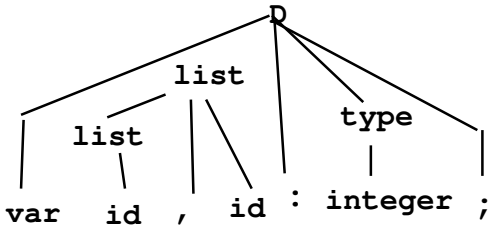
   `var` *list* `, id : integer ;`

   `var` *list* `: integer ;`

   `var` *list* `:` *type* `;`

   D
3. **The sentential forms happen to be a right most derivation in the reverse order.**

FIGURE 3.13 : Example of Bottom-Up Parsing

Remaining Input	Rule used in reduction	Parse Tree
var id,id:integer;	- - -	null tree
id,id:integer;	- - -	
,id:integer;	list → id	
id:integer;	- - -	
:integer;	list → list,id	
;	type → integer	
empty	D → var list : type ;	

BOTTOM-UP PARSING

PRINCIPLES OF BOTTOM UP PARSING

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, i.e.,**
   $S = \alpha_1 \Longrightarrow \alpha_2 \Longrightarrow \ \ldots \ \Longrightarrow \alpha_n = w$ **;**
   then given α_n it should be possible to construct α_{n-1} .
3. **α_{n-1} contains only one nonterminal, say A, but its position in this sentential form is not known. Examining tokens of w from left to right, one at a time, till the rhs of A is found, and then performing a reduction may 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.**

BOTTOM-UP PARSING

PRINCIPLES OF BOTTOM UP PARSING

1. **Example : Consider the grammar for expressions, whose rules are given by**

   $E \rightarrow E + T \mid T \qquad T \rightarrow T * F \mid F \qquad F \rightarrow \texttt{id} \mid (E)$

   along with the input `id + id * id`.
   Leftmost derivation of this sentence is :

   $E \Longrightarrow_{lm} E + T \Longrightarrow_{lm} T + T \Longrightarrow_{lm} F + T$
   $\Longrightarrow_{lm} \texttt{id} + T \Longrightarrow_{lm} \texttt{id} + T * F \Longrightarrow_{lm} \texttt{id} + F * F$
   $\Longrightarrow_{lm} \texttt{id} + \texttt{id} * F \Longrightarrow_{lm} \texttt{id} + \texttt{id} * \texttt{id}$

   (a) **There are three instances of** `id` **in the input that matches with the rhs of the rule,** $F \rightarrow \texttt{id}$.

   (b) **Reducing the first** `id` **in** `id + id * id` **produces** F + `id * id` **which is not the previous sentential.**

   (c) **In order to produce the previous sentential, only the third** `id` **in** `id + id * id` **should be used in the reduction.**

   (d) **The parser would have several candidates for reduction (only one correct) and it is not easy to identify the right one.**

BOTTOM-UP PARSING

PRINCIPLES OF BOTTOM UP PARSING

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.

BOTTOM-UP PARSING

PRINCIPLES OF BOTTOM UP PARSING

Definition of a Handle

- **A *handle* of a right sentential form , say γ, is a production rule $A \rightarrow \beta$, and a position of 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.**
- **Formally, if** $S \overset{*}{\Longrightarrow}_{rm} \alpha\ A\ \mathtt{w} \overset{*}{\Longrightarrow}_{rm} \alpha\beta\ \mathtt{w}$
  then rule $A \rightarrow \beta$ in the position following α in γ is a handle of γ ($= \alpha\beta\ \mathtt{w}$).

  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.**

  i. a buffer for holding the input string to be parsed.

BOTTOM-UP PARSING

SHIFT REDUCE PARSING

ii. a data structure for detecting handles (a stack happens to be adequate)

iii. 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 (popping off the rhs of a rule from the stack and pushing its lhs), this move is called a *reduce* action.

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

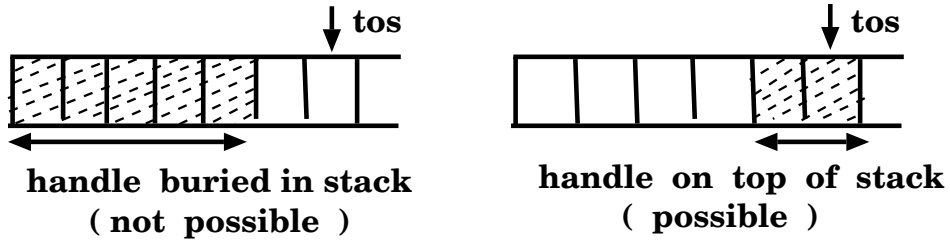
Step 4. A situation when the parser can neither shift nor reduce nor accept, it declares an error, *error* action and halts.

SHIFT REDUCE PARSING

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, 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 (i.e., the right end of the handle is the top of stack) and is never buried inside the stack.

FIGURE 3.15 : Occurrence of Handle in the stack



SHIFT REDUCE PARSING

Figure 3.14 : Example of Shift Reduce Parsing

grammar is

$$E \rightarrow E + T \mid E - T \mid T$$
$$T \rightarrow T * F \mid T / F \mid F$$
$$F \rightarrow P \uparrow F \mid P$$
$$P \rightarrow - P \mid B$$
$$B \rightarrow (E) \mid id$$

Stack	Input	Parser Moves
tos ↓ $	- id↑ id / id $	shift -
$-	id↑ id / id $	shift id
$- id	↑ id / id $	reduce by r12
$- B	↑ id / id $	reduce by r10
$- P	↑ id / id $	reduce by r9
$ P	↑ id / id $	shift ↑

Complete the rest of the parse

SHIFT REDUCE PARSING

HANDLE DETECTION IN SHIFT REDUCE PARSING

- **Example : If the stack contents happens to be αAz, with z on *tos*, then possible handles are :**
  z A z αAz and such strings
  while *substrings of* α α αA are not handles.

v) *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 $a_1 a_2 \ldots a_n$ be the sentence to be parsed and
$S \overset{*}{\Longrightarrow}_{rm} a_1 a_2 \ldots a_i A a_{i+k+1} \ldots a_n \overset{*}{\Longrightarrow}_{rm} a_1 a_2 \ldots a_n$
The last two sentential forms (say m and m-1) are shown above and $A \rightarrow a_{i+1} \ldots a_{i+k}$ is the rule used.
In this case shifting symbols upto a_{i+k} would expose the handle on top of the stack . Similar arguments hold for the other possibilities of placement of A.
general case **: Consider the stack contents at some intermediate stage of parsing , say j^{th} sentential form, and assume the handle is on (*tos*) as shown below.**

stack	input
α B	x y z

SHIFT REDUCE PARSING

Proof Outline **(Continued)**

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

	$(j-1)^{th}$ **sentential**	j^{th} **sentential**	**rule used**
a)	$\alpha B\mathtt{x}A\mathtt{z}$	$\alpha B\mathtt{xyz}$	$A \rightarrow \mathtt{y}$
b)	$\delta A\mathtt{yz}$	$\alpha B\mathtt{xyz}$	$A \rightarrow \beta B\mathtt{x}$

- **In (i) after shifting y , the handle would be found on *tos* as claimed.**
- **In (ii), $\alpha = \delta\beta$ and even in this situation, shifting of zero or more symbols would allow this handle to occur on *tos* and get detected.**

vi) The major implication of the above theoretical result is

1. **right end of a handle is guaranteed to occur at the top of the stack**
2. **left end of the handle would have to be found by examining as many symbols as in the rhs of the rule**
3. **searching space for a handle is reduced considerably**
4. **justifies the use of stack for detecting handles**

SHIFT REDUCE PARSING

PARSE TREE CONSTRUCTION

A parse tree can be easily constructed along with parsing by a shift reduce parser

- **the stack is extended to have another field for each entry, a pointer to a tree associated with a stack symbol. The code for tree construction is added to the basic moves of the parser.**

  (i) *shift* `a` **;** **create a leaf** **labeled** `a`**; the root of this tree is associated with the stack symbol,** `a`

  (ii) *reduce* **by** $A \rightarrow X_1 X_2 \ldots X_n$ **;**
  A **new node** **labeled** A **is created ;** **subtrees** **rooted at** $X_1, \ldots, X_n$ **are** **made its children in the left to right order**.
  The pointer for this tree rooted at A **is stored alongwith** A **in the stack.**

  (iii) reduction by a rule, $A \rightarrow \epsilon$**, is handled by creating a tree with root labeled** A **which has a leaf labeled** ϵ **as its only child. The pointer to the root of this tree is saved alongwith** A **in the stack.**

SHIFT REDUCE PARSING

LIMITATIONS OF SHIFT REDUCE PARSER

- **What kind of grammars (languages) would not admit a shift reduce parser?**

1. **a handle , say β, occurs at *tos*; the *nexttoken*, say a, is such that βaγ happens to be another handle. The parser has two options**

(a) **reduce the handle using $A \rightarrow \beta$; in such a case the other handle , $B \rightarrow \beta$aγ , may not get exposed for reduction and hence missed**

(b) **ignore the handle β; shift a and continue parsing in which case reduction using $B \rightarrow \beta$aγ would be possible, but at the cost of missing the handle β.**
This situation is described as a *shift reduce* conflict.

2. **the handle β, is such that it forms the rhs of two or more rules, say $A \rightarrow \beta$ and $B \rightarrow \beta$**
Then the parser has two or more reduce possibilities. This situation is known as a *reduce reduce* conflict.

- **How do shift reduce parsers handle such conflicts ?**
The nexttoken could be used to prefer one move over the other. Choose shift (or reduce) in a shift reduce conflict, prefer one reduce (over others) in a reduce reduce conflict.

LR PARSERS

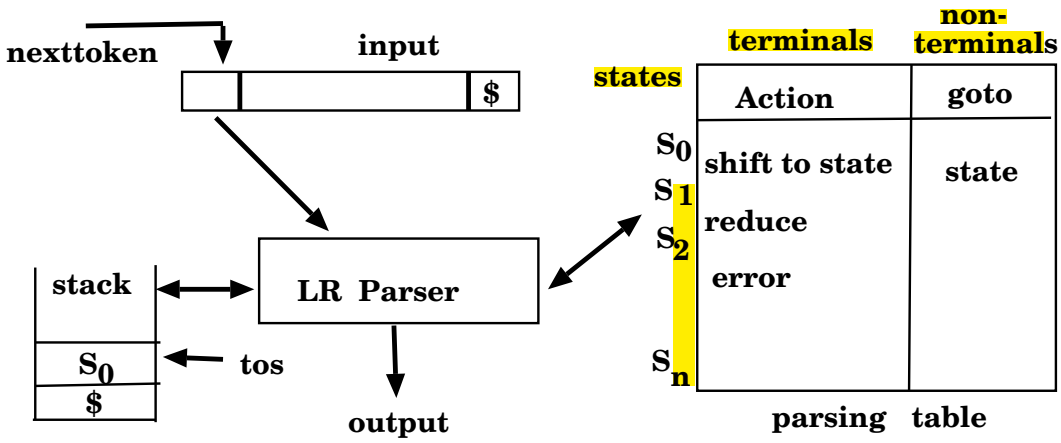
SOME FACTS ABOUT LR PARSERS

- **all the parsers in the family are shift reduce parsers, the basic parsing actions are *shift*, *reduce*, *accept* and *error***
- **these are the most powerful of all deterministic parsers in practice; they subsume the class of grammars that can be parsed by predictive parsers**
- **LR parsers can be written to recognize most of the PL constructs which can be expressed through a cfg**
- **the overall structure of all the parsers is the same. All are table driven , 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 Figure 3.16. The parser apart from grammar symbols, uses extra symbols (called *states*). While both grammar and state symbols can appear on the stack, *tos* , however, always is a state.

FIGURE 3.16 : LR PARSER MODEL



Initial Configuration

Working of a LR Parser

tos	nexttoken	Action	parser action
S_j	a	s i	push a ; push S_i ; continue
	a	rj	rj : A → α ; |α| = r ; pop 2 r symbols ; tos= S_k ; goto[S_k , a] =S_l ; push A ; push S_l ; continue
S_j	$	acc	successful parse ; halt
S_j	a	error	error handling

LR PARSERS

EXPLANATION OF TERMS USED IN FIGURE 3.16

The parser uses two data structures prominently.

1. **a stack which contains strings of the form $s_0X_1s_1X_2 \ldots X_ms_m$, 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**
     i) s_i **which means shift to state i**
     ii) r_j **which stands for reduce by the** j^{th} **rule,**
     iii) *accept*
     iv) *error*

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

SLR(1) PARSER

CONFIGURATION OF A LR PARSER

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 in Figure 3.17.**
- **The working of this parser for a sample input is shown in Figure 3.18.**

SLR(1) PARSER

FIGURE 3.17 : SLR(1) PARSING TABLE

LR Parsing Table for an expression grammar

Grammar : E → E + T | T
T → T * F | F
F → (E) | id

	Action						Goto		
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			

SLR(1) PARSER

FIGURE 3.18 : WORKING OF A SLR(1) PARSER

Parsing the Input * a = * * b using LR Parsing Table of Fig 3.17

Stack	Input	Action table	Goto table
$ 0	* a = * * b $	[0 , *] = s4	—
$ 0 * 4	a = * * b $	[4 , id] = s5	—
$ 0 *4 id 5	= * * b $	[5 , =] = r 4	[4 , L] = 8
$ 0 *4 L 8	= * * b $	[8 , =] = r 5	[4 , R] = 7
$ 0 *4 R 7	= * * b $	[7 , =] = r 3	[0, L] = 2
$ 0 L 2	= * * b $	[2 , =] = s 6	—
$ 0 L 2 = 6	* * b $	[6 , *] = s12	—
$ 0 L 2 = 6 * 12	* b $	[12 , *] = s12	—
$ 0L2=6*12*12	b $	[12 , id] = s11	—
$ 0L2=6*12*12id 11	$	[11 , $] = r 4	[12, L] = 10
$ 0L2=6*12*12L10	$	[10 , $] = r 5	[12 , R] = 13
$ 0L2=6*12*12R13	$	[13 , $] = r 3	[12 , L] = 10
$ 0 L 2 = 6 *12 L 10	$	[10 , $] = r 5	[12 , R] =13
$ 0L 2 = 6 *12 R 13	$	[13, $] = r 3	[6, L] = 10
$ 0 L 2 = 6 L 10	$	[10, $] = r5	[6, R] = 9
$ 0 L 2 = 6 R 9	$	[9, $] = r1	[0, S] = 1
$ 0 S 1	$	[1, $] = acc	—

SLR(1) PARSER

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** $(s_0, a_1 a_2 \ldots a_n\$)$**, where** s_0 **is a designated *start* state and the second component is the entire sentence to be parsed.**
- **Let an intermediate configuration be given by** $(s_0 X_1 s_1 \ldots X_i s_i$ **,** $a_j a_{j+1} \ldots a_n\$)$ **, then resulting configuration**

  i) after a shift action is given by
  $(s_0 X_1 s_1 \ldots X_i s_i a_j s_k, a_{j+1} \ldots a_n\$)$
  provided $Action[s_i, a_j] = s_k$ **; both *stack* and *nexttoken* change after a shift.**

  ii) after a reduce action is given by
  $(s_0 X_1 s_1 \ldots X_{i-r} A s, a_j \ldots a_n\$)$
  where $Action[s_i, a_j] = r_l$ **; rule** $p_l : \ A \rightarrow \beta$**,** 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. The material would be covered in the following order.

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 : For the rule given below**

  $decls \rightarrow decls\ decl$

  the possible LR(0) items are :

  $I_1 : decls \rightarrow \bullet decls\ decl$

  $I_2 : decls \rightarrow decls\ \bullet decl$

  $I_3 : decls \rightarrow decls\ decl \bullet$

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

  $I_4 : decls \rightarrow\ \bullet$

SLR(1) PARSER CONSTRUCTION

DEFINITION OF LR(0) ITEM AND RELATED TERMS

- *Incomplete and Complete LR(0) items* : **An LR(0) item is complete if the • 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 : I_3 and I_4 are complete items and I_1 and I_2 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.

- *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 $S' \rightarrow \bullet S$ 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 : I_1, I_2, I_3 are all kernel items and I_4 is a nonkernel item.

SLR(1) PARSER CONSTRUCTION

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 has the form** $f : U \rightarrow 2^U$ **and is constructed as follows :**

  i) *closure*(**I**) = **{ I }, for I** $\in$ **U**

  ii) If $A \rightarrow \alpha \bullet B\beta \in$ *closure*(**I**)**, then for every rule of the form** $B \rightarrow \eta$**, the item** $B \rightarrow \bullet\eta$ **is added to (if it is not already there)** *closure*(**I**)**.**

  iii) Apply step (ii) above repeatedly till no more new items can be added to *closure*(**I**)**.**

  closure(**I**) $\neq$ **{I} only when I is an item in which a nonterminal immediately follows the dot.**

- **The** *goto* **function has the form** $f : U \times X \rightarrow 2^U$**, where** X **is a grammar symbol.**

SLR(1) PARSER CONSTRUCTION

CANONICAL COLLECTION OF LR(0) ITEMS

- ***goto* is defined using *closure* as follows**
  $goto(A \rightarrow \alpha \bullet X\beta\ ,X) = closure(A \rightarrow \alpha X \bullet \beta\ )$.
  Clearly $goto$**(I,**X**) for a complete item I is** Φ**, for all** X**.**

- **Example of *closure* and *goto* : Consider** $A \rightarrow A$ `a` | `b`
  (a) $closure(A \rightarrow \bullet A$ `a`$) =$ **{** $A \rightarrow \bullet A$ `a` **}, by rule (i).**
  The item $A \rightarrow \bullet$ `b` **is added to it by rule (ii).**
  $closure(A \rightarrow \bullet A$ `a`$)$ **= {** $A \rightarrow \bullet A$ `a`,$A \rightarrow \bullet$ `b` **} after step(iii).**

  (b) $goto(A \rightarrow \bullet A$ `a`, $A) = closure(A \rightarrow\ A\bullet$ `a`$)$
  = { $A \rightarrow\ A\bullet$ `a` **}**

- **The functions *closure* and *goto* can be extended to a set S of LR(0) items by appropriate generalizations :**
  $closure$**(S)** $= \cup_{\mathtt{I} \in \mathtt{S}}$ **{**$closure$**(I) }** ;
  $goto$**(S,**X**)** $= \cup_{\mathtt{I} \in \mathtt{S}}$ **{**$goto$**(I,**X**) }**
  Algorithm 3.2 gives a pseudo code for computing 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 *closure* and *goto* as given in Algorithm 3.2.**

SLR(1) PARSER CONSTRUCTION

CANONICAL COLLECTION OF LR(0) ITEMS

Algorithm 3.2

procedure *items*(G', **C**);

    begin

        i := 0; I_0 **:= {** *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** *goto*(I_i,X) **is not empty and** $\notin$ **C,** do

            i := i + 1 ; I_i **:=** *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

SLR(1) PARSER CONSTRUCTION

CANONICAL COLLECTION OF LR(0) ITEMS

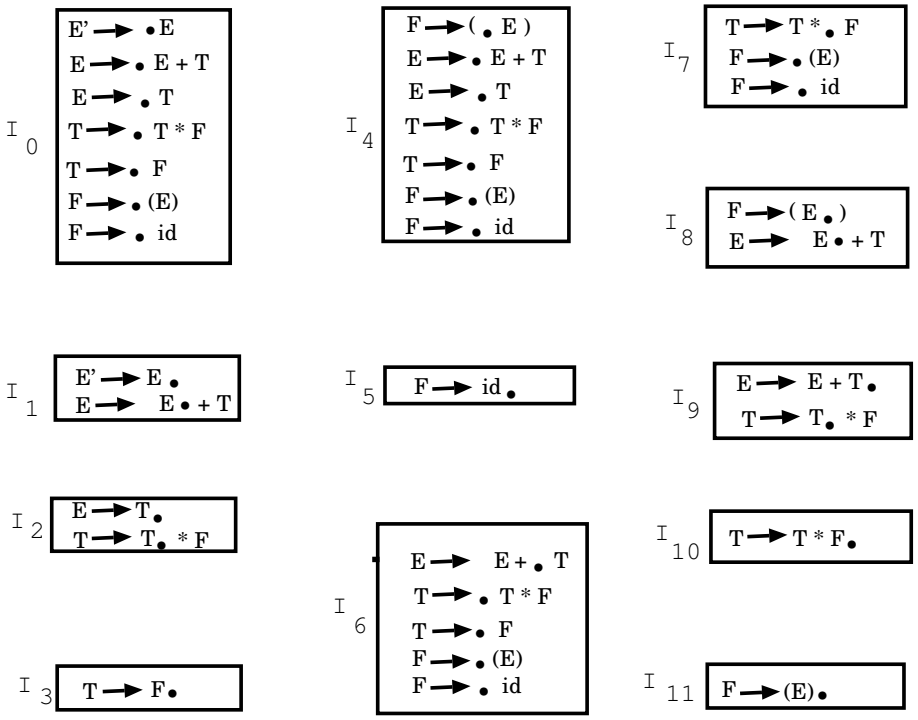
```
function goto(I, X)
    begin
        goto_items := φ ;
        for  each item A → α • Xβ ∈ I,
             such that A → αX • β is not in goto_items
             do add A → αX • β to goto_items
        goto := closure(goto_items);
        return goto ;
    end
```

Illustration of construction of canonical collection

- **Consider the expression grammar of Figure 3.17 . I_0 is closure($E' \rightarrow \bullet E$). The items $E' \rightarrow \bullet E$, $E \rightarrow \bullet E+T$ and $E \rightarrow \bullet T$ are added to I_0. The last one in turn causes the addition of $T \rightarrow \bullet T * F$ and $T \rightarrow \bullet F$. The item $T \rightarrow \bullet F$ leads to the addition of $F \rightarrow \bullet(E)$ and $F \rightarrow \bullet$ `id`. No more items can be added and the collection I_0 is complete (see Figure 3.19)**
- **I_0 has items having $\bullet\alpha$ combination for $\alpha =$ { E , T, F, (, `id` }. $goto(I_0, \alpha)$ is defined for all these values of α which in turn give rise to different collections as shown in Figure 3.19.**

SLR(1) PARSER CONSTRUCTION

FIGURE 3.19 : Canonical Collection of LR(0) items



SLR(1) PARSER CONSTRUCTION

CONSTRUCTION OF SLR(1) PARSING TABLE

From the collection C constructed using Algorithm 3.2, one can directly construct a SLR(1) parsing table. The procedure is described in Algorithm 3.3.

Algorithm 3.3 :
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** $C = \{I_0, I_1, \ldots, I_n\}$**.**
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 goto(**I_i**,a)** $= I_j$ **, then**
   Action[i,a] $= s_j$ **, i.e., shift to state j**

SLR(1) PARSER CONSTRUCTION

CONSTRUCTION OF SLR(1) PARSING TABLE

5.2 If $A \rightarrow \alpha\bullet \in I_i$, then
Action[i, a] $= r_j$, $\forall$ **a** $\in$ **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
Action[i,$] $=$ **accept**

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

5.5 For nonterminals A , such that $goto(I_i, A) = I_j$ create
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 i are decided by the $goto(I_i, A)$ **states where** A **is a nonterminal.**

The SLR(1) table for the canonical collection of Figure 3.19 is partly constructed in Figure 3.20.
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.

SLR(1) PARSER CONSTRUCTION

FIGURE 3.20 : Illustration of Algorithm 3.3

State i	Item with . t in I_i	GOTO (I_i, t)	Complete item	Action Table
0	F → . id F → . (E)	I_5 I_4	none	shift 5 shift 4
1	E → E. + T	I_6	E' → E.	shift 6 acc on $

State i	Item with • A in I_i	GOTO (I_i, A)	goto Table
0	A = E A = T A = F	I_1 I_2 I_3	1 2 3
1	none		

SLR(1) PARSER CONSTRUCTION

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.

- 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 $\mathtt{a} \in$ **FOLLOW**(A)
  and also
  (b) an incomplete item of the form $B \rightarrow \beta\bullet\ \mathtt{a}\ \gamma$

- 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) **FOLLOW**$(A) \cap$ **FOLLOW**$(B) \neq \Phi$

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

1. What are the state symbols used by the parser and what information do these contain ?

THEORY OF LR PARSING

VIABLE PREFIXES AND VALID ITEMS

2. **Where exactly is handle detection taking place in the parser?**
3. **Why is FOLLOW information used to create the reduce entries in the action table ?**

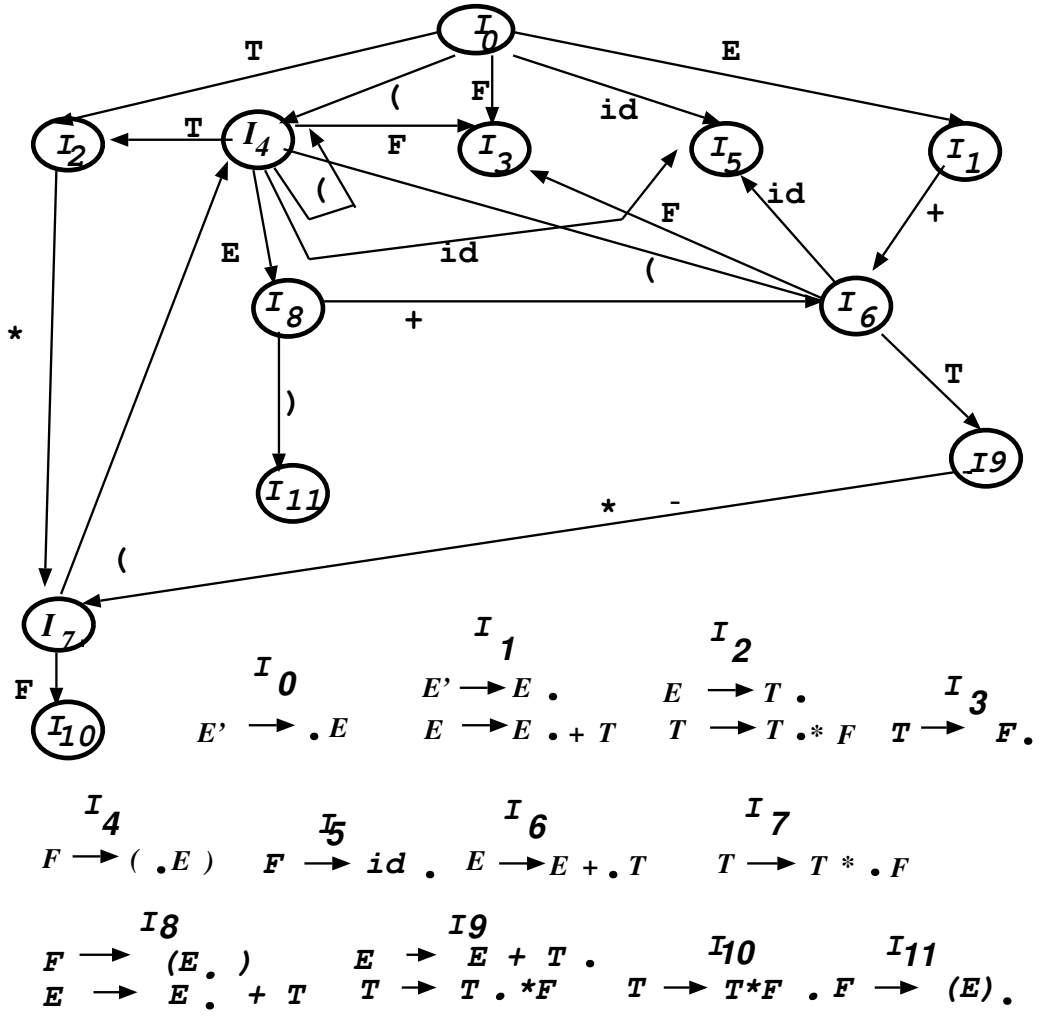
The canonical collection of LR(0) items can also be represented by a directed labeled graph.

- **A node labeled I_i is constructed for each member of C.**
- **For every nonempty goto(I_i, X) = I_j , a directed edge (I_i, I_j) is added labeled with X.**
- **The graph is a deterministic finite automaton if the node labeled I_0 is treated as the *start* state and all other nodes are made final states.**
- **The finite automaton associated with the collection of items given in Figure 3.19 is shown in Figure 3.21 . The crucial question is to understand what this automata recognizes. The following terms are defined in order to precisely characterize these string of symbols.**

VIABLE PREFIX

A *viable prefix* of a grammar is defined to be the prefixes of right sentential forms that do not contain any symbols to the right of a handle.

FIGURE 3.21 : LR AUTOMATON FOR EXPRESSION GRAMMAR



Only the kernel items of each state is shown in the above

THEORY OF LR PARSING

VIABLE PREFIX AND VALID ITEMS

1. **By adding terminal symbols to viable prefixes, rightmost sentential forms can be constructed.**
2. **Viable prefixes are precisely the set of symbols that can ever appear on the stack of a shift reduce parser**
3. **A viable prefix either contains a handle or contains parts of one or more handles.**
4. **Given a viable prefix, if one could identify the set of potential handles associated with it then a recognizer for viable prefixes would also recognize handles.**

The significance of LR(0) item can now be understood. A complete item corresponds to a handle while an incomplete item indicates the part of a handle seen so far.
A LR(0) item $A \rightarrow \beta_1 \bullet \beta_2$ is defined to be *valid* for a viable prefix, $\alpha\beta_1$, provided $S \overset{*}{\Longrightarrow}_{rm} \alpha A \texttt{ w} \Longrightarrow_{rm} \alpha\beta_1\beta_2 \texttt{ w}$

1. **It is interesting to note that in above, if $\beta_2 = B\gamma$ and $B \rightarrow \delta$, then $B \rightarrow \bullet\delta$ is also a valid item for this viable prefix.**
2. **There could be several distinct items which are valid for same viable prefix γ.**

THEORY OF LR PARSING

VIABLE PREFIXES AND VALID ITEMS

3. **A particular item may be valid for many distinct viable prefixes.**

4. **The parser would halt when the viable prefix S is seen, in the item sense, it indicates a move from $\bullet S$ to $S\bullet$. This is the reason for augmenting the input grammar.**

- **Example : For the LR-automaton given in Figure 3.21, these concepts and their relationships are explained in the following.**

- **Consider the path { I_0, I_4, I_8, I_6 } which has the label (E + along it. The items that are valid for the viable prefix, (E + are determined below.**

  1. $E' \Longrightarrow_{rm} E \Longrightarrow T \Longrightarrow F \Longrightarrow (E) \Longrightarrow (E+T)$ **shows that** $E \rightarrow E + \bullet T$ **is a valid item**

  2. $E' \Longrightarrow E \Longrightarrow T \Longrightarrow F \Longrightarrow (E) \Longrightarrow (E+T) \Longrightarrow (E+T*F)$ **shows that** $T \rightarrow \bullet T * F$ **is also a valid item.**

  3. $E' \overset{*}{\Longrightarrow}_{rm} (E+T) \Longrightarrow (E+F)$ **shows that** $T \rightarrow \bullet F$ **is another such item.**

  4. $E' \overset{*}{\Longrightarrow}_{rm} (E+T) \Longrightarrow (E+F) \Longrightarrow (E+(E))$ **shows that** $F \rightarrow \bullet(E)$ **is a valid item.**

  5. $E' \overset{*}{\Longrightarrow}_{rm} (E+F) \Longrightarrow (E+$`id`) **shows that** $F \rightarrow \bullet$ `id` **is a valid item.**

  It should be noted that are no other valid items for this viable prefix.

THEORY OF LR PARSING

VIABLE PREFIXES AND VALID ITEMS

- **Given a LR(0) item, say $T \rightarrow T \bullet *F$, there may be several viable prefixes for which it is valid.**

  1. $E' \Longrightarrow_{rm} E \Longrightarrow T \Longrightarrow T * F$ **shows that this item is valid for the viable prefix** T **.**

  2. $E' \Longrightarrow E \Longrightarrow T \Longrightarrow F \Longrightarrow (E) \Longrightarrow (T) \Longrightarrow (T*F)$ **shows that it is also valid for (** T **.**

  3. $E' \Longrightarrow E \Longrightarrow T \Longrightarrow T*F \Longrightarrow T*(E) \Longrightarrow T*(T) \Longrightarrow T*(T*F)$ **shows that it is valid also for** T *** (** T **.**

  4. $E' \Longrightarrow E \Longrightarrow E+T \Longrightarrow E+T*F$ **shows validity for** $E+T$ **. There may be several other viable prefixes for which this item is valid.**

- *THEOREM :* **The set of all valid items for a viable prefix γ is exactly the set of items reached from the start state along a path labeled γ in the DFA that can be constructed from the canonical collection of sets of items, with the transitions given by** *goto***.**

THEORY OF LR PARSING

VIABLE PREFIXES AND VALID ITEMS

- **The theorem stated without proof above is a key result in LR Parsing. It provides the basis for the correctness of the construction process we learnt earlier.**

- **An LR parser does not scan the entire stack to determine when and which handle appears on top of stack (compare with shift reduce parser).**

- **The state symbol on top of stack provides all the information that is present in the stack.**

- **In a state which contains a complete item a reduction is called for. However, the lookahead symbols for which the reduction should be applied is not obvious.**

- **In SLR(1) parser the FOLLOW information is used to guide reductions.**

CANONICAL LR(1) PARSER

Canonical LR(1) parser is the next parser that we study. In order to motivate the need for having another LR parser, the limitations of SLR(1) parser are pointed out.

FOLLOW information is imprecise

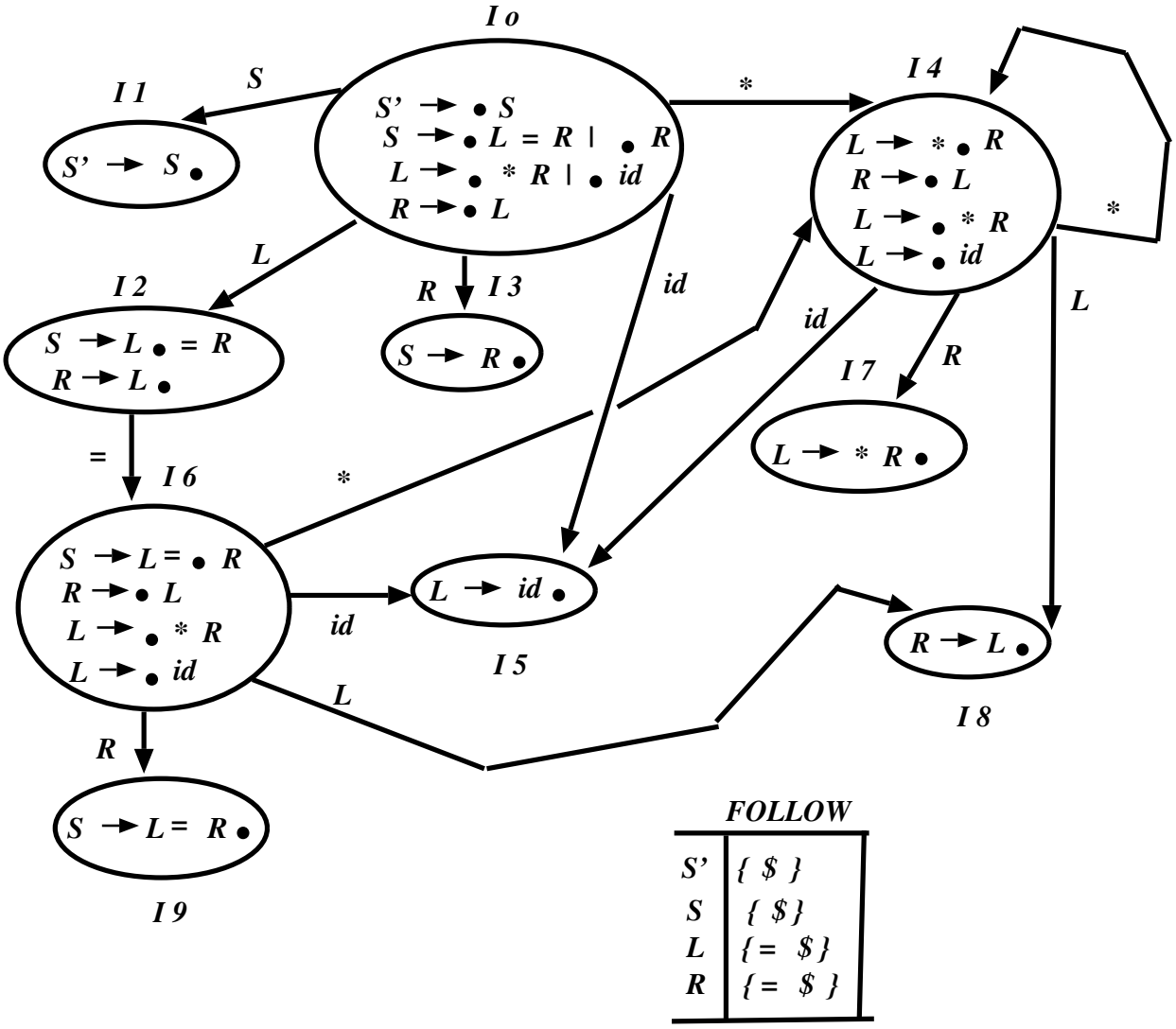
- **An example to illustrate this fact is the following grammar, which can be seen to be SLR(1) unambiguous.**

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

- **The SLR automaton and the parsing table for this grammar are given in the Figures 3.22 and 3.23 respectively.**
- **Observe that $\{=\} \in$ FOLLOW(R), and $R \rightarrow L\bullet$ is an item in state 2. There is another item, $S \rightarrow L\bullet = R$ in the same state**
- **Note that state 2 recognizes the viable prefix L and the shift entry is justifiable.**
- **How about the reduce entry ? After seeing L, if we reduce it to R, with the expectation of = to follow, there must exist a viable prefix $R = \ldots$ and hence a right sentential form of the form $R =$ z. It can be shown that such is not possible. The problem seems to be with our FOLLOW information.**

LR(1) PARSER CONSTRUCTION

FIGURE 3.22 : SLR AUTOMATON FOR A GRAMMAR



LR(1) PARSER CONSTRUCTION

FIGURE 3.23 : SLR PARSING TABLE

Grammar : S′ → S
S → L = R | R
L → * R | id
R → L

	Action				Goto		
state	id	*	=	$	S	L	R
0	s5	s4	–	–	1	2	3
1	–	–	–	acc	–	–	–
2	–	–	s6 r5	r5	–	–	–
3	–	–	–	r2	–	–	–
4	s5	s4	–	–	–	8	7
5	–	–	r4	r4	–	–	–
6	s5	s4	–	–	–	8	9
7	–	–	r3	r3	–	–	–
8	–	–	r5	r5	–	–	–
9	–	–	–	r1	–	–	–

LR(1) PARSER CONSTRUCTION

LR(1) ITEM AND LOOKAHEAD SYMBOLS

What is wrong with FOLLOW ?

- **This information is not correct for state 2.**
- **The sentential forms that permit = to follow R are of the form $*L\ldots$ and taken care of in the state 8 of the parser.**
- **The context in state 2 is different (viable prefix is L), and use of FOLLOW constrains the parser.**

Given an item and a state the need is to identify the terminal symbols that can actually follow the lhs nonterminal.

An item of the form, $A \rightarrow \alpha \bullet \beta$, `a` where the first component is a LR(0) item and the second component is a set of terminal symbols is called a LR(1) item. The second component is known as *lookahead* symbol. The value 1 indicates that the length of lookahead is 1.

- **The lookahead symbol is used in a reduce operation only. For an item of the form, $A \rightarrow \alpha \bullet \beta$, `a` if β is not ϵ, the lookahead has no effect. But for an item, $A \rightarrow \alpha\bullet$, `a` the reduction is applied only if the *nexttoken* is `a`.**

LR(1) PARSER CONSTRUCTION

VALID LR(1) ITEM

- **An LR(1) item $A \to \alpha \bullet \beta$, a is valid for a viable prefix γ, if there is a derivation**
  $S \overset{*}{\Longrightarrow}_{rm} \delta A \texttt{ w} \Longrightarrow_{rm} \delta\alpha\beta \texttt{ w}$, **where**
  i) $\gamma = \delta\alpha$ **,and**
  ii) either a **is the first symbol of** w**, or** w **is** ϵ **and** a **is** $

LR(1) sets of items collection

The basic procedure is same as for LR(0) sets of items collection, if we specify how to compute the lookahead for the new item obtained by closure.

- **If $A \to \alpha \bullet B\beta$,** a **is a valid item for a viable prefix γ, then the item $B \to \bullet\eta$,** b **is also valid for the same prefix γ. Here $B \to \eta$ is a rule and** b $\in$ **FIRST(β** a**).**

CONSTRUCTING LR(1) SETS OF ITEMS

The algorithm is similar to Algorithm 3.3 and given in the following.

LR(1) PARSER CONSTRUCTION

CANONICAL COLLECTION OF LR(1) SETS OF ITEMS

```
procedure items(G');
    begin
        C := { closure ( S' → •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 already in C, do add goto(I,X) to C
        until no more sets of items can be added to C
    end
```

```
function closure(I);
    begin
        repeat
            for each item A → α • Bβ, a
            and each rule B → γ and each b ∈ FIRST(β a)
            such that B → •γ, b is not in I
            do add B → •γ, b to I;
        until no more items can be added to I;
        return I;
    end
```

LR(1) PARSER CONSTRUCTION

LR(1) PARSING TABLE CONSTRUCTION

```
function goto(I,X)
    begin
        goto_items := φ ;
        for  each item A → α • Xβ, a in I,
             such that A → αX • β,a is not in goto_items
             do add A → αX • β,a to goto_items
        goto := closure(goto_items);
        return goto;
    end
```

- **The application of this algorithm to construct a canonical collection of sets of LR(1) items is illustrated in the following.**

  1. **Consider the grammar given in Figure 3.22. The first collection, i.e., state I_0, is given by $closure(S' \rightarrow \bullet S, \$)$.**
  2. **This causes the LR(1) items $S \rightarrow \bullet R, \$$ and $S \rightarrow \bullet L = R, \$$ to be added to I_0.**
  3. **Closure of the item $S \rightarrow \bullet R, \$$ causes the addition of the item $R \rightarrow \bullet L, \$$ whose closure in turn adds the items $L \rightarrow \bullet * R, \$$ and $L \rightarrow \bullet$ id,$ to I_0.**

LR(1) PARSER CONSTRUCTION

LR(1) PARSING TABLE CONSTRUCTION

4. **Closure of the item $S \rightarrow \bullet L = R, \$$ leads to the inclusion of items $L \rightarrow \bullet * R, =$ and $L \rightarrow \bullet$ `id`, = . The lookaheads for these items are given by FIRST(= R $).**

5. **Since some LR(1) items added by steps 3 and 4 above have the same core, but different lookaheads, we combine the lookaheads to write them in a compact form, $L \rightarrow \bullet * R, \$ =$ and and $L \rightarrow \bullet$ `id`, $ =**

6. **Collection $I_0 = \{S' \rightarrow \bullet S, \$$; $S \rightarrow \bullet R, \$$;**
   $S \rightarrow \bullet L = R, \$$; $L \rightarrow \bullet * R, \$ =$; $L \rightarrow \bullet$ `id`, $ = ;
   $R \rightarrow \bullet L, \$$ }

7. **The *goto* function on I_0 is defined for the symbols S, L, R, `id` and *. The resulting states are obtained by taking closures of the kernel items as given below.** $I_1 = goto(I_0, S) = closure(S' \rightarrow S\bullet, \$)$
   $I_2 = goto(I_0, L) = closure(S \rightarrow L\bullet = R, \$; R \rightarrow L\bullet, \$)$
   $I_3 = goto(I_0, R) = closure(S \rightarrow R\bullet, \$)$
   $I_4 = goto(I_0, *) = closure(L \rightarrow * \bullet R, \$ =)$
   $I_5 = goto(I_0$,`id`) $= closure(L \rightarrow$ `id`$\bullet$, $ =)$
   The rest of the collection can be constructed on the same lines.

The canonical collection $C = \{I_0, I_1, \ldots, I_n\}$ is used to construct the Action and Goto part of the table.

LR(1) PARSER CONSTRUCTION

LR(1) PARSING TABLE CONSTRUCTION

The set I_i and its constituent items define the entries for state i of the Action table as follows

1. **if $A \rightarrow \alpha\bullet$ a β, b $\in I_i$ and $goto(I_i$, a$) = I_j$**
   Action[i,a] = shift j
2. **$A \rightarrow \alpha\bullet$,a $\in I_i$**
   Action[i,a] = reduce by $A \rightarrow \alpha$
3. **$S' \rightarrow S\bullet, \$ \in I_i$**
   Action[i,$] = accept
4. **All remaining entries of state i are marked as error.**
5. **The Goto entries of the state i are**
   $goto(I_i, A) = I_j$ leads to Goto[i,A] = j
   The remaining entries are marked as error.

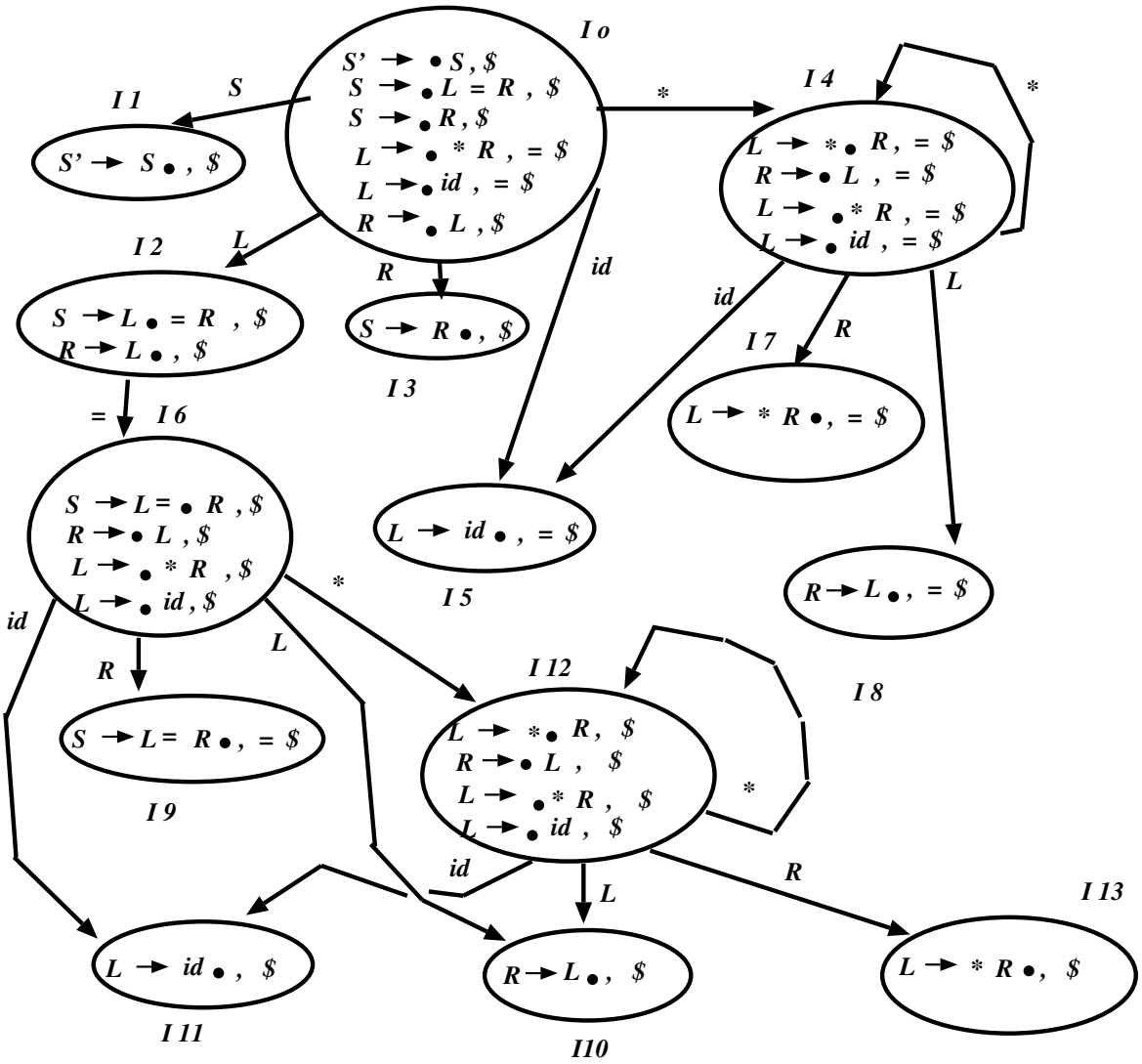
LR(1) automaton for the grammar of Figure 3.22 is given in Figure 3.24. The LR(1) parsing table is given in Figure 3.25.

LR(1) GRAMMAR AND PARSER

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

LR(1) PARSER CONSTRUCTION

FIGURE 3.24 : LR(1) AUTOMATON FOR GRAMMAR OF FIGURE 3.22



LR(1) PARSER CONSTRUCTION

FIGURE 3.25 : LR(1) PARSING TABLE FOR GRAMMAR OF FIGURE 3.22

	Action				*Goto*		
state	id	*	=	$	*S*	*L*	*R*
0	s5	s4	–	–	1	2	3
1	–	–	–	acc	–	–	–
2	–	–	s6	r5	–	–	–
3	–	–	–	r2	–	–	–
4	s5	s4	–	–	–	8	7
5	–	–	r4	r4	–	–	–
6	s11	s12	–	–	–	10	9
7	–	–	r3	r3	–	–	–
8	–	–	r5	r5	–	–	–
9	–	–	–	r1	–	–	–
10	–	–	–	r5	–	–	–
11	–	–	–	r4	–	–	–
12	s11	s12	–	–	–	10	13
13	–	–	–	r3	–	–	–

LALR(1) PARSER CONSTRUCTION

LALR(1) PARSER

- **This parser is of intermediate capability as compared to SLR(1) and LR(1). The parser has the size advantage of SLR and lookahead capability like LR.**
- **In a LR parser, several states may have the same first components (LR(0) items) but differ in the lookaheads associated and hence are represented as distinct states.**
- **In terms of the first component only, both SLR and LR define the same collection of LR(0) items.**
- **LALR automaton is created by merging states of LR automaton that have the same core (set of first components). The merge results in the union of the lookahead symbols of the respective items.**

LALR(1) PARSING TABLE CONSTRUCTION

1. **Construct the collection $C = \{I_0, I_1, \ldots, , I_n\}$ of LR(1) items.**
2. **For each core among the set of LR(1) items, find all sets having the same core and replace these sets by their union.**
3. **From the reduced collection, say, $C' = \{J_0, J_1, \ldots, J_m\}$, the Action table is constructed the same way.**

LALR(1) PARSER CONSTRUCTION

LALR(1) PARSER

4. **To compute Goto[J,X], let $J = \{I_1 \cup I_2 \cup \ldots \cup I_k$ }. Find K which is the union of all the LR(1) sets that have the same core as $goto(I_1, X)$. Then Goto[J,X] = K.**

To illustrate the construction of a LALR automaton, consider the LR(1) automaton given in Figure 3.24.

1. **This automaton has the following states whose cores are identical. States I_4 and I_{12}, states I_5 and I_{11}, states I_7 and I_{13}, and states I_8 and I_{10}.**

2. **The merger of the states given above results in $I_4 = I_4 \cup I_{12}$, $I_5 = I_5 \cup I_{11}$, $I_7 = I_7 \cup I_{13}$ and $I_8 = I_8 \cup I_{10}$. This happens since in each case the lookahead sets are proper subsets.**

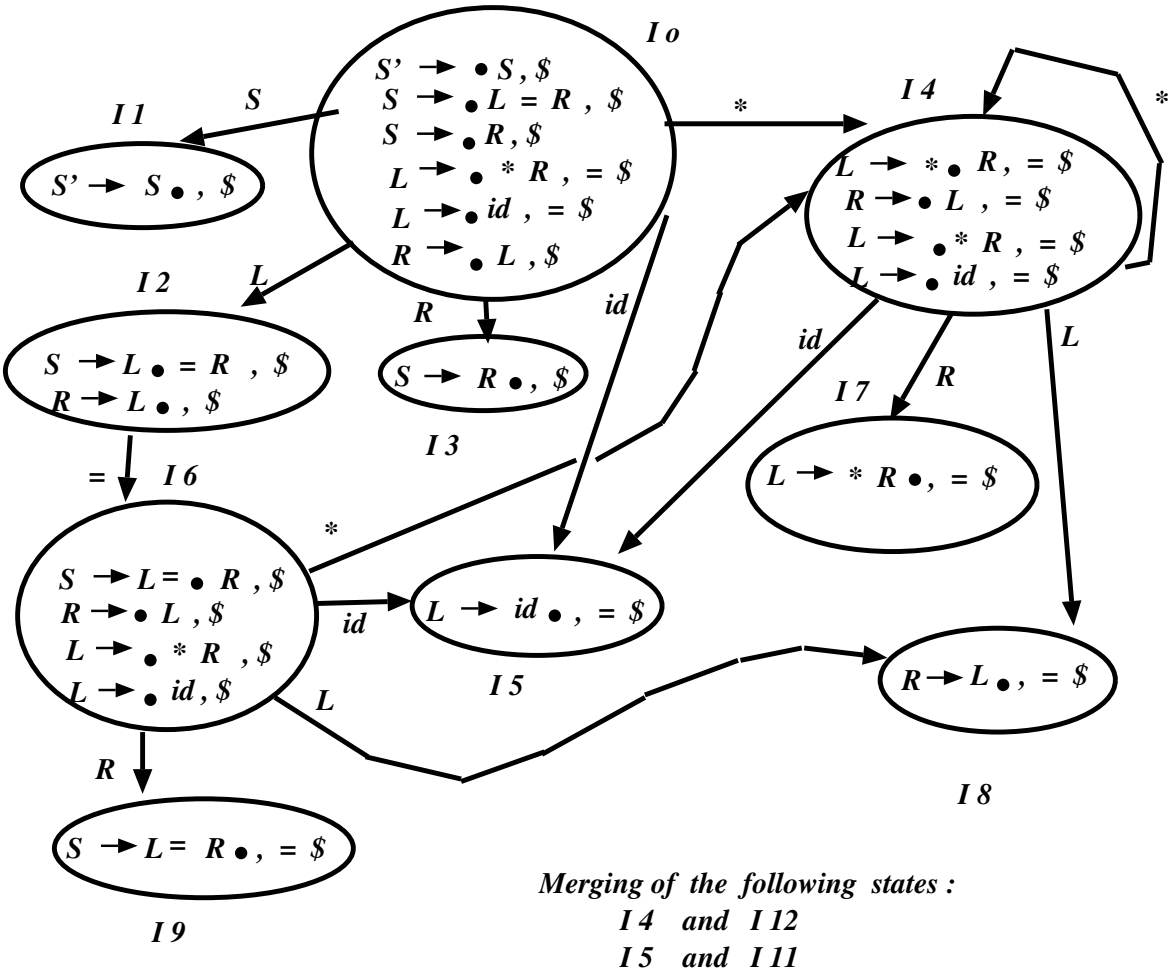
The LALR automaton and the parsing table for the grammar of Figure 3.22 are given in the Figures 3.26 and 3.27 respectively.

LALR(1) GRAMMAR AND PARSER

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

LALR(1) PARSER CONSTRUCTION

FIGURE 3.26 : LALR(1) AUTOMATON FOR GRAMMAR OF FIGURE 3.22



Merging of the following states :

I 4 and I 12
I 5 and I 11
I 7 and I 13
I 8 and I 10

LALR(1) PARSER CONSTRUCTION

FIGURE 3.27 : LALR(1) PARSING TABLE FOR GRAMMAR OF FIGURE 3.22

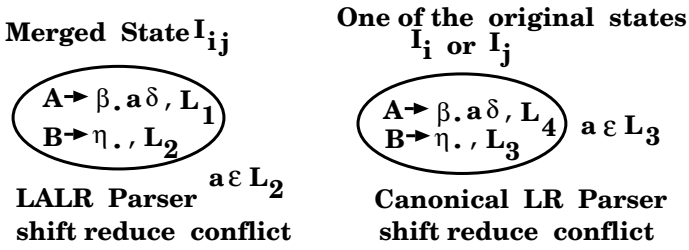
Grammar : $S' \rightarrow S$
$S \rightarrow L = R \mid R$
$L \rightarrow * R \mid id$
$R \rightarrow L$

	Action				***Goto***		
state	**id**	*****	**=**	**$**	***S***	***L***	***R***
0	**s5**	**s4**	**–**	**–**	**1**	**2**	**3**
1	**–**	**–**	**–**	**acc**	**–**	**–**	**–**
2	**–**	**–**	**s6**	**r5**	**–**	**–**	**–**
3	**–**	**–**	**–**	**r2**	**–**	**–**	**–**
4	**s5**	**s4**	**–**	**–**	**–**	**8**	**7**
5	**–**	**–**	**r4**	**r4**	**–**	**–**	**–**
6	**s5**	**s4**	**–**	**–**	**–**	**8**	**9**
7	**–**	**–**	**r3**	**r3**	**–**	**–**	**–**
8	**–**	**–**	**r5**	**r5**	**–**	**–**	**–**
9	**–**	**–**	**–**	**r1**	**–**	**–**	**–**

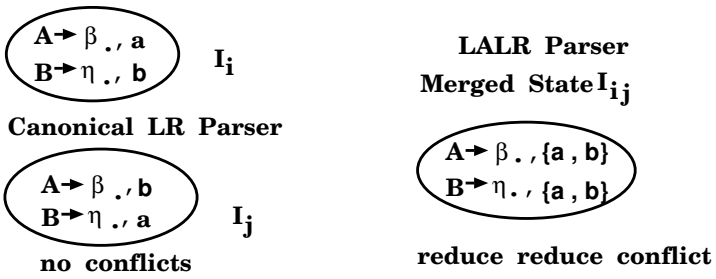
LALR(1) PARSER CONSTRUCTION

RESULTS IN LALR(1) PARSING

- A theorem states that a LALR(1) parsing table is always free of shift reduce conflicts, provided the corresponding LR(1) table is so.
  The key observation towards proving this fact is given in the following.



- However the theorem does not exclude the possibility of a LALR(1) parsing table having reduce reduce conflicts while the corresponding LR(1) table is conflict free as shown below.



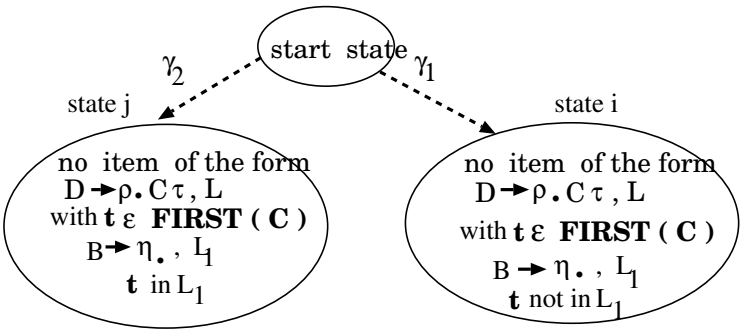
LALR(1) PARSER CONSTRUCTION

RESULTS IN LALR(1) PARSING

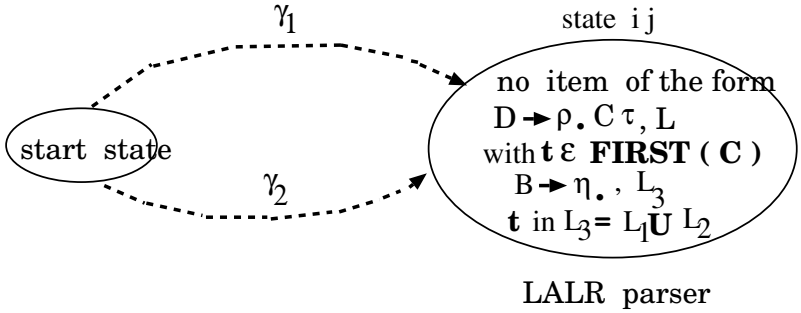
- Another important result relates the behaviour of these two parsers.
- The behaviour of both LALR and LR parsers are identical (same sequence of shifts and reduces) for a sentence of the language.
- However, for an erroneous input, a LALR parser may continue to perform reductions after the LR parser has caught the error.
- An interesting fact is that a LALR parser does not shift any symbol beyond the point that the LR parser declared an error.
- Use the parsers of Figures 3.25 and 3.27 to parse the input `id = *id = $` and note the difference in behaviour.
- The reasons are analyzed in the following. The discussion can be converted into a proof for this result if LALR has only one merged state.
- Let a LR(1) parser be in state i with `t` as the next symbol. This state recognizes a viable prefix γ_1. In the figure given below, state i indicates error on `t` since the first form of items do not permit `t` to be shifted and the other prevents any reduce on `t`.

LALR(1) PARSER CONSTRUCTION

Behaviour of LR and LALR Parsers on Erroneous Inputs



- **State i of the corresponding LALR parser is shown below. This state after recognizing γ_1 permits a reduce on t because of the state j of the LR parser.**

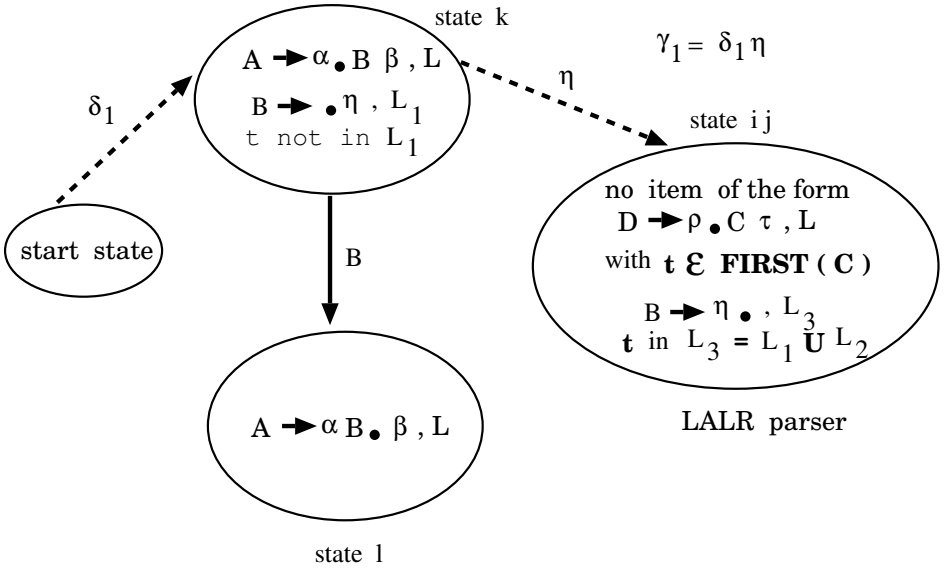


LALR parser

LALR(1) PARSER CONSTRUCTION

Behaviour of LR and LALR Parsers on Erroneous Inputs

- **After reducing by $B \rightarrow \eta\bullet$, the LALR parser would be in state l whose relevant contents are shown below. It is assumed that states k and l of both the parsers are identical.**



- **Presence of the item $A \rightarrow \alpha\bullet B\beta, L$ in state k is justified since $B \rightarrow \bullet\eta, L_1$ is an item which can only be added through closure. Terminal t must not be in FIRST(β) since otherwise t $\in L_1$ which in turn shows that state l does not shift t.**

LR PARSERS

COMPARISON OF THE LR PARSERS

1. **SLR parser is the one with the smallest size and easiest to construct. It uses FOLLOW information to guide reductions.**
2. **Canonical LR parser has the largest size among the other parsers of this family. The lookaheads are associated with the items and they make use of the left context available to the parser.**
3. **LR grammar is a larger subclass of context free grammars as compared to that of SLR and LALR grammars.**
4. **LALR parser has the same size as that of a SLR parser but is applicable to a wider class of grammars than SLR grammar.**
5. **LALR is less powerful as compared to LR, however most of the syntactic features of programming languages can be expressed in this grammar.**
6. **LALR parser for an erroneous input behaves differently than a LR parser. Error detection capability is immediate in LR but not so in LALR.**
7. **Both LR and LALR parsers are expensive to construct in terms of time and space. However, efficient methods exist for constructing LALR parsers directly.**

LR PARSERS

AUTOMATIC GENERATION OF THE LR PARSERS

- **All the parsers of this family can be generated automatically given a context free grammar as input. The parser generation tool YACC available in UNIX uses such a strategy for generating a LALR parser.**
- **The basic steps to automate the process for generating a SLR parser from a grammar G are given below.**
  1. **Augment the given grammar G.**
  2. **Construct FOLLOW information for all the nonterminals of G.**
  3. **Use the algorithms for Closure , goto and sets-of-items construction to construct the canonical collection of LR(0) sets of items.**
  4. **Use the procedure for constructing a parsing table from the canonical collection to create the table. If the table contains conflicts, report them.**
  5. **Supply a driver routine which uses a stack, an input buffer and the parsing table for actually parsing an input.**
- **The other parsers can also be generated along similar lines.**