# \*Shift Reduce Parsers

* Build the parse tree from leaves to root.
* Bottom-up parsing can be defined as an attempt to reduce the input string w to the start symbol of grammar by tracing out the rightmost derivations of w in reverse.

|  |
| --- |
| A shift-reduce parser is a bottom-up parser. |

It takes the given input string and builds a parse tree-

* Starting from the bottom at the leaves.
* And growing the tree towards the top to the root.



## **Data Structures-**

## 

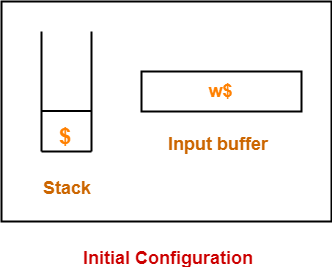
Two data structures are required to implement a shift-reduce parser-

* A **Stack** is required to hold the grammar symbols.
* An **Input buffer** is required to hold the string to be parsed.

## **Working-**

Initially, shift-reduce parser is present in the following configuration where-

* Stack contains only the $ symbol.
* Input buffer contains the input string with $ at its end.

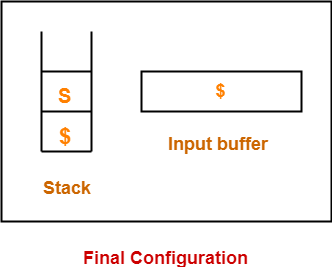


The parser works by-

* Moving the input symbols on the top of the stack.
* Until a handle β appears on the top of the stack.

The parser keeps on repeating this cycle until-

* An error is detected.
* Or stack is left with only the start symbol and the input buffer becomes empty.



After achieving this configuration,

* The parser stops / halts.
* It reports the successful completion of parsing.

## **Possible Actions-**

A shift-reduce parser can possibly make the following **four actions-**

### **1. Shift-**

In a shift action,

* The next symbol is shifted onto the top of the stack.

### **2. Reduce-**

In a reduce action,

* The handle appearing on the stack top is replaced with the appropriate non-terminal symbol.

### **3. Accept-**

In an accept action,

* The parser reports the successful completion of parsing.

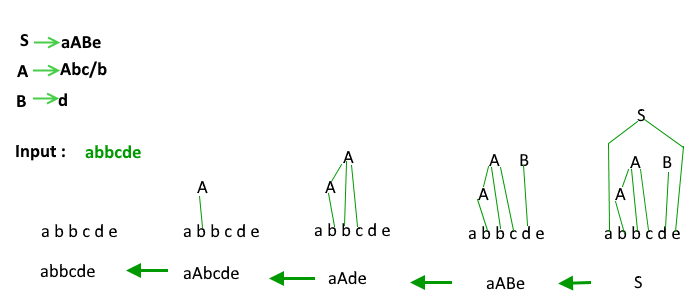
### **4. Error-**

In this state,

* The parser becomes confused and is not able to make any decision.
* It can neither perform shift action nor reduce action nor accept action.

|  |
| --- |
| **Rules To Remember**   It is important to remember the following rules while performing the shift-reduce action-   * If the priority of incoming operator is more than the priority of in stack operator, then shift action is performed. * If the priority of in stack operator is same or less than the priority of in stack operator, then reduce action is performed. |

Eg.



**PRACTICE PROBLEMS BASED ON SHIFT-REDUCE PARSING-**

**Problem-01:**

Consider the following grammar-

**E → E – E**

**E → E x E**

**E → id**

Parse the **input string id – id x id** using a shift-reduce parser.

**Solution-**

The priority order is: id > x > –

|  |  |  |
| --- | --- | --- |
| **Stack** | **Input Buffer** | **Parsing Action** |
| $ | id – id x id $ | Shift |
| $ id | – id x id $ | Reduce E → id |
| $ E | – id x id $ | Shift |
| $ E – | id x id $ | Shift |
| $ E – id | x id $ | Reduce E → id |
| $ E – E | x id $ | Shift |
| $ E – E x | id $ | Shift |
| $ E – E x id | $ | Reduce E → id |
| $ E – E x E | $ | Reduce E → E x E |
| $ E – E | $ | Reduce E → E – E |
| $ E | $ | Accept |

**Problem-02:**

Consider the following grammar-

S → ( L ) | a

L → L , S | S

Parse the input string ( a , ( a , a ) ) using a shift-reduce parser.

**Solution-**

|  |  |  |
| --- | --- | --- |
| **Stack** | **Input Buffer** | **Parsing Action** |
| $ | ( a , ( a , a ) ) $ | Shift |
| $ ( | a , ( a , a ) ) $ | Shift |
| $ ( a | , ( a , a ) ) $ | Reduce S → a |
| $ ( S | , ( a , a ) ) $ | Reduce L → S |
| $ ( L | , ( a , a ) ) $ | Shift |
| $ ( L , | ( a , a ) ) $ | Shift |
| $ ( L , ( | a , a ) ) $ | Shift |
| $ ( L , ( a | , a ) ) $ | Reduce S → a |
| $ ( L , ( S | , a ) ) $ | Reduce L → S |
| $ ( L , ( L | , a ) ) $ | Shift |
| $ ( L , ( L , | a ) ) $ | Shift |
| $ ( L , ( L , a | ) ) $ | Reduce S → a |
| $ ( L , ( L , S ) | ) ) $ | Reduce L → L , S |
| $ ( L , ( L | ) ) $ | Shift |
| $ ( L , ( L ) | ) $ | Reduce S → (L) |
| $ ( L , S | ) $ | Reduce L → L , S |
| $ ( L | ) $ | Shift |
| $ ( L ) | $ | Reduce S → (L) |
| $ S | $ | Accept |

**Problem-03:**

Consider the following grammar-

S → T L

T → int | float

L → L , id | id

Parse the input string int id , id ; using a shift-reduce parser.

**Solution-**

|  |  |  |
| --- | --- | --- |
| **Stack** | **Input Buffer** | **Parsing Action** |
| $ | int id , id ; $ | Shift |
| $ int | id , id ; $ | Reduce T → int |
| $ T | id , id ; $ | Shift |
| $ T id | , id ; $ | Reduce L → id |
| $ T L | , id ; $ | Shift |
| $ T L , | id ; $ | Shift |
| $ T L , id | ; $ | Reduce L → L , id |
| $ T L | ; $ | Shift |
| $ T L ; | $ | Reduce S → T L |
| $ S | $ | Accept |

**Problem-04:**

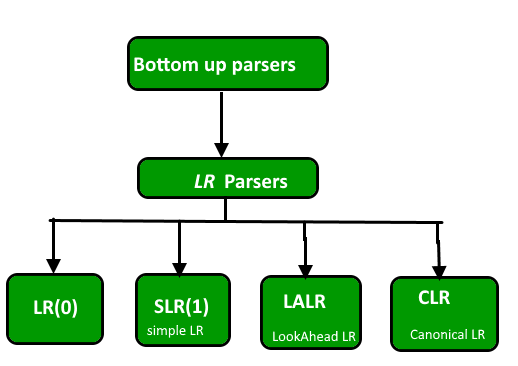
Considering **the string “10201”,** design a shift-reduce parser for the following grammar-

**S → 0S0 | 1S1 | 2**

**Solution-**

|  |  |  |
| --- | --- | --- |
| **Stack** | **Input Buffer** | **Parsing Action** |
| $ | 1 0 2 0 1 $ | Shift |
| $ 1 | 0 2 0 1 $ | Shift |
| $ 1 0 | 2 0 1 $ | Shift |
| $ 1 0 2 | 0 1 $ | Reduce S → 2 |
| $ 1 0 S | 0 1 $ | Shift |
| $ 1 0 S 0 | 1 $ | Reduce S → 0 S 0 |
| $ 1 S | 1 $ | Shift |
| $ 1 S 1 | $ | Reduce S → 1 S 1 |
| $ S | $ | Accept |

**Classification of bottom up parsers**



A general shift reduce parsing is **LR parsing**. The **L stands** for scanning the input from left to right and **R stands** for constructing a rightmost derivation in reverse.

**Benefits of LR parsing:**

1. Many programming languages using some variations of an LR parser. It should be noted that C++ and Perl are exceptions to it.
2. LR Parser can be implemented very efficiently.
3. Of all the Parsers that scan their symbols from left to right, LR Parsers detect syntactic errors, as soon as possible.

**\*LR(0) Parser**  
We need **two functions** –  
1. Closure ()

2.Goto ()

**Augmented Grammar**

If G is a grammar with start symbol S then G’, the augmented grammar for G, is the grammar with new start symbol S’ and **a production S’ -> S**. The purpose of this new starting production is to indicate to the parser when it should stop parsing and announce acceptance of input.

Let a grammar be S -> AA  
 A -> aA | b  
The augmented grammar for the above grammar will be  
S’ -> S  
S -> AA  
A -> aA | b

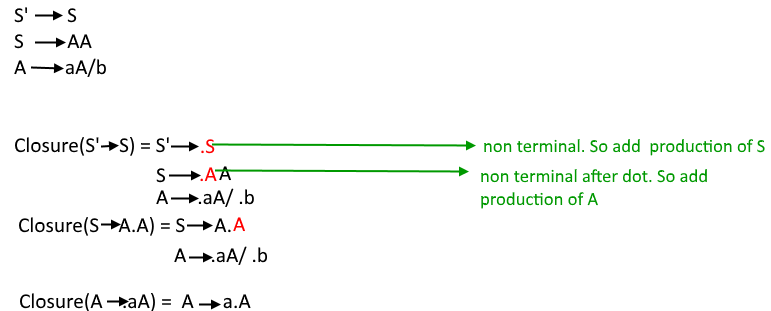
**LR(0) Items**  
An LR(0) is the item of a grammar G is a production of G **with a dot at some position** in the right side.

S -> ABC yields four items  
S -> .ABC  
S -> A.BC  
S -> AB.C  
S -> ABC.  
The production A -> ε generates only one item A -> .ε

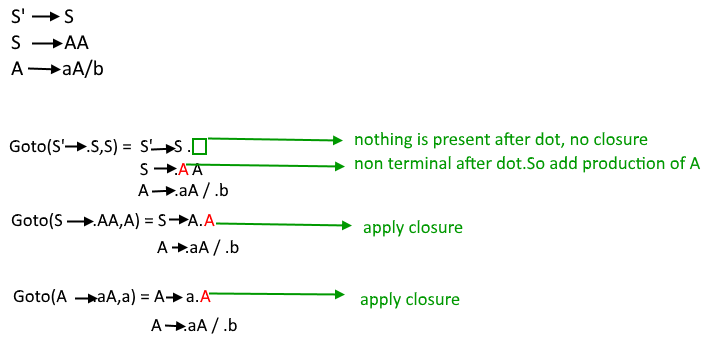
**Closure Operation**:  
If I is a set of items for a grammar G, then closure(I) is the set of items constructed from I by the **two rules:**

1. Initially every item in I is added to closure(I).
2. If A -> α.Bβ is in closure(I) and B -> γ is a production then add the item B -> .γ to I, If it is not already there. We apply this rule until no more items can be added to closure(I).

Eg:



**Goto Operation**:  
Goto(I, X) = 1. Add I by moving dot after X.  
                      2. Apply closure to first step.



**Construction of GOTO graph-**

* State I0 – closure of augmented LR(0) item
* Using I0 find all collection of sets of LR(0) items with the help of DFA
* Convert DFA to LR(0) parsing table

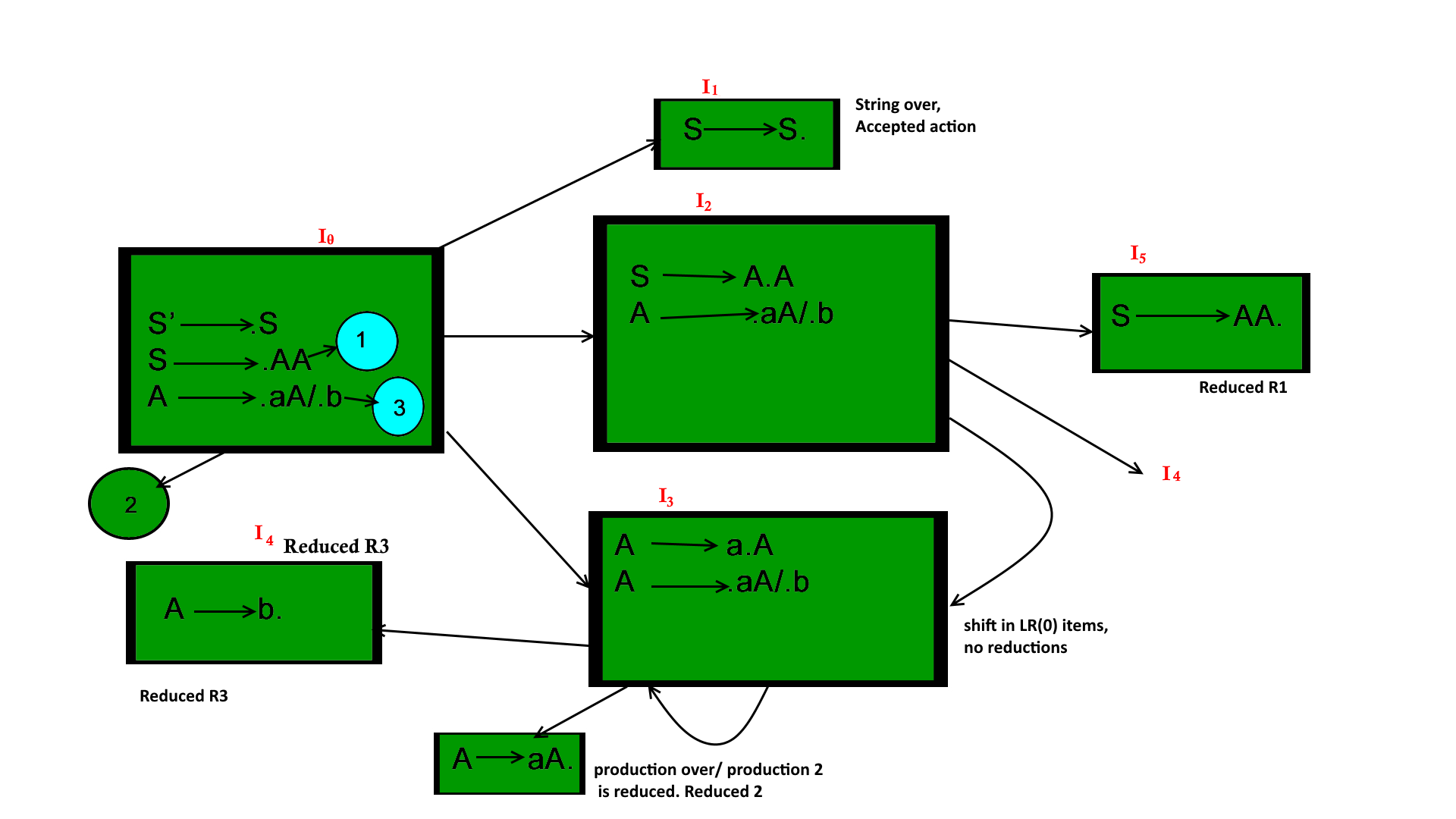
**Construction of LR(0) parsing table**:

* The action function takes as arguments a state i and a terminal a (or $ , the input end marker). The value of ACTION[i, a] can have one of four forms:
  1. Shift j, where j is a state.
  2. Reduce A -> β.
  3. Accept
  4. Error
* We extend the GOTO function, defined on sets of items, to states: if GOTO[Ii , A] = Ij then GOTO also maps a state i and a nonterminal A to state j.

Eg:  
Consider the grammar **S ->AA  
 A -> aA | b  
Augmented grammar**

**S’ -> S  
S -> AA  
A -> aA | b**

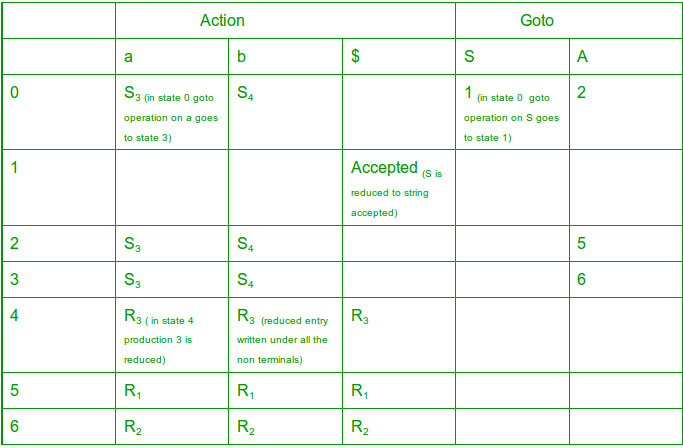
The LR(0) parsing table for above GOTO graph will be –



I6

Action part of the table contains all the terminals of the grammar whereas the goto part contains all the nonterminal. For every state of goto graph we write all the goto operations in the table. If goto is applied to a terminal than it is written in the action part if goto is applied on a nonterminal it is written in goto part. If on applying goto a production is reduced ( i.e if the dot reaches at the end of production and no further closure can be applied) then it is denoted as Ri and if the production is not reduced (shifted) it is denoted as Si.  
If a production is reduced it is written under the terminals given by follow of the left side of the production which is reduced for ex: in I5 S->AA is reduced so R1 is written under the terminals in follow(S)={$} (To know more about how to calculate follow function: [Click here](https://www.geeksforgeeks.org/compiler-design-follow-set-in-syntax-analysis/) ) in LR(0) parser.  
If in a state the start symbol of grammar is reduced it is written under $ symbol as accepted.

**NOTE:** If in any state both reduced and shifted productions are present or two reduced productions are present it is called a conflict situation and the grammar is not LR grammar.

****

**NOTE:**  
1. Two reduced productions in one state – RR conflict.  
2. One reduced and one shifted production in one state – SR conflict.  
  
If no SR or RR conflict present in the parsing table then the grammar is LR(0) grammar.  
In above grammar no conflict so it is LR(0) grammar.  
  
**NOTE —**In solving GATE question we don’t need to make the parsing table, by looking at the GOTO graph only we can determine if the grammar is LR(0) grammar or not. We just have to look for conflicts in the goto graph i.e if a state contains two reduced or one reduced and one shift entry for a TERMINAL variable then there is a conflict and it is not LR(0) grammar. (In case of one shift with a VARIABLE and one reduced there will be no conflict because then the shift entries will go to GOTO part of table and reduced entries will go in ACTION part and thus no multiple entries).

## **\*Handling Parsing Conflicts**

### **Types of conflicts**

There are two kinds of conflicts that can occur in an SLR(1) parsing table.

1. A *shift-reduce* conflict occurs in a state that requests both a shift action and a reduce action.
2. A *reduce-reduce* conflict occurs in a state that requests two or more different reduce actions.

**\*SLR Parser**  
The SLR parser is similar to LR(0) parser except that the reduced entry. The reduced productions are written only in the FOLLOW of the variable whose production is reduced.  
  
**Construction of SLR parsing table –**

1. Construct C = { I0, I1, ……. In}, the collection of sets of LR(0) items for G’.
2. State i is constructed from Ii. The parsing actions for state i are determined as follow :
   * If [ A -> ?.a? ] is in Ii and GOTO(Ii , a) = Ij , then set ACTION[i, a] to “shift j”. Here a must be terminal.
   * If [A -> ?.] is in Ii, then set ACTION[i, a] to “reduce A -> ?” for all a in FOLLOW(A); here A may not be S’.
   * Is [S -> S.] is in Ii, then set action[i, $] to “accept”. If any conflicting actions are generated by the above rules we say that the grammar is not SLR.
3. The goto transitions for state i are constructed for all nonterminals A using the rule:  
   if GOTO( Ii , A ) = Ij then GOTO [i, A] = j.
4. All entries not defined by rules 2 and 3 are made error.

Eg:  
If in the parsing table we have multiple entries then it is said to be a conflict.

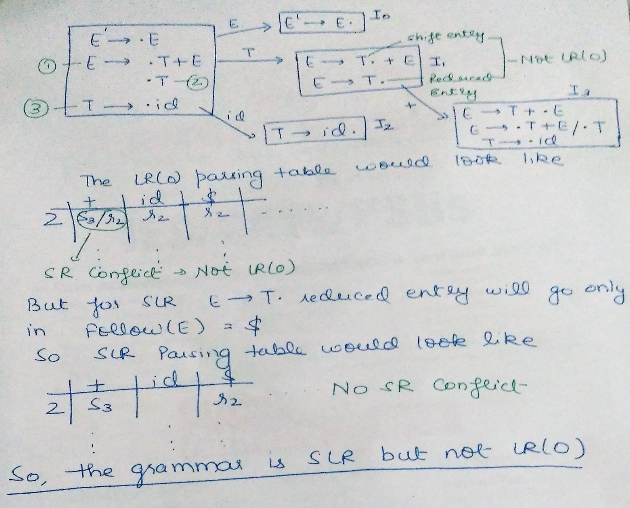
Consider the grammar E -> T+E | T

T ->id

Augmented grammar - E’ -> E

E -> T+E | T

T -> id

[](https://media.geeksforgeeks.org/wp-content/cdn-uploads/gq/2017/02/parser_16.png)

**Note 1**– for GATE we don’t have to draw the table, in the GOTO graph just look for the reduce and shifts occurring together in one state. In case of two reductions, if the follow of both the reduced productions have something common then it will result in multiple entries in table hence not SLR. In case of one shift and one reduction, if there is a GOTO operation from that state on a terminal which is the follow of the reduced production than it will result in multiple entries hence not SLR.  
**Note 2** – Every SLR grammar is unambiguous but there are many unambiguous grammars that are not SLR.

**\*CLR PARSER**

CLR refers to canonical lookahead. CLR parsing use the canonical collection of LR (1) items to build the CLR (1) parsing table. CLR (1) parsing table produces the more number of states as compare to the SLR (1) parsing.

In the CLR (1), we place the reduce node only in the lookahead symbols.

Various steps involved in the CLR (1) Parsing:

* For the given input string write a context free grammar
* Check the ambiguity of the grammar
* Add Augment production in the given grammar
* Create Canonical collection of LR (0) items
* Draw a data flow diagram (DFA)
* Construct a CLR (1) parsing table

**LR (1) item**

LR (1) item is a collection of LR (0) items and a look ahead symbol.

**LR (1) item = LR (0) item + look ahead**

The look ahead is used to determine that where we place the final item.

The look ahead always add $ symbol for the argument production.

### **Example**

**CLR ( 1 ) Grammar**

1. S → AA
2. A → aA
3. A → b

Add Augment Production, insert '•' symbol at the first position for every production in G and also add the lookahead.

1. S` → •S, $
2. S  → •AA, $
3. A  → •aA, a/b
4. A → •b, a/b

**I0 State:**

Add Augment production to the I0 State and Compute the Closure

**I0 =** Closure (S` → •S)

Add all productions starting with S in to I0 State because "." is followed by the non-terminal. So, the I0 State becomes

**I0 =** S` → •S, $  
        S → •AA, $

Add all productions starting with A in modified I0 State because "." is followed by the non-terminal. So, the I0 State becomes.

**I0=**  S` → •S, $  
        S → •AA, $  
        A → •aA, a/b  
        A → •b, a/b

**I1=** Go to (I0, S) = closure (S` → S•, $) = S` → S•, $  
**I2=** Go to (I0, A) = closure ( S → A•A, $ )

Add all productions starting with A in I2 State because "." is followed by the non-terminal. So, the I2 State becomes

**I2=** S → A•A, $  
       A → •aA, $  
       A → •b, $

**I3=** Go to (I0, a) = Closure ( A → a•A, a/b )

Add all productions starting with A in I3 State because "." is followed by the non-terminal. So, the I3 State becomes

**I3=**A → a•A, a/b  
       A → •aA, a/b  
       A → •b, a/b

Go to (I3, a) = Closure (A → a•A, a/b) = (same as I3)  
Go to (I3, b) = Closure (A → b•, a/b) = (same as I4)

**I4=** Go to (I0, b) = closure ( A → b•, a/b) = A → b•, a/b  
**I5=** Go to (I2, A) = Closure (S → AA•, $) =S → AA•, $  
**I6=** Go to (I2, a) = Closure (A → a•A, $)

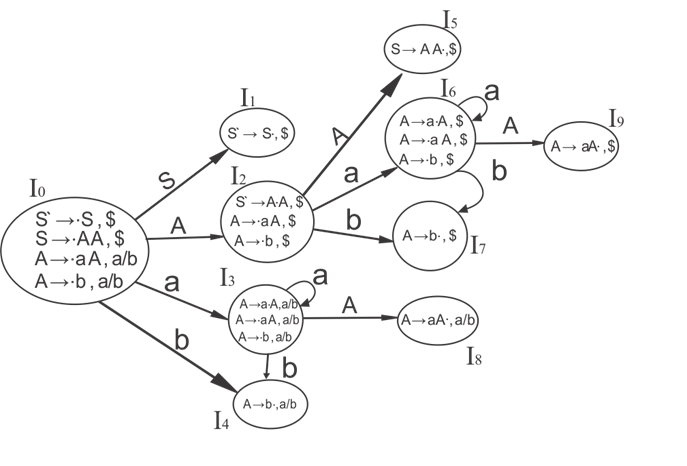
Add all productions starting with A in I6 State because "." is followed by the non-terminal. So, the I6 State becomes

**I6 =** A → a•A, $  
       A → •aA, $  
       A → •b, $

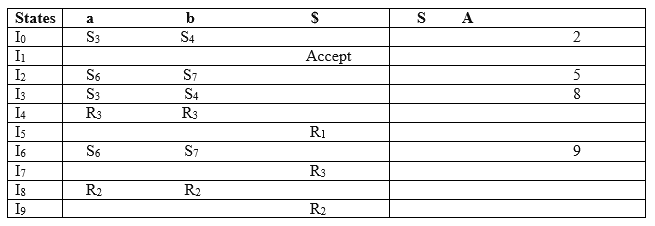
Go to (I6, a) = Closure (A → a•A, $) = (same as I6)  
Go to (I6, b) = Closure (A → b•, $) = (same as I7)

**I7=** Go to (I2, b) = Closure (A → b•, $) = A → b•, $  
**I8=** Go to (I3, A) = Closure (A → aA•, a/b) = A → aA•, a/b  
**I9=** Go to (I6, A) = Closure (A → aA•, $) = A → aA•, $

### **Drawing DFA:**



## **CLR (1) Parsing table:**



Productions are numbered as follows:

1. S  →  AA      ... (1)
2. A  → aA       ....(2)
3. A  →  b     ... (3)

The placement of shift node in CLR (1) parsing table is same as the SLR (1) parsing table. Only difference in the placement of reduce node.

I4 contains the final item which drives ( A → b•, a/b), so action {I4, a} = R3, action {I4, b} = R3.  
I5 contains the final item which drives ( S → AA•, $), so action {I5, $} = R1.  
I7 contains the final item which drives ( A → b•,$), so action {I7, $} = R3.  
I8 contains the final item which drives ( A → aA•, a/b), so action {I8, a} = R2, action {I8, b} = R2.  
I9 contains the final item which drives ( A → aA•, $), so action {I9, $} = R2.

**\*LALR PARSER**

LALR refers to the lookahead LR. To construct the LALR (1) parsing table, we use the canonical collection of LR (1) items.

In the LALR (1) parsing, the LR (1) items which have same productions but different look ahead are combined to form a single set of items

LALR (1) parsing is same as the CLR (1) parsing, only difference in the parsing table.

### **Example**

**LALR ( 1 ) Grammar**

1. S → AA
2. A  → aA
3. A → b

Add Augment Production, insert '•' symbol at the first position for every production in G and also add the look ahead.

1. S` → •S, $
2. S  → •AA, $
3. A  → •aA, a/b
4. A  → •b, a/b

**I0 State:**

Add Augment production to the I0 State and Compute the ClosureL

**I0 =** Closure (S` → •S)

Add all productions starting with S in to I0 State because "•" is followed by the non-terminal. So, the I0 State becomes

**I0 =**S` → •S, $  
        S → •AA, $

Add all productions starting with A in modified I0 State because "•" is followed by the non-terminal. So, the I0 State becomes.

**I0=** S` → •S, $  
       S → •AA, $  
       A → •aA, a/b  
       A → •b, a/b

**I1=** Go to (I0, S) = closure (S` → S•, $) = S` → S•, $  
**I2=** Go to (I0, A) = closure ( S → A•A, $ )

Add all productions starting with A in I2 State because "•" is followed by the non-terminal. So, the I2 State becomes

**I2=** S → A•A, $  
       A → •aA, $  
       A → •b, $

**I3=** Go to (I0, a) = Closure ( A → a•A, a/b )

Add all productions starting with A in I3 State because "•" is followed by the non-terminal. So, the I3 State becomes

**I3=** A → a•A, a/b  
       A → •aA, a/b  
       A → •b, a/b

Go to (I3, a) = Closure (A → a•A, a/b) = (same as I3)  
Go to (I3, b) = Closure (A → b•, a/b) = (same as I4)

**I4=** Go to (I0, b) = closure ( A → b•, a/b) = A → b•, a/b  
**I5=** Go to (I2, A) = Closure (S → AA•, $) =S → AA•, $  
**I6=** Go to (I2, a) = Closure (A → a•A, $)

Add all productions starting with A in I6 State because "•" is followed by the non-terminal. So, the I6 State becomes

**I6 =** A → a•A, $  
       A → •aA, $  
       A → •b, $

Go to (I6, a) = Closure (A → a•A, $) = (same as I6)  
Go to (I6, b) = Closure (A → b•, $) = (same as I7)

**I7=** Go to (I2, b) = Closure (A → b•, $) = A → b•, $  
**I8=** Go to (I3, A) = Closure (A → aA•, a/b) = A → aA•, a/b  
**I9=** Go to (I6, A) = Closure (A → aA•, $) A → aA•, $

If we analyze then LR (0) items of I3 and I6 are same but they differ only in their lookahead.

**I3 =** { A → a•A, a/b  
      A → •aA, a/b  
      A → •b, a/b  
       }

**I6=** { A → a•A, $  
      A → •aA, $  
      A → •b, $  
      }

Clearly I3 and I6 are same in their LR (0) items but differ in their lookahead, so we can combine them and called as I36.

**I36 =** { A → a•A, a/b/$  
       A → •aA, a/b/$  
       A → •b, a/b/$  
        }

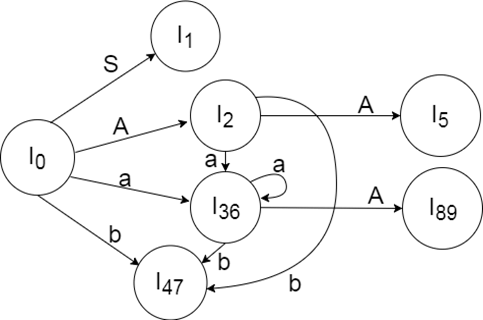
The I4 and I7 are same but they differ only in their look ahead, so we can combine them and called as I47.

**I47 =** {A → b•, a/b/$}

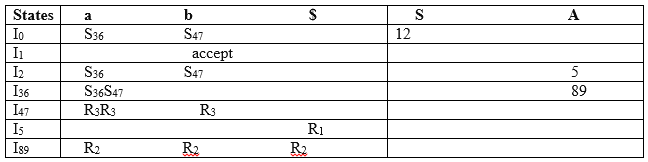
The I8 and I9 are same but they differ only in their look ahead, so we can combine them and called as I89.

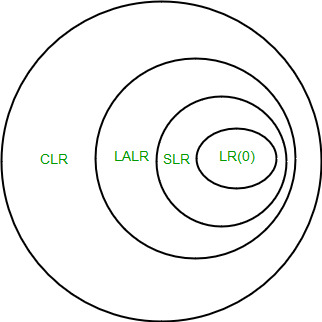
**I89 =** {A → aA•, a/b/$}

### **Drawing DFA:**



**LALR (1) Parsing table:**



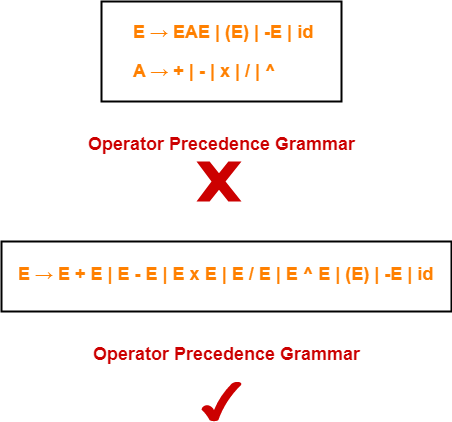
**Important Notes**  
1. Even though CLR parser does not have RR conflict but LALR may contain RR conflict.  
2. If number of states LR(0) = n1,  
  number of states SLR = n2,  
  number of states LALR = n3,  
  number of states CLR = n4 then,  
  n1 = n2 = n3 <= n4  
  
[](https://media.geeksforgeeks.org/wp-content/uploads/Parsing.jpg)

## **\*Operator Precedence Grammar-**

|  |
| --- |
| A grammar that satisfies the following 2 conditions is called as **Operator Precedence Grammar**–   * There exists no production rule which contains ε on its RHS. * There exists no production rule which contains two non-terminals adjacent to each other on its RHS. |

* It represents a small class of grammar.
* But it is an important class because of its widespread applications.

### **Examples-**



## **\*Operator Precedence Parser-**

|  |
| --- |
| A parser that reads and understand an operator precedence grammar  is called as **Operator Precedence Parser**. |

## **Designing Operator Precedence Parser-**

In operator precedence parsing,

* Firstly, we define precedence relations between every pair of terminal symbols.
* Secondly, we construct an operator precedence table.

## **Defining Precedence Relations-**

The precedence relations are defined using the following rules-

### **Rule-01:**

* If precedence of b is higher than precedence of a, then we define a < b
* If precedence of b is same as precedence of a, then we define a = b
* If precedence of b is lower than precedence of a, then we define a > b

### **Rule-02:**

* An **identifier** is always given the **higher precedence** than any other symbol.
* **$ symbol** is always given the **lowest precedence**.

### **Rule-03:**

* If two operators have the **same precedence**, then we go by **checking their associativity**.

## **Parsing A Given String-**

The given input string is parsed using the following steps-

### **Step-01:**

Insert the following-

* $ symbol at the beginning and ending of the input string.
* Precedence operator between every two symbols of the string by referring the operator precedence table.

### **Step-02:**

* Start scanning the string from LHS in the forward direction until > symbol is encountered.
* Keep a pointer on that location.

### **Step-03:**

* Start scanning the string from RHS in the backward direction until < symbol is encountered.
* Keep a pointer on that location.

### **Step-04:**

* Everything that lies in the middle of < and > forms the handle.
* Replace the handle with the head of the respective production.

### **Step-05:**

Keep repeating the cycle from Step-02 to Step-04 until the start symbol is reached.

## **Advantages-**

The advantages of operator precedence parsing are-

* The implementation is very easy and simple.
* The parser is quite powerful for expressions in programming languages.

## **Disadvantages-**

The disadvantages of operator precedence parsing are-

* The handling of tokens known to have two different precedence becomes difficult.
* Only small class of grammars can be parsed using this parser.

## **Important Note-**

* In practice, operator precedence table is not stored by the operator precedence parsers.
* This is because it occupies the large space.
* Instead, operator precedence parsers are implemented in a very unique style.
* They are implemented using operator precedence functions.

## **Operator Precedence Functions-**

|  |
| --- |
| Precedence functions perform the mapping of terminal symbols to the integers. |

* To decide the precedence relation between symbols, a numerical comparison is performed.
* It reduces the space complexity to a large extent.

## **PRACTICE PROBLEMS BASED ON OPERATOR PRECEDENCE PARSING-**

## **Problem-01:**

Consider the following grammar-

E → EAE | id

A → + | x

Construct the operator precedence parser and parse the string id + id x id.

## **Solution-**

### **Step-01:**

We convert the given grammar into operator precedence grammar.

The equivalent operator precedence grammar is-

**E → E + E | E x E | id**

### **Step-02:**

The terminal symbols in the grammar are { id, + , x , $ }

We construct the operator precedence table as-

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | **id** | **+** | **x** | **$** |
| **id** |  | > | > | > |
| **+** | < | > | < | > |
| **x** | < | > | > | > |
| **$** | < | < | < |  |

#### **Operator Precedence Table**

### **Parsing Given String-**

Given string to be parsed is **id + id x id**.

We follow the following steps to parse the given string-

### **Step-01:**

We insert $ symbol at both ends of the string as-

**$ id + id x id $**

We insert precedence operators between the string symbols as-

**$ < id > + < id > x < id > $**

### **Step-02:**

We scan and parse the string as-

$ **< id >** + < id > x < id > $

$ E + **< id >** x < id > $

$ E + E x **< id >** $

$ E + E x E $

$ + x $

$ < + **< x >** $

$ **< + >** $

$ $

## **Problem-02:**

Consider the following grammar-

S → ( L ) | a

L → L , S | S

Construct the operator precedence parser and parse the string ( a , ( a , a ) ).

## **Solution-**

The terminal symbols in the grammar are { ( , ) , a , , }

We construct the operator precedence table as-

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | **a** | **(** | **)** | **,** | **$** |
| **a** |  | > | > | > | > |
| **(** | < | > | > | > | > |
| **)** | < | > | > | > | > |
| **,** | < | < | > | > | > |
| **$** | < | < | < | < |  |

#### **Operator Precedence Table**

### **Parsing Given String-**

Given string to be parsed is **( a , ( a , a ) )**.

We follow the following steps to parse the given string-

### **Step-01:**

We insert $ symbol at both ends of the string as-

**$ ( a , ( a , a ) ) $**

We insert precedence operators between the string symbols as-

**$ < ( < a > , < ( < a > , < a > ) > ) > $**

### **Step-02:**

We scan and parse the string as-

$ < ( **< a >** , < ( < a > , < a > ) > ) > $

$ < ( S , < ( **< a >** , < a > ) > ) > $

$ < ( S , < ( S , **< a >** ) > ) > $

$ < ( S , **< ( S , S ) >** ) > $

$ < ( S , **< ( L , S ) >** ) > $

$ < ( S , **< ( L ) >** ) > $

$ **< ( S , S ) >** $

$ **< ( L , S ) >** $

$ **< ( L ) >** $

$ **< S >** $

$ $

## **Problem-03:**

Consider the following grammar-

E → E + E | E x E | id

1. Construct Operator Precedence Parser.
2. Find the Operator Precedence Functions.

## **Solution-**

The terminal symbols in the grammar are **{ + , x , id , $ }**

We construct the operator precedence table as-

**id + x $**

**id - > > >**

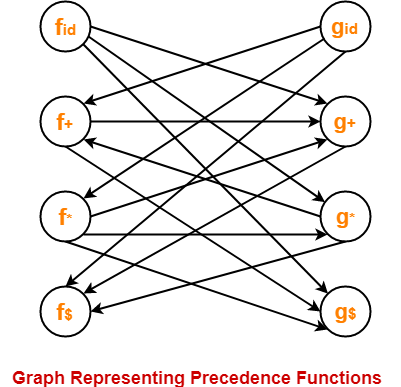
**+ < > < >**

**x < > > >**

**$ < < < -**

**Operator Precedence Table**

The graph representing the precedence functions is-



Here, the longest paths are-

* fid→ gx → f+ → g+ → f$
* gid→ fx → gx → f+ → g+ → f$

The resulting precedence functions are-

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

 Size of the table is **2n**.

**\*Introduction to YACC**

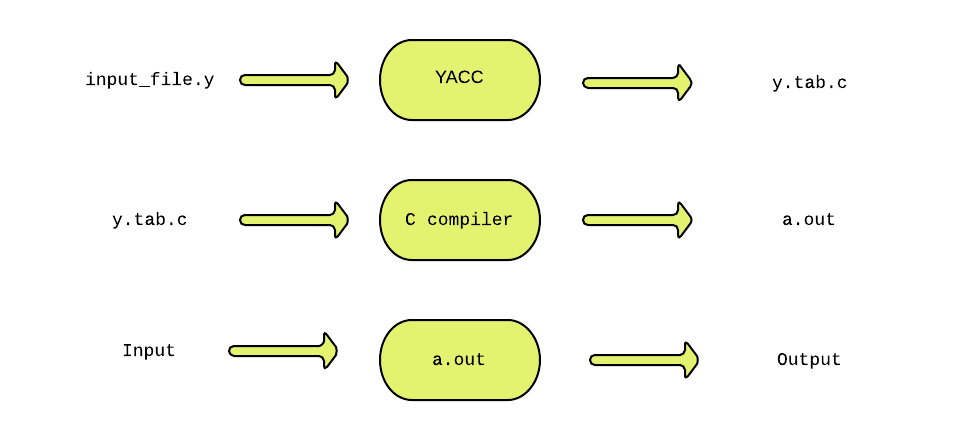
* YACC (Yet Another Compiler Compiler) is a tool used to generate a parser.
* YACC translates a given [Context Free Grammar (CFG)](https://silcnitc.github.io/yacc.html#navcfg) specifications (input in input\_file.y) into a C implementation (y.tab.c) of a corresponding [push down automaton](http://en.wikipedia.org/wiki/Pushdown_automaton)(i.e., a finite state machine with a stack).
* This C program when compiled, yields an executable parser.
* YACC stands for **Yet Another Compiler Compiler**.
* YACC provides a tool to produce a parser for a given grammar.
* YACC is a program designed to compile a LALR (1) grammar.
* It is used to produce the source code of the syntactic analyzer of the language produced by LALR (1) grammar.
* The input of YACC is the rule or grammar and the output is a C program.

These are some points about YACC:

**Input: A CFG- file.y**

**Output: A parser y.tab.c (yacc)**

* The output file "file.output" contains the parsing tables.
* The file "file.tab.h" contains declarations.
* The parser called the yyparse ().
* Parser expects to use a function called yylex () to get tokens.



The source SIL program is fed as the input to the generated parser ( a.out ). The *parser* checks whether the program satisfies the syntax specification given in the input\_file.y file.

**Input File: Definition Part:**

* The definition part includes information about the tokens used in the syntax definition:
* %token NUMBER

%token ID

* Yacc automatically assigns numbers for tokens, but it can be overridden by

%token NUMBER 621

* Yacc also recognizes single characters as tokens. Therefore, assigned token numbers should no overlap ASCII codes.
* The definition part can include C code external to the definition of the parser and variable declarations, within **%{** and **%}** in the first column.
* It can also include the specification of the starting symbol in the grammar:

%start nonterminal

**Input File: Rule Part:**

* The rules part contains grammar definition in a modified BNF form.
* Actions is C code in { } and can be embedded inside (Translation schemes).

**Input File: Auxiliary Routines Part:**

* The auxiliary routines part is only C code.
* It includes function definitions for every function needed in rules part.
* It can also contain the main() function definition if the parser is going to be run as a program.
* The main() function must call the function yyparse().

**Input File:**

* If yylex() is not defined in the auxiliary routines sections, then it should be included:

#include "lex.yy.c"

* YACC input file generally finishes with:

.y

**Output Files:**

* The output of YACC is a file named **y.tab.c**
* If it contains the **main()** definition, it must be compiled to be executable.
* Otherwise, the code can be an external function definition for the function **int yyparse()**
* If called with the **–d** option in the command line, Yacc produces as output a header file **y.tab.h** with all its specific definition (particularly important are token definitions to be included, for example, in a Lex input file).
* If called with the **–v** option, Yacc produces as output a file **y.output** containing a textual description of the LALR(1) parsing table used by the parser. This is useful for tracking down how the parser solves conflicts.

**Parser**:

A parser is a program that checks whether its input (viewed as a stream of tokens) meets a given grammar specification. The syntax of SIL can be specified using a Context Free Grammar. As mentioned earlier, YACC takes this specification and generates a parser for SIL.

## **The structure of YACC programs**

A YACC program consists of **three sections**: Declarations, Rules and Auxiliary functions. (Note the similarity with the structure of LEX programs).

DECLARATIONS

%%

RULES

%%

AUXILIARY FUNCTIONS

**Declarations**

The declarations section consists of two parts: (i) C declarations and (ii) YACC declarations . The C Declarations are delimited by **%{** and **%}**. This part consists of all the declarations required for the C code written in the *Actions* section and the ***Auxiliary functions*** section. YACC copies the contents of this section into the generated y.tab.c file without any modification.

The following example shows an abstract outline of the structure of the declarations part of a YACC program:

**Example :**

**/\* Beginning of Declarations part \*/**

**%{**

**/\*Beginning of C declarations\*/**

**/\*End of C declarations\*/**

**%}**

**/\*Beginning of YACC declarations \*/**

**/\*End of YACC declarations \*/**

**/\* End of Declarations Part \*/**

**%%**

The YACC declarations part comprises of declarations of tokens (usually returned by the lexical analyzer). The parser reads the tokens by invoking the function yylex()

**Rules**

A rule in a YACC program comprises of **two parts** (i) the production part and (ii) the action part. In this project, the syntax of SIL programming language will be specified in the form of a context free grammar. A rule in YACC is of the form:

production\_head : production\_body {action in C } ;

The following example shows an abstract outline of the structure of the rules part of a YACC program:

%%  
/\* Rules Section begins here \*/  
  
/\* Rules Section ends here \*/  
%%

**Yacc File (.y)**

%{

#include <ctype.h>

#include <stdio.h>

#define YYSTYPE double /\* double type for yacc stack \*/

%}

%%

Lines : Lines S '\n' { printf("OK \n"); }

| S '\n’

| error '\n' {yyerror("Error: reenter last line:");

yyerrok; };

S : '(' S ')’

| '[' S ']’

| /\* empty \*/ ;

%%

#include "lex.yy.c"

void yyerror(char \* s)

/\* yacc error handler \*/

{

fprintf (stderr, "%s\n", s);

}

int main(void)

{

return yyparse();

}

**Lex File (.l)**

%{

%}

%%

[ \t] { /\* skip blanks and tabs \*/ }

\n|. { return yytext[0]; }

%%

**For Compiling YACC Program:**

1. Write lex program in a file file.l and yacc in a file file.y
2. Open Terminal and Navigate to the Directory where you have saved the files.
3. type lex file.l
4. type yacc file.y
5. type cc lex.yy.c y.tab.h -ll
6. type ./a.out