UPES

UNIVERSITY WITH A PURPOSE

Content of this lecture

- Bottom up parser

Bottom Up Parser

- Bottom up Parser
  - Constructs 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.
  - Input is scanned from left to right one symbol at a time.

Bottom Up Parser

- Bottom up Parser
  - Tries to find the right-most derivation of the given input in the reverse order
    - S ⇒ ... ⇒ ω (the right-most derivation of ω)
  - Bottom-up parsing is also known as shift-reduce parsing because its two main actions are shift and reduce.
    - At each **shift** action, the current symbol in the input string is pushed to a stack.
    - At each **reduction** step, the symbols at the top of the stack (this symbol sequence is the right side of a production) will replaced by the non-terminal at the left side of that production.
    - Two more actions: accept and error

UPES
UNIVERSITY WITH A PURPOSE

Bottom Up Parser

- Shift Reduce Parser
  - Shift-reduce parser tries to reduce the given input string into the starting symbol
    - a string →(reduced to) the starting symbol
  - At each reduction step, a substring of the input matching to the right side of a production rule is replaced by the non-terminal at the left side of that production rule
  - If the substring is chosen correctly, the right most derivation of that string is created in the reverse order
    - Rightmost Derivation: $S \Rightarrow \omega$
    - Shift-Reduce Parser finds: $\omega \Leftarrow \ldots \Leftarrow S$

Shift Reduce Parsing Example

Consider the grammar

S → aABe

A → Abc | b

B → d

Input string : abbcde

aAbcde

aAde ⇓ reduction

aABe

S

- We can scan abbcde looking for a substring that matches the right side of some production. The substrings b and d qualify. Let us choose left most b and replace it by A, the left side of the production A→b; we thus obtain the string aAbcde.
- Now the substrings Abc, b and d match the right side of some production. Although b is the leftmost substring that matches the right side of the some production, we choose to replace the substring Abc by A, the left side of the production A→Abc. We obtain aAde.
- Then replacing d by B and then replacing the entire string by S.
- Thus, by a sequence of four reductions we are able to reduce abbcde to S

UPES
UNIVERSITY WITH A PURPOSE

Shift Reduce Parsing Example

$S \rightarrow aABb$

$A \rightarrow aA \mid a$

$B \rightarrow bB \mid b$

input string: aaabb

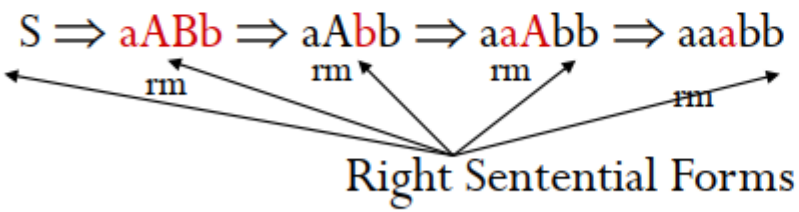
aaAbb

aAbb

⇓ reduction

aABb

S



- How do we know which substring to be replaced at each reduction step?

Sentential Form

- A left-sentential form is a sentential form that occurs in the leftmost derivation of some sentence.
- A right-sentential form is a sentential form that occurs in the rightmost derivation of some sentence.

Handle

- **Handle** of a string is a substring that matches the right side of a production rule.
  - *Not every substring that matches the right side of a production rule is handle*

- A **handle** of a right sentential form γ (≡ αβω) :
  - Is a production rule A → β 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
    - $S \Rightarrow \alpha A \omega \Rightarrow \alpha \beta \omega$
    - ω *is a string of terminals*

- If the grammar is *unambiguous*, then every right-sentential form of the grammar has exactly one handle

UPES
UNIVERSITY WITH A PURPOSE

Handle Pruning

- A right-most derivation in reverse can be obtained by **handle-pruning**

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

input string

- Start from γ_n, find a handle $A_n \rightarrow \beta_n$ in γ_n, and replace β_n by A_n to get γ_{n-1}
- Then find a handle $A_{n-1} \rightarrow \beta_{n-1}$ in γ_{n-1}, and replace β_{n-1} by A_{n-1} to get γ_{n-2}
- Repeat this, until we reach S

UPES
UNIVERSITY WITH A PURPOSE

Shift Reduce Parser

$E \rightarrow E+T \mid T$

$T \rightarrow T*F \mid F$

$F \rightarrow (E) \mid id$

Right-Most Derivation of `id + id*id`

$E \Rightarrow E+T \Rightarrow E+T*F \Rightarrow E+T*id \Rightarrow E+F*id$

$\Rightarrow E+id*id \Rightarrow T+id*id \Rightarrow F+id*id \Rightarrow id+id*id$

Right-Most Sentential Form	Reducing Production
id+id*id	$F \rightarrow id$
F+id*id	$T \rightarrow F$
T+id*id	$E \rightarrow T$
E+id*id	$F \rightarrow id$
E+F*id	$T \rightarrow F$
E+T*id	$F \rightarrow id$
E+T*F	$T \rightarrow T*F$
E+T	$E \rightarrow E+T$
E	

Handles are red and underlined in the right-sentential forms.

UPES
UNIVERSITY WITH A PURPOSE

Stack Implementation of Shift Reduce Parser

- Four possible actions of a shift-parser :

  1. **Shift** :The next input symbol is shifted onto the top of the stack

  2. **Reduce**: Replace the handle on the top of the stack by the non-terminal

  3. **Accept**: Successful completion of parsing

  4. **Error**: Parser discovers a syntax error, and calls an error recovery routine

- Initial stack: contains only the end-marker $

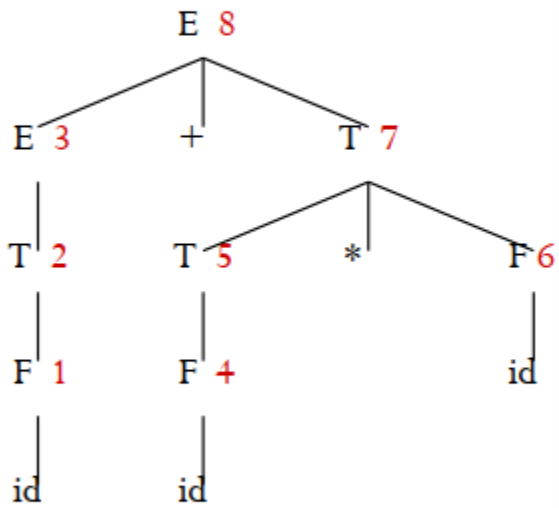
- End of the input string: marked by the end-marker $

UPES
UNIVERSITY WITH A PURPOSE

Stack Implementation of Shift Reduce Parser

Stack	Input	Action
$	id+id*id $	shift
$id	+id*id$	reduce by F → id
$F	+id*id$	reduce by T → F
$T	+id*id$	reduce by E → T
$E	+id*id$	shift
$E+	id*id$	shift
$E+id	*id$	reduce by F → id
$E+F	*id$	reduce by T → F
$E+T	*id$	shift
$E+T*	id$	shift
$E+T*id	$	reduce by F → id
$E+T*F	$	reduce by T → T*F
$E+T	$	reduce by E → E+T
$E	$	accept

Parse Tree

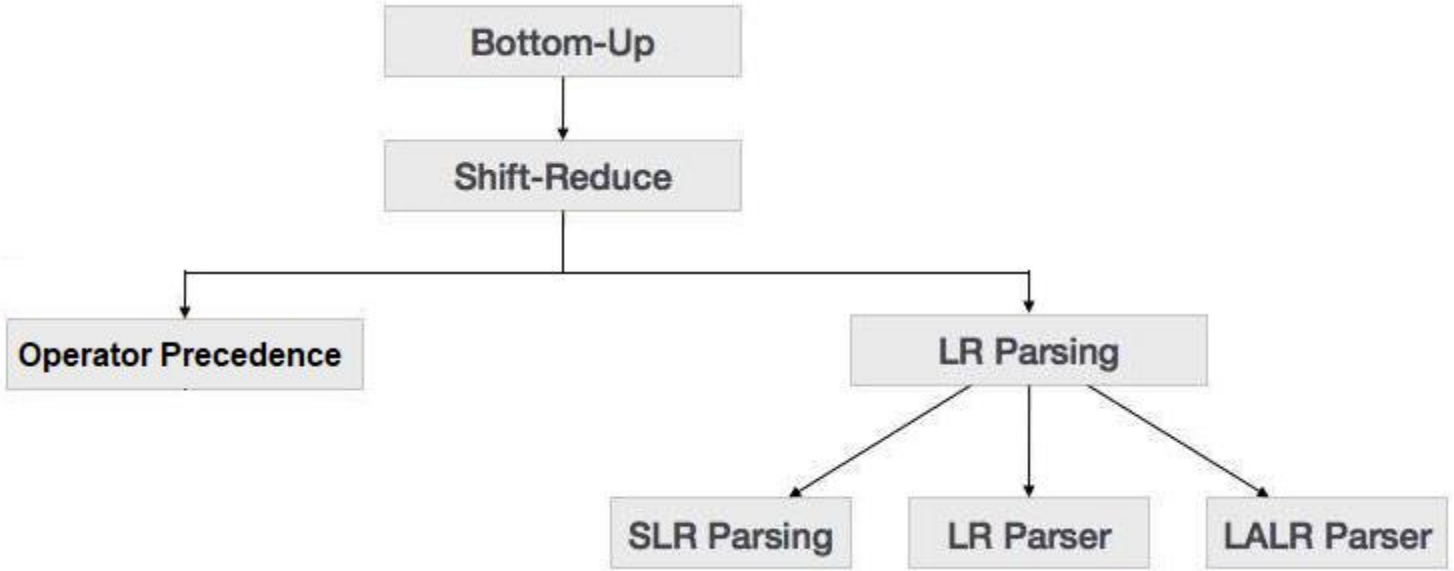


Conflict During Shift Reduce Parser

- For certain class of CFGs, shift-reduce parsers cannot be used
- Stack contents and the next input symbol may not decide action:
  - **shift/reduce conflict**: Whether make a shift operation or a reduction
  - **reduce/reduce conflict**: The parser cannot decide which of several reductions to make
- If a shift-reduce parser cannot be used for a grammar, that grammar is called as non-LR(k) grammar, here L stands for Left to right scanning, R stands for right most derivation and k is look ahead.
- An ambiguous grammar can never be a LR grammar.

UPES
UNIVERSITY WITH A PURPOSE

Types of Shift Reduce Parser



Operator Grammar

- A grammar G is said to be an Operator Grammar, if the following conditions are satisfied in all the production rules:
  - No Ɛ-transition.
  - No two adjacent non-terminals.
  - There exists unique relationship between each pair or terminals or no relationship.

E.g.

E → E op E | id

op → + | *

The above grammar is not an operator grammar but:

E → E + E | E* E | id

UPES
UNIVERSITY WITH A PURPOSE

Operator Precedence Parsing

- OPP is especially suitable for parsing expressions since it can use information about the precedence and associativity of operator to guide the parser.
- We can’t use OPP to make a parser of entire language. It is only used to make a parser for expression of language.
- Due to its simplicity and easy to implement, this technique is used in compiler for parsing of expressions only.
- Tokens like ‘-’, which has two different precedence are hard to handle.

Operator Precedence Parsing

- This parser relies on the following three precedence relations:
  - If a has higher precedence over b then a .> b
  - If a has lower precedence over b then a <. b
  - If a and b have equal precedence then a =. b
- Note:
  - id has higher precedence than any other symbol and no relationship exists between two different identifiers.
  - $ has lowest precedence.
  - if two operators have equal precedence, then we check the Associativity of that particular operator.

Operator Precedence Parsing

- Precedence Table

	id	+	*	$
id		.>	.>	.>
+	<.	.>	<.	.>
*	<.	.>	.>	.>
$	<.	<.	<.	.>

- Example:
  - The input string:

    id1 + id2 * id3
  - After inserting precedence relations becomes:

    $ <· id1 ·> + <· id2 ·> * <· id3 ·> $

UPES
UNIVERSITY WITH A PURPOSE

Rules for finding precedence relationship without using Precedence & Associativity Rules

- Leading:
  - If production is of the form A→aα or A→Baα then Leading(A)={a}
  - If production is of the form A→Bα and if a is in Leading(B) then a will also be in Leading(A).

- Trailing:
  - If production is of the form A→αa or A→αaB then Trailing(A)={a}
  - If production is of the form A→αB and if a is in Trailing(B) then a will also be in Trailing(A).

UPES
UNIVERSITY WITH A PURPOSE

Rules for finding precedence relationship without using Precedence & Associativity Rules

- Compute Leading and Trailing for following non terminals:
- E→E+T|T
- T→T*F|F
- F→(E)|id

Rules for finding precedence relationship without using Precedence & Associativity Rules

- Compute Leading and Trailing for following non terminals:
- E→E+T|T
- T→T*F|F
- F→(E)|id

- Leading(F)={(,id}
- Leading(T)={*,(,id}
- Leading(E)={+,*,(,id}

UPES
UNIVERSITY WITH A PURPOSE

Rules for finding precedence relationship without using Precedence & Associativity Rules

- Compute Leading and Trailing for following non terminals:
- E→E+T|T
- T→T*F|F
- F→(E)|id

- Leading(F)={(,id} Trailing(F)={),id}
- Leading(T)={*,(,id} Trailing(T)={*,),id}
- Leading(E)={+,*,(,id} Trailing(E)={+,*,),id]

UPES
UNIVERSITY WITH A PURPOSE

Rules for finding precedence relationship without using Precedence & Associativity Rules

- a=b if RHS of production is of the form αaβbγ where β can be epsilon or non terminal and α, γ are any combination of terminal or non terminals. E.g. S→mAcBed implies that m=c and c=e
- a<b if RHS of production is of the form αaAβ and A→γb where γ is either epsilon or single non terminal. E.g. S→mAcD and A→i then m=c and m<i
- a>b if RHS of production is of the form αAbβ and A→ γa where γ is either epsilon or single non terminal. E.g. S→mAcD and A→i then m=c and m<i and i>c

UPES
UNIVERSITY WITH A PURPOSE

Operator Precedence Parsing

- Basic Principle
  - Scan input string left to right, try to detect .> and put a pointer on its location.
  - Now scan backwards till reaching <.
  - String between <. And .> is our **handle**.
  - Replace handle by the head of the respective production.
  - REPEAT until reaching start symbol.

Operator Precedence Parsing

$E \rightarrow E + E \mid E * E \mid (E) \mid id.$

Find handles at each step for reducing the string $id_1 + id_2 * id_3$ *using operator Precedence Parsing.*

Ans. First, Attach $ at starting and ending of string *i.e.* $ $id_1 + id_2 * id_3$ $.

Put Precedence relation between operators & symbols using Precedence Relation Table.

$\therefore \quad \$ \lessdot id_1 \gtrdot + \lessdot id_2 \gtrdot * \lessdot id_3 \gtrdot \$$

Apply 3 steps as explained above on this string.

String	Handle	Production used
$\lessdot id_1 \gtrdot + \lessdot id_2 \gtrdot * \lessdot id_3 \gtrdot \$$	$\lessdot id_1 \gtrdot$	$E \rightarrow id$
$\$ E + \lessdot id_2 \gtrdot * \lessdot id_3 \gtrdot \$$	$\lessdot id_2 \gtrdot$	$E \rightarrow id$
$\$ E + E * \lessdot id_3 \gtrdot \$$	$\lessdot id_3 \gtrdot$	$E \rightarrow id$
$\$ E + E * E \$$	Remove all Non-Terminals	
$\$ + * \$$	Insert precedence relation between operators	
$\$ \lessdot + \lessdot * \gtrdot \$$	$\lessdot * \gtrdot$ *i.e.* $E * E$	$E \rightarrow E * E$
$\$ \lessdot + \gtrdot \$$	$\lessdot + \gtrdot$ *i.e.*, $E + E$	$E \rightarrow E + E$
$		

Algorithm for Operator Precedence Relationship

- For each production of the form $A \rightarrow B_1B_2....B_n$
- For i=1 to n-1 do the following
  - If B_i and $B_{i+1}$are both terminals then set $B_i=B_{i+1}$
  - If i<=n-2 and B_i and $B_{i+2}$are both terminals and B_{i+1} is non terminals then set $B_i=B_{i+2}$
  - If B_i is terminal and $B_{i+1}$is non terminal then for all a in Leading of B_{i+1} B_i <a
  - If B_i is non terminal and $B_{i+1}$is terminal then for all a in Trailing of B_i a > B_{i+1}

Operator Precedence Parsing Algorithm

- Algorithm

```
w ← String with $ symbol at the end
a ← Present input symbol in w
b ← Symbol on stack top
Repeat
{
        if(a is $ and b is $)
                return
        if(a .> b or a=.b)
        {
                shift a into stack
                move input pointer one step rightward
        }
        else if(a <. b)
        {
                repeat
                {
                        c ← pop stack
                }until(c .> b)
        }
        else
                error()
}
```

Operator Precedence Parsing

- Example

STACK	INPUT	ACTION/REMARK
$	id + id * id$	$ <. Id
$ id	+ id * id$	id >. +
$	+ id * id$	$ <. +
$ +	id * id$	+ <. Id
$ + id	* id$	id .> *
$ +	* id$	+ <. *
$ + *	id$	* <. Id
$ + * id	$	id .> $
$ + *	$	* .> $
$ +	$	+ .> $
$	$	accept

UPES
UNIVERSITY WITH A PURPOSE

Operator Precedence Parsing

- Precedence Function
  - Precedence relations between any two operators or symbols in precedence table can be converted to two precedence functions f and g that map terminals symbols to integers.
    - If a<.b then f(a)<.g(b)
    - If a=b then f(a)=g(b)
    - If a.>b then f(a).>g(b)

  - Here a and b are terminal symbols, f(a) and g(b) represents the precedence functions which have integer values

Operator Precedence Parsing

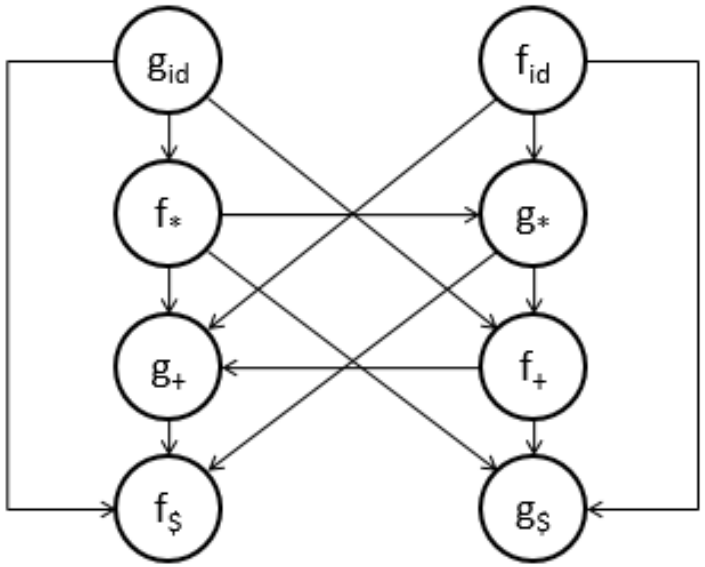
- Computation of Precedence Function
  - For each terminal a, create symbol f_a and g_b
  - Make node for each symbol
    - If a=b then f_a and g_b are in the same group.
    - If a<b then mark an edge from g_b to f_a
    - If a>b then mark an edge from f_a to g_b
  - If graph constructed has cycle then no precedence function exists.
  - If there are no cycles then
  - f_a=Length of longest path beginning at group of f_a
  - g_a=Length of longest path beginning at group of g_a

UPES
UNIVERSITY WITH A PURPOSE

Operator Precedence Parsing

- Precedence Function

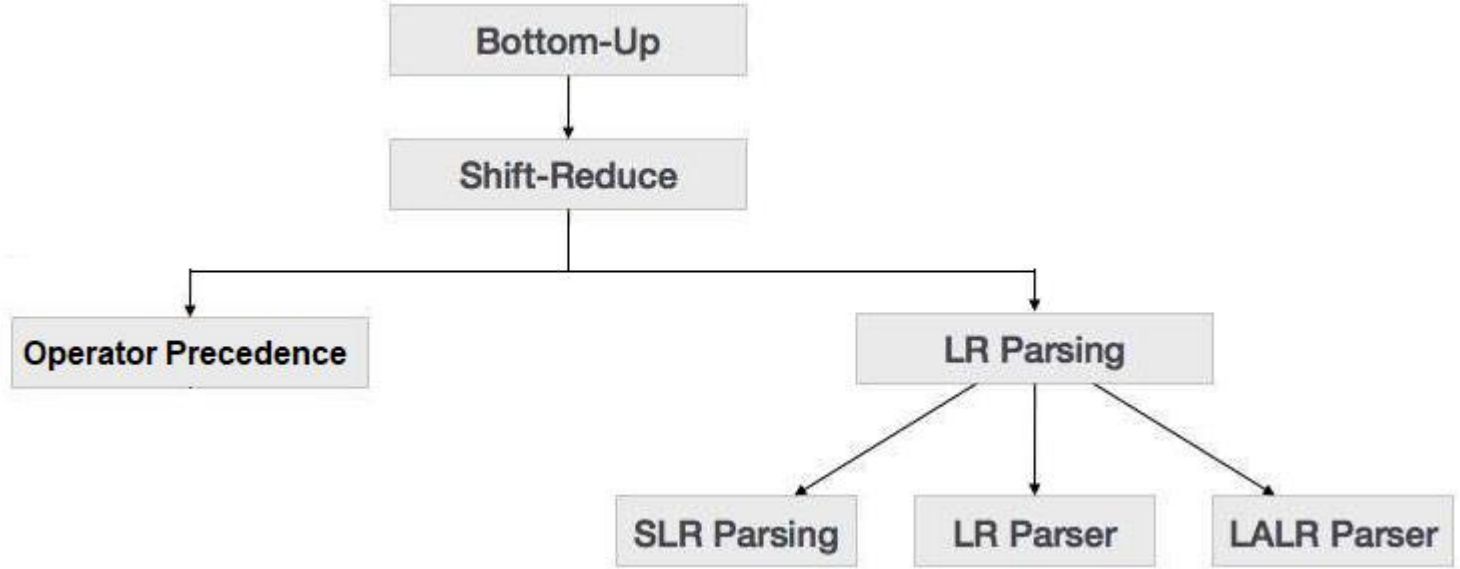
	id	+	*	$
id		.>	.>	.>
+	<.	.>	<.	.>
*	<.	.>	.>	.>
$	<.	<.	<.	.>



	id	+	*	$
f	4	2	4	0
g	5	1	3	0

LR Parser

- LR parser is a class of parser in which parser work in bottom up manner.
- Parser of this class scans the input from left to right and construct a rightmost derivation in reverse.



UPES
UNIVERSITY WITH A PURPOSE

LR Parser

- Advantages:
  - Can recognize all programming language for which context free grammar exists
  - More general than operator precedence parser.
  - Can detect syntactic error

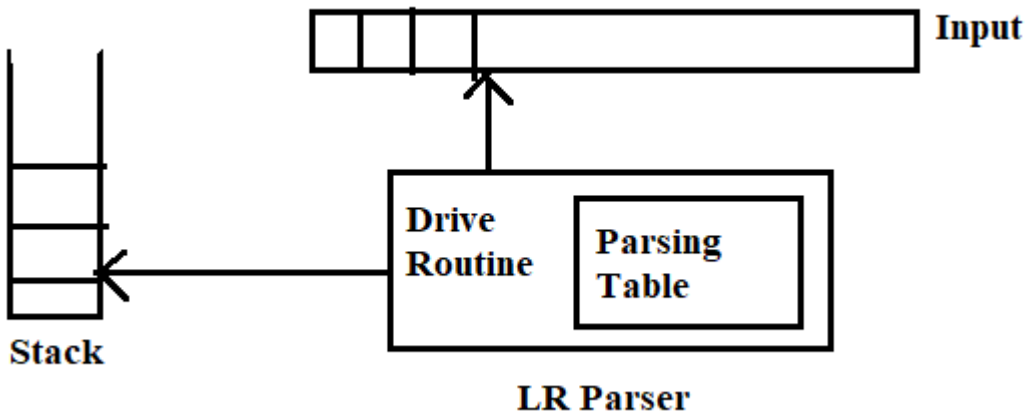
- Disadvantages:
  - Its implementation requires too much manual work.
  - Its implementation also requires a specialized parser generator tool. Using such tool, we can write context free grammar and automatically produce the parser for that grammar.

LR Parser

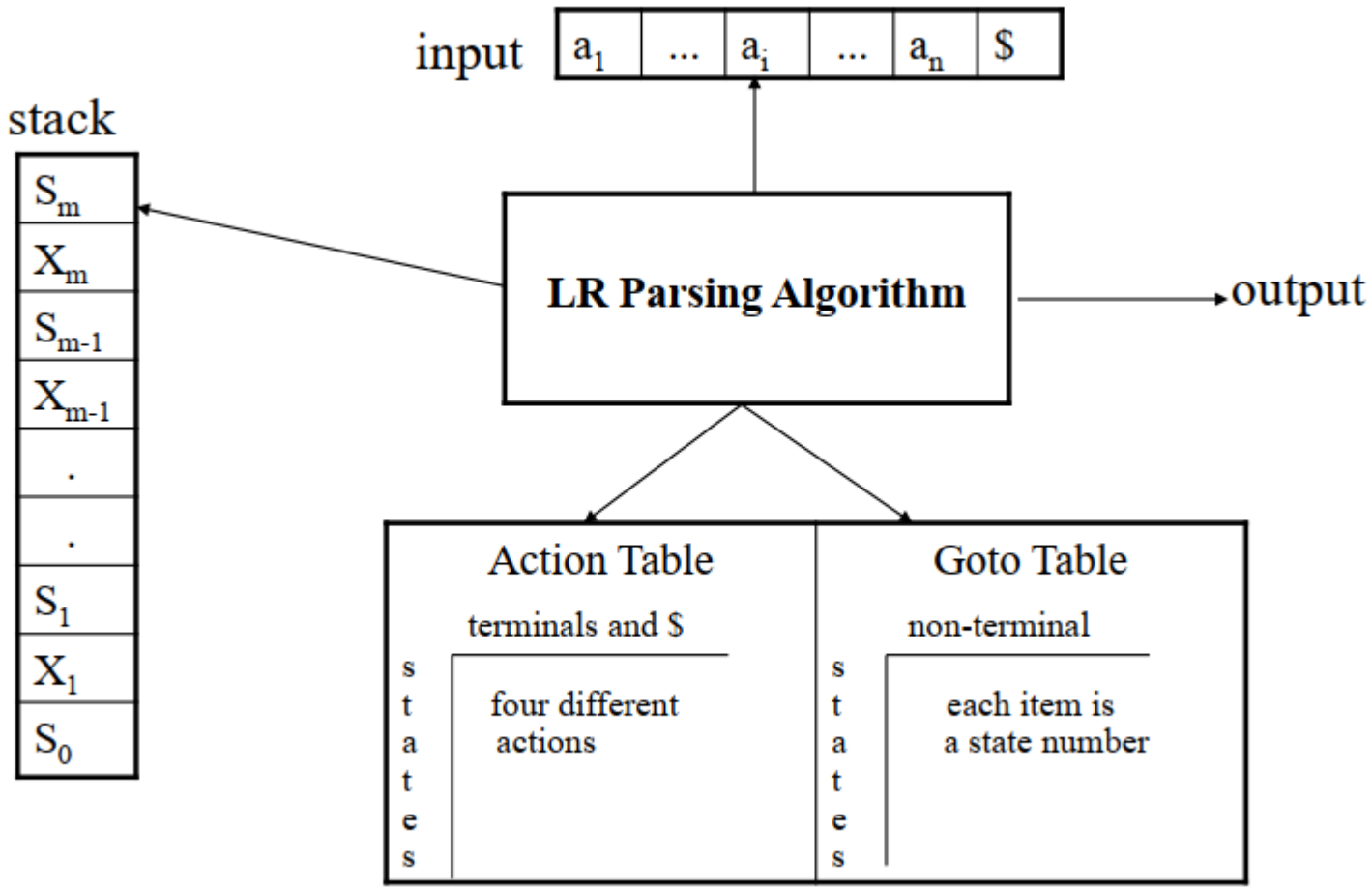
- There are three variants of LR parser:
  - Simple LR (SLR):
    - It is easiest to implement.
    - It fails to produce parsing table for some grammar.
    - We use canonical collection of LR(0) items.
  - Look ahead LR (LALR):
    - Its power is intermediate among SLR and CLR.
    - With some effort, can be implemented efficiently.
    - We use canonical collection of LR(1) items.
    - Work on most programming language grammar.
  - Canonical LR (CLR):
    - It is most power full.
    - It is expensive to implement.
    - We use canonical collection of LR(1) items.
    - It can generate parsing table for large class of grammars

LR Parser

- Every LR parser has the following components:
  - Input buffer: Content from this buffer is read from left to right one symbol at a time
  - Stack: Stack consists of a string of the form $s_0X_1s_1X_2s_2X_3$……. $s_mX_{m+1}s_{m+1}X_{m+2}$ s_{m+2}……, s_0 is at the bottom of the stack
    - s_i= symbol called state which summarize the information contained in the stack below it and it is used for shift reduce decision.
    - X_i is a grammar symbol (terminal or non terminal)
  - Parsing Table: It consists of two parts:
    - A parsing action function ACTION
      Its arguments are state and terminal.
    - A parsing goto function called GOTO
      Its arguments are state and non terminal.

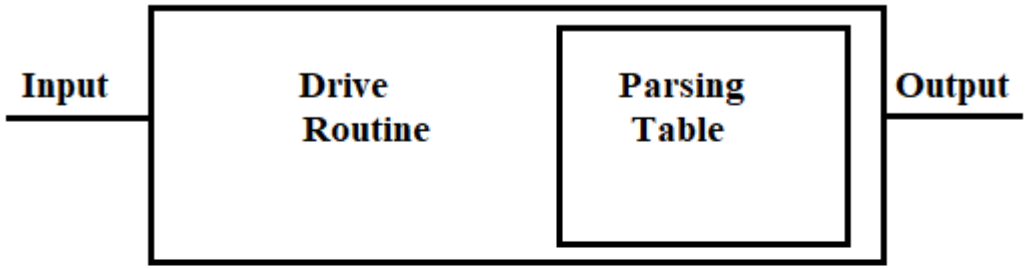


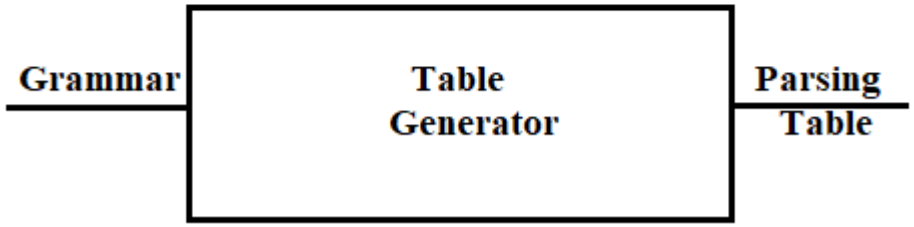
LR Parser


input
a_1 ... a_i ... a_n $
stack
S_m
X_m
S_{m-1}
X_{m-1}
.
.
S_1
X_1
S_0
LR Parsing Algorithm
output
Action Table
terminals and $
states
four different actions
Goto Table
non-terminal
states
each item is a state number

LR Parser

- Every LR parser consists of two parts:
  - Driver Routine: It is same for all variants of LR parser. It is simple to implement. Usually it is a finite automata.
  - Parsing table: It is different for different variants of LR parser. Also, it is difficult to implement.

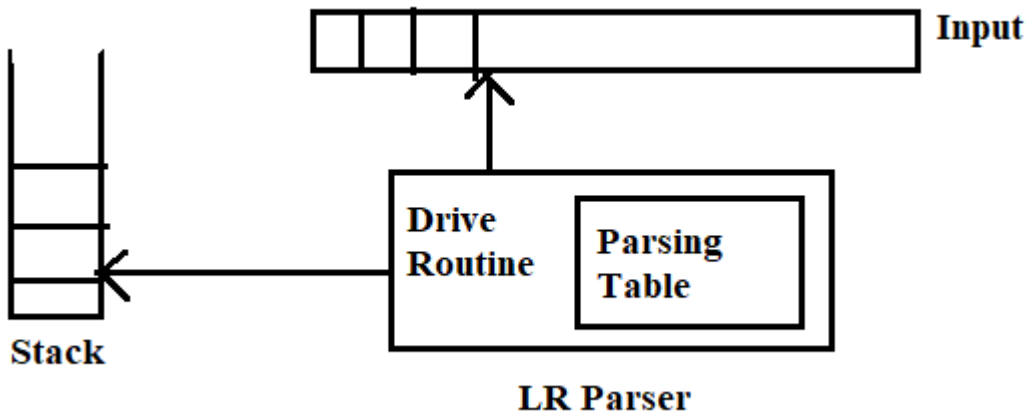




UPES
UNIVERSITY WITH A PURPOSE

LR Parser

- Working of LR parser:
  1. Determine the state currently on the top of the stack and current input symbol.
  2. Consult the ACTION table for current state and current input symbol. Corresponding entry in the action table can have one of the 4 values:
     - Shift
     - Reduce
     - Accept
     - Error
  3. Consult GOTO table when state and grammar symbol appears successively.



Configuration of LR Parser

- Configuration of LR parser is a pair, whose first component is the stack content and second component is unexpended input.
- Initial configuration of LR parser is $(s_0, a_1a_2\ldots.a_n\$)$

- A configuration of a LR parsing is:

$$(S_o X_1 S_1 \ldots X_m S_m,\ a_i a_{i+1} \ldots a_n \$)$$

Stack — Rest of Input

- S_m and a_i decides the parser action by consulting the parsing action table (*Initial Stack* contains just S_o)
- S_o : does not represent any grammar symbol

LR & Non-LR Grammar

- A grammar is no-LR if shift reduce parser for that grammar can reach a configuration in which the parser knowing the entire stack content and next input symbol, but can't decide whether to shift or reduce or can't decide which of several reductions to make.
- A grammar for which we can construct a parsing table in which every entry is uniquely defined is said to be LR grammar. There are some context free gramma which are not LR grammar.
- An LR parser can determine everything from the state on top of stack. A grammar that can be parsed by an LR parser by examining up to k input symbol on each move is called an LR(k) grammar.

LR(0) items

- Any production rule with . symbol at some position on RHS is called items.
- LR(0) items of a grammar G are the production of G with a dot at some position of the right side. E.g. A→XYZ generates 4 items A→.XYZ A→X.YZ A→XY.Z A→XYZ.
- State is a group of items.
- A→λ generate only one item A→.
- Significance of . is that it helps us in taking decision about reduction. We do reduction when we have seem the entire symbols on RHS. An item indicates how much of a production we have seen at a point in the parsing process.

UPES
UNIVERSITY WITH A PURPOSE

Viable Prefixes

- Viable prefixes are those prefixes which are in the right sentential form that do not contain any symbol to the right of handle.

Canonical Collection of LR(0) items

- Canonical collection of LR(0) Items provides the basis for constructing a class of LR parser called Simple LR(SLR).
- To construct the canonical collection of items for a grammar, we need to define the following:
  1. Augmented Grammar
  2. Closure Function
  3. GOTO function

UPES
UNIVERSITY WITH A PURPOSE

Augmented Grammar

- If G is a grammar with start symbol S, then G` is the augmented grammar for G with a new start symbol S` and production S`→S
- E.g. if G is a given grammar with the following production rules:
  S→AA
  A→aA
  A→b
- Then the equivalent Augmented Grammar G` will be:
  S`→S
  S→AA
  A→aA
  A→b

UPES
UNIVERSITY WITH A PURPOSE

Closure Function

- If I is a set of items for a grammar G then closure(I) will be:
  - Every item in I is in closure(I)
  - If A→α.Bβ is in closure(I) and B→γ is a production then add the item B→.γ to closure(I) (Whenever there is . symbol on the left of a variable, we add all the productions of that variable with . symbol in the beginning.
  - We will apply this rule until no more new LR(0) items can be added to closure(I).

Closure Function -Example

$E' \rightarrow E$

$E \rightarrow E+T$

$E \rightarrow T$

$T \rightarrow T*F$

$T \rightarrow F$

$F \rightarrow (E)$

$F \rightarrow id$

closure($\{E' \rightarrow \bullet E\}$) =

$\{ E' \rightarrow \bullet E$

$E \rightarrow \bullet E+T$

$E \rightarrow \bullet T$

$T \rightarrow \bullet T*F$

$T \rightarrow \bullet F$

$F \rightarrow \bullet (E)$

$F \rightarrow \bullet id \ \}$

UPES
UNIVERSITY WITH A PURPOSE

GOTO Function

- I=set of items
- X=grammar symbol
- If A → α.Xβ in I then every item in closure({A → αX.β}) will be in goto (I,X)

Example:

I = { E' → • E, E → • E+T, E → • T,
T → • T*F, T → • F,
F → • (E), F → • id }

goto (I,E) = { E' → E • , E → E • +T }

goto (I,T) = { E → T • , T → T • *F }

goto (I,F) = {T → F • }

goto (I,() = { F → (• E), E → • E+T, E → • T, T → • T*F, T → • F,
F → • (E), F → • id }

goto (I,id) = { F → id • }

Construction of Canonical LR(0) items construction

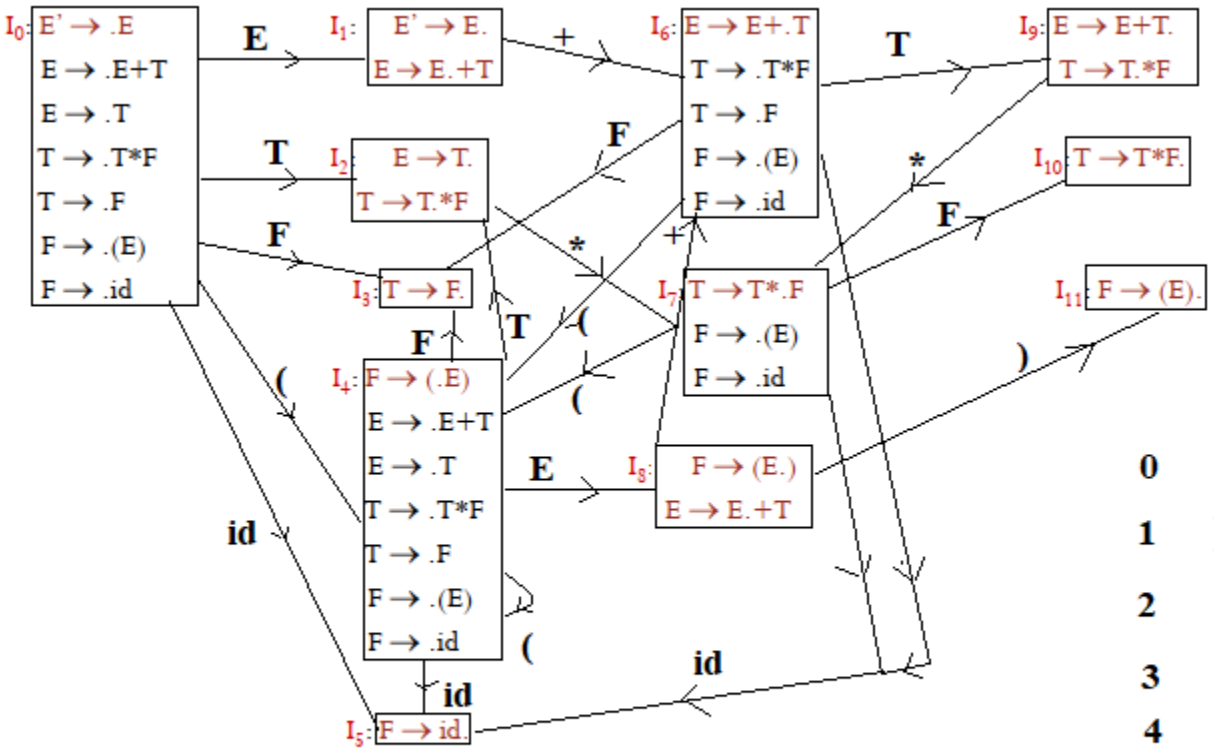
- To create the SLR parsing tables for a grammar G, we will create the canonical LR(0) collection of the grammar G`.
- C={closure(S`→.S)}
- Repeat the followings until no more set of LR(0) items can be added to C.

```
{
    For each set of items I in C and each grammar symbol X
    {
        If (GOTO(I,X) is not empty and is not in C)
        {
            add GOTO(I,X) to C.
        }
    }
```

- GOTO function of the canonical collection of set of items defines a deterministic finite automata that recognize the viable prefixes of G.

UPES
UNIVERSITY WITH A PURPOSE

Canonical LR(0) Collection-Example



0	$E' \rightarrow E$
1	$E \rightarrow E+T$
2	$E \rightarrow T$
3	$T \rightarrow T*F$
4	$T \rightarrow F$
5	$F \rightarrow (E)$
6	$F \rightarrow id$

First(E)={(,id}

First(T)={(,id}

First(F)={(,id}

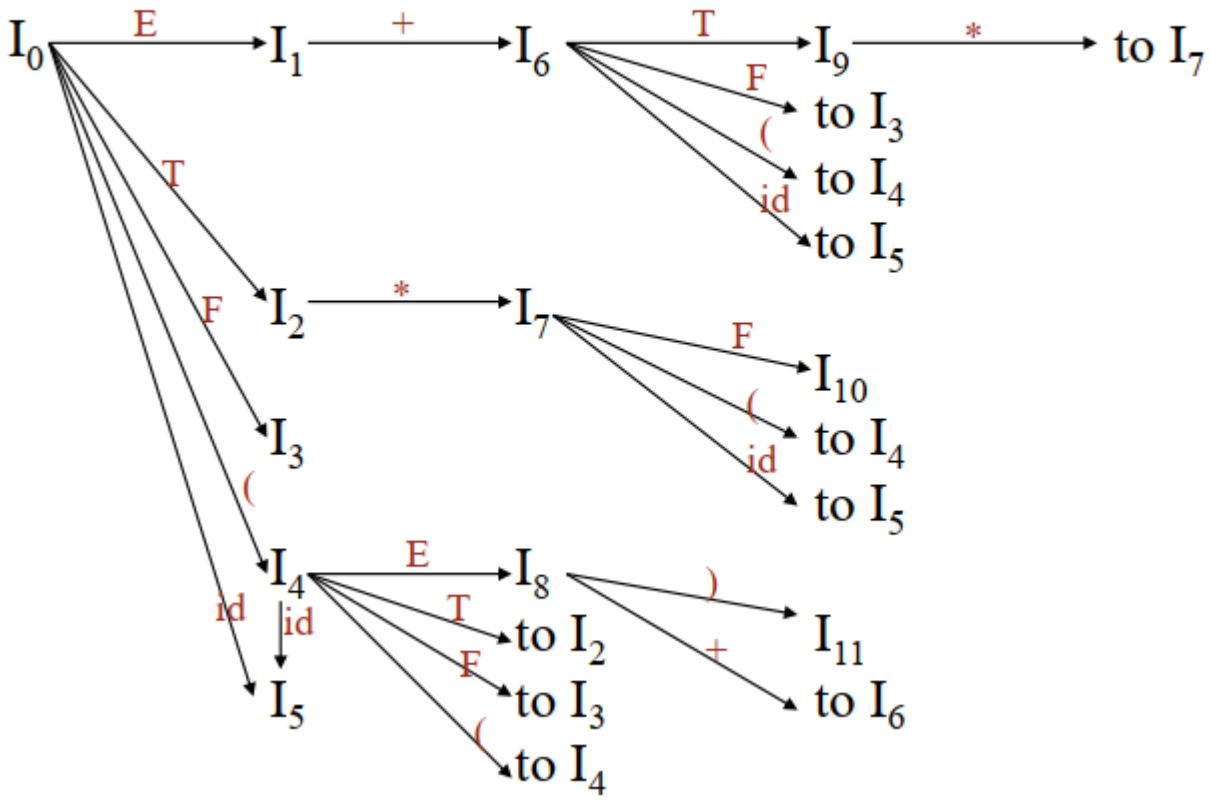
Follow(E)={$,+,)}

Follow(T)={$,+,),*}

Follow(F)={$,+,),*}

Canonical LR(0) Collection-Example

Transition Diagram (DFA)



SLR parser

- Construct DFA from the grammar.
- Convert DFA into LR parsing table.

Constructing SLR Parsing Table

(of 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. Create the parsing action table 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 contains $S' \rightarrow .S$

UPES
UNIVERSITY WITH A PURPOSE

Parsing Tables of Expression Grammar

Sr. No	stack	input
1	0	id*id+id$
2	0id5	*id+id$
3	0F3	*id+id$
4	0T2	*id+id$
5	0T2*7	id+id$
6	0T2*7id5	+id$
7	0T2*7F10	+id$
8	0T2	+id$
9	0E1	+id$
10	0E1+6	id$
11	0E1+6id5	$
12	0E1+6F3	$
13	0E1+6T9	$
14	0E1	$
15	Accept	$

	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 LR Parser

1. **shift s** -- shifts the next input symbol and the state **s** onto the stack

   $(S_o X_1 S_1 \dots X_m S_m, a_i a_{i+1} \dots a_n \$) \rightarrow (S_o X_1 S_1 \dots X_m S_m a_i s, a_{i+1} \dots a_n \$)$

2. **reduce $A \rightarrow \beta$** (or **rn** where n is a production number)
   - pop $2|\beta|$ (=r) items from the stack;
   - then push **A** and **s** where **$s = goto[s_{m-r}, A]$**

   $(S_o X_1 S_1 \dots X_m S_m, a_i a_{i+1} \dots a_n \$) \rightarrow (S_o X_1 S_1 \dots X_{m-r} S_{m-r} A s, a_i \dots a_n \$)$

   - Output is the reducing production $A \rightarrow \beta$

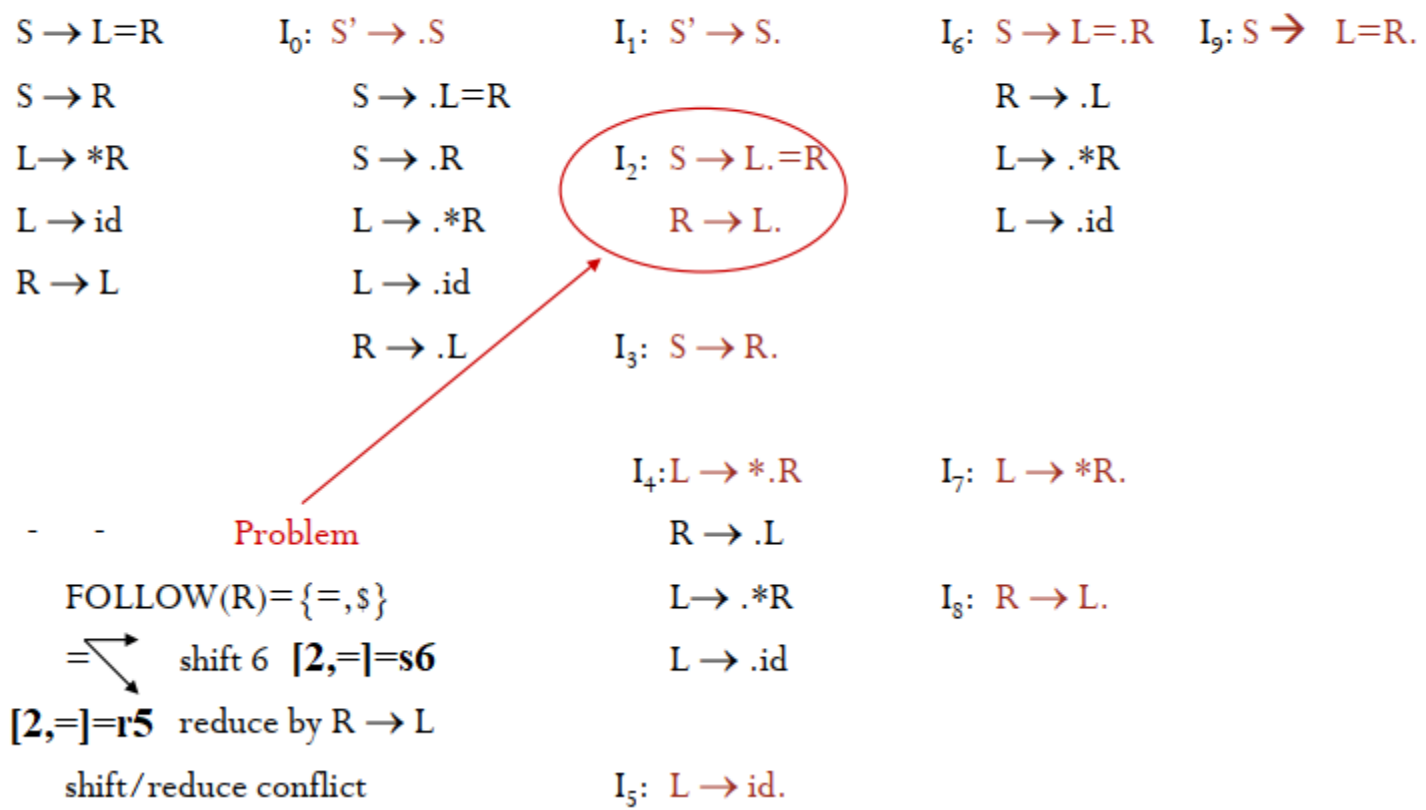
3. **Accept** – Parsing successfully completed

4. **Error** -- Parser detected an error (an empty entry in the action table)

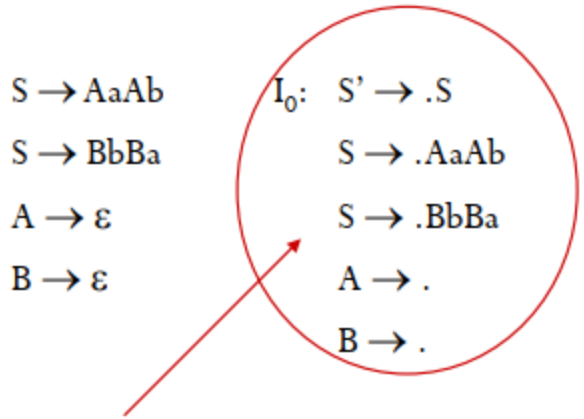
shift/reduce and reduce/reduce conflicts

- **shift/reduce conflict:** choice between a shift operation or reduction for a terminal
- **reduce/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
- If the SLR parsing table of a grammar G has a conflict, we say that the grammar is not SLR grammar

Conflict Example



Conflict Example2



Problem

FOLLOW(A)={a,b}

FOLLOW(B)={a,b}

a → reduce by A → ε

a → reduce by B → ε

[0,a]=r3

[0,a]=r4

reduce/reduce conflict

b → reduce by A → ε

b → reduce by B → ε

reduce/reduce conflict

LR(1) Item

- To avoid different kind of conflicts, 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 of the following form:

  $A \rightarrow \alpha.\beta, a$ where a is the look-ahead of the LR(1) item (a is a terminal or end-marker)
- When β (in the LR(1) item $A \rightarrow \alpha.\beta,a$) is not empty, the look-ahead does not have any effect
- When β is empty ($A \rightarrow \alpha.,a$), then Reduce by $A \rightarrow \alpha$ only if the next input symbol is **a** (not for any terminal in FOLLOW(A))
- A state of LR(1) items contains

  $A \rightarrow \alpha.,a_1$ where $\{a_1,...,a_n\} \subseteq FOLLOW(A)$

  ...

  $A \rightarrow \alpha.,a_n$

UPES
UNIVERSITY WITH A PURPOSE

Canonical Collection of Sets of LR(1) Items

- Process: similar to the construction of the canonical collection of the sets of LR(0) items,

  but *closure* and *goto* operations work a little bit different

closure(I) : (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 belongs to closure(I) for each terminal b in FIRST(βa)

UPES
UNIVERSITY WITH A PURPOSE

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) defined as follows:

- If $A \rightarrow \alpha.X\beta$, a in I

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

Construction of The Canonical LR(1) Collection

- ***Algorithm***:

  C is { closure({S'→.S,$}) }

  repeat the followings until no more set of LR(1) items can be added to ***C***.

  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***

- goto function is a DFA on the sets in C

A Short Notation for The Sets of LR(1) Items

- A set of LR(1) items containing the following items

$$A \rightarrow \alpha . \beta, a_1$$

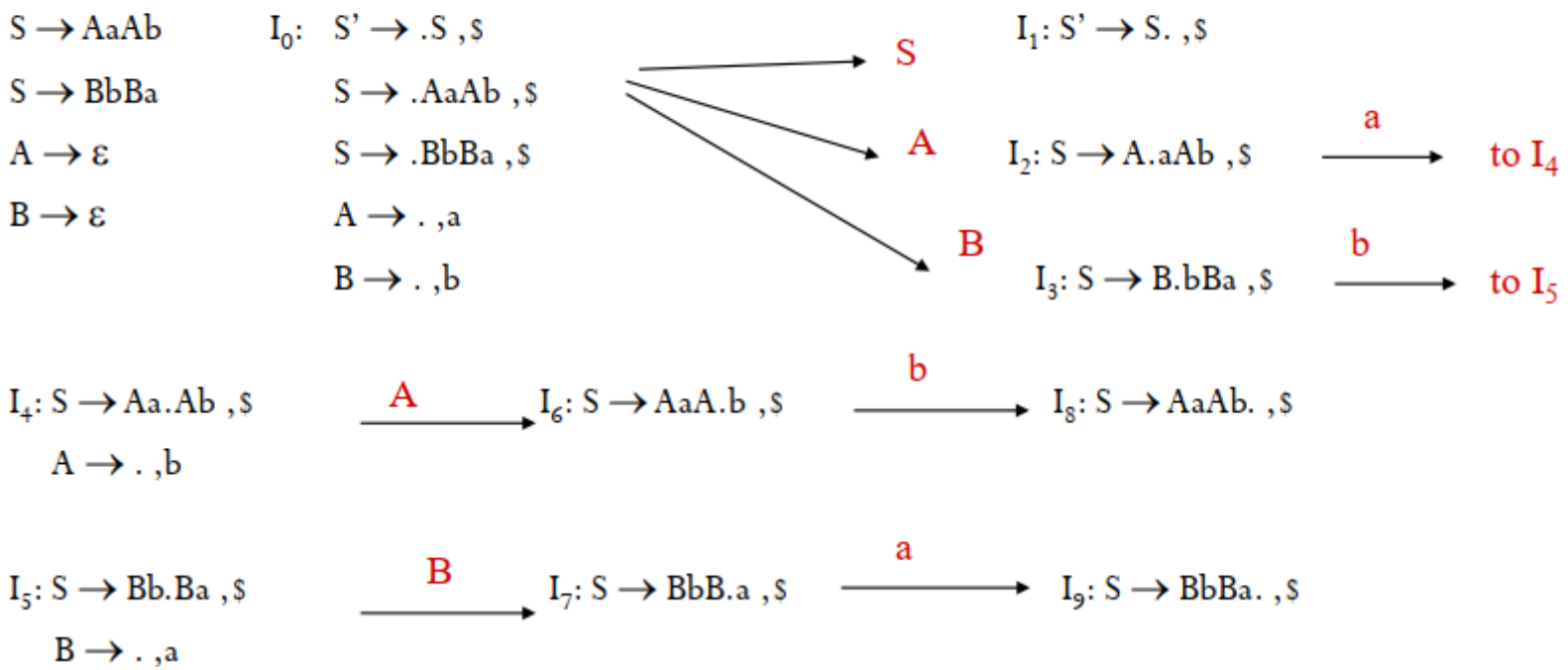
$$\ldots$$

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

can be written as

$$A \rightarrow \alpha . \beta, a_1 / a_2 / \ldots / a_n$$

Canonical LR(1) Collection -- Example 1



Construction of LR(1) Parsing Tables

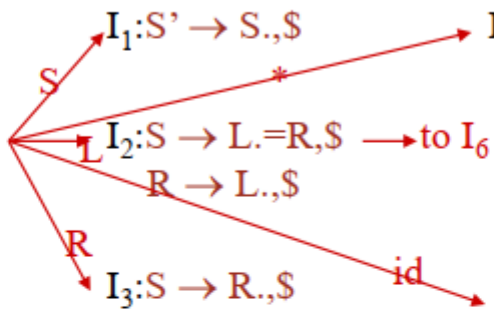
1. Construct the canonical collection of sets of LR(1) items for G'.
   $C \leftarrow \{I_0, \ldots, I_n\}$
2. Create the parsing action table as follows
   - If a is a terminal, $A \rightarrow \alpha \cdot a\beta, b$ in I_i and $goto(I_i, a) = I_j$ then action[i,a] is ***shift j.***
   - If $A \rightarrow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

UPES
UNIVERSITY WITH A PURPOSE

Canonical LR(1) Collection – Example 2

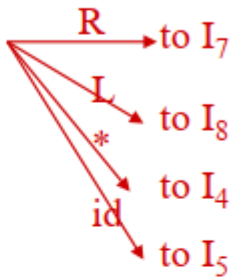
S' → S
1) S → L=R
2) S → R
3) L → *R
4) L → id
5) R → L

I_0: S' → .S,$
S → .L=R,$
S → .R,$
L → .*R,=
L → .id,=
R → .L,$

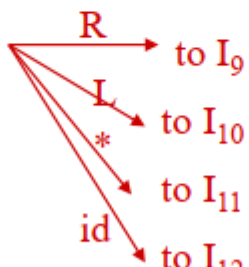


I_4:L → *.R,=
R → .L, =
L → .*R, =
L → .id, =

I_5:L → id., =



I_6:S → L=.R,$
R → .L,$
L → .*R,$
L → .id,$



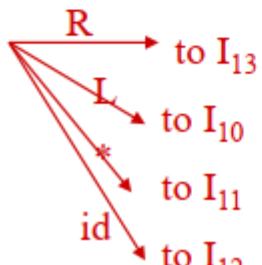
I_7:L → *R.,=

I_8: R → L.,=

I_9:S → L=R.,$

I_{10}:R → L.,$

I_{11}:L → *.R,$
R → .L,$
L → .*R,$
L → .id,$



I_{12}:L → id.,$

I_{13}:L → *R.,$

I_4 and I_{11}

I_5 and I_{12}

I_7 and I_{13}

I_8 and I_{10}

LR(1) Parsing Tables – (for Example2)

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

UPES
UNIVERSITY WITH A PURPOSE

LALR Parsing Tables

- **LALR** stands for **LookAhead LR**
- LALR parsers are often used in practice because LALR parsing tables are smaller than LR(1) parsing tables
- Number of states in SLR and LALR parsing tables for a grammar G are equal
- But, LALR parsers recognize more grammars than SLR parsers

- *yacc* creates a LALR parser for the given grammar

- A state of LALR parser will be again a set of LR(1) items

UPES
UNIVERSITY WITH A PURPOSE

Creating LALR Parsing Tables

Canonical LR(1) Parser → LALR Parser

shrink # of states

- 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

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

- Find the states (sets of LR(1) items) in a canonical LR(1) parser with the same cores. 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

- Do this for all states of a canonical LR(1) parser to get the states of the LALR parser

number of the states of the LALR parser = number of states of the SLR parser for any grammar

UPES
UNIVERSITY WITH A PURPOSE

Creation of LALR Parsing Tables

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

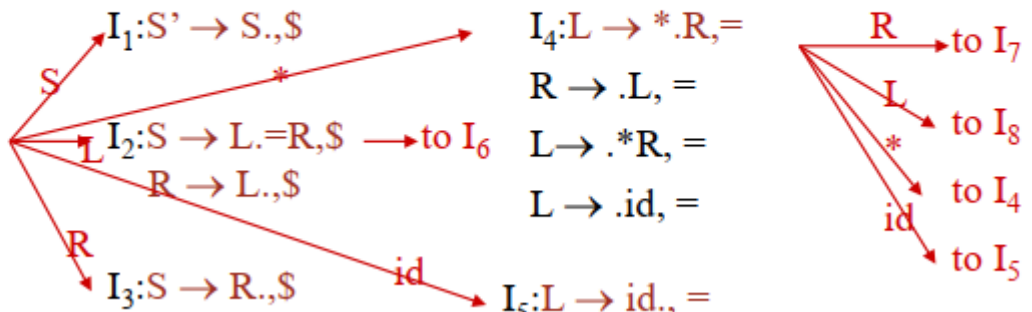
  $C=\{I_0,\ldots,I_n\} \rightarrow C'=\{J_1,\ldots,J_m\}$ where $m \leq n$
- Create the parsing tables (action and goto tables) same as the construction of the parsing tables of LR(1) parser
  - Note that: If $J=I_1 \cup \ldots \cup I_k$ since $I_1,\ldots,I_k$ have same cores

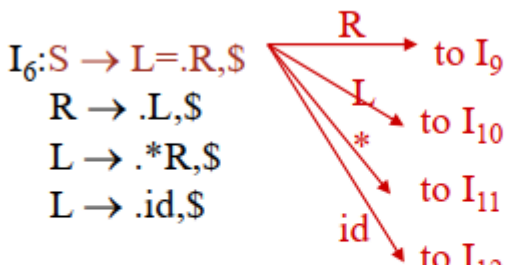
    → cores of goto $(I_1,X),\ldots,$ goto (I_2,X) must be same
  - So, goto $(J,X)=K$, where K is the union of all sets of items having same cores as goto (I_1,X)
- Grammar is LALR(1) if no conflict is introduced
  - possible to introduce reduce/reduce conflicts
  - cannot introduce a shift/reduce conflict

Canonical LR(1) Collection – Example 2

S' → S
1) S → L=R
2) S → R
3) L→ *R
4) L → id
5) R → L

I_0: S' → .S,$
S → .L=R,$
S → .R,$
L → .*R,=
L → .id,=
R → .L,$



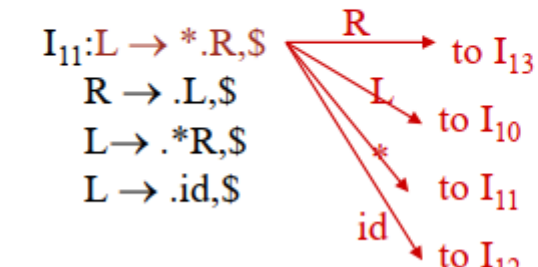


I_7:L → *R.,=

I_8: R → L.,=

I_9:S → L=R.,$

I_{10}:R → L.,$



I_{12}:L → id.,$

I_{13}:L → *R.,$

I_4 and I_{11}

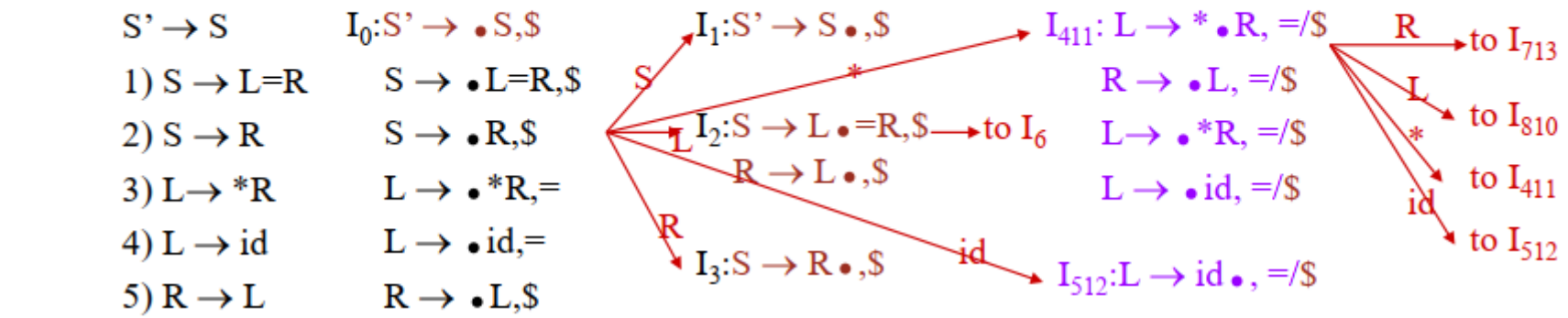
I_5 and I_{12}

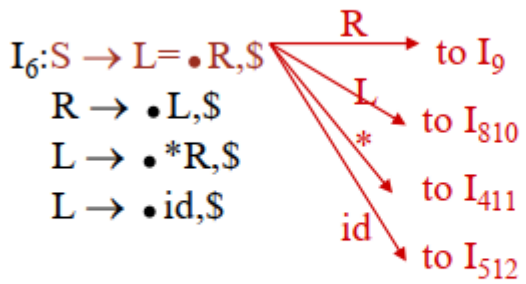
I_7 and I_{13}

I_8 and I_{10}

Canonical LALR(1) Collection – Example2

S' → S
1) S → L=R
2) S → R
3) L→ *R
4) L → id
5) R → L





I713:L → *R•,=/$

I810: R → L•, =/$

I9:S → L=R•,$

Same Cores

I_4 and I_{11}

I_5 and I_{12}

I_7 and I_{13}

I_8 and I_{10}

UPES
UNIVERSITY WITH A PURPOSE

LR(1) Parsing Tables – (for Example2)

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

no shift/reduce or
no reduce/reduce conflict

so, it is a LR(1) grammar

LALR(1) Parsing Tables – (for Example2)

	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	s12	s11				10	9
7			r3				
8			r5				
9				r1			

no shift/reduce or
no reduce/reduce conflict

so, it is a LALR(1) grammar

THANK YOU

UPES
UNIVERSITY WITH A PURPOSE

Dr Anurag Jain
Assistant Professor (SG), Virtualization Department, School of Computer Science,
University of Petroleum & Energy Studies, Dehradun - 248 007 (Uttarakhand)
Email: anurag.jain@ddn.upes.ac.in , dr.anuragjain14@gmail.com
Mobile: (+91) -9729371188