# Unit 3
# Syntax Analyzer

…………………………………………………………………………………………………

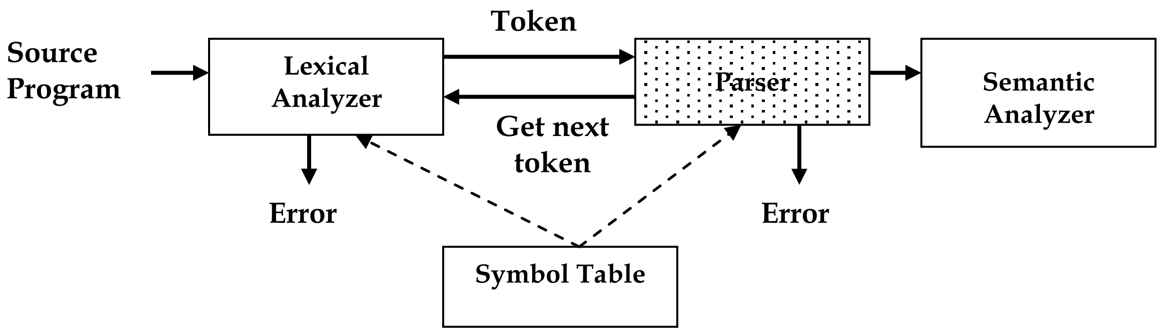
**Topics**

Syntax Analysis: Its role, Basic parsing techniques: Problem of Left Recursion, Left Factoring, Ambiguous Grammar, Top-down parsing, Bottom-up parsing, LR parsing.

…………………………………………………………………………………………………

## Syntax Analysis

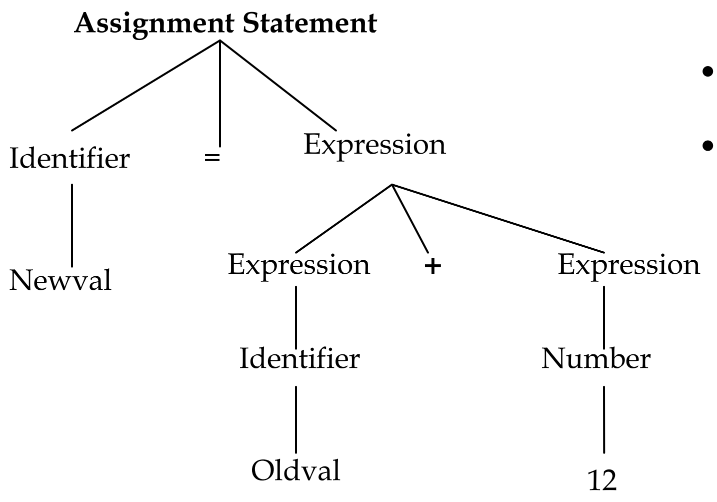
Syntax Analysis or Parsing is the second phase, i.e. after lexical analysis. It checks the syntactical structure of the given input, i.e. whether the given input is in the correct syntax or not. It does so by building a data structure, called a Parse tree or Syntax tree. The parse tree is constructed by using the pre-defined Grammar of the language and the input string. If the given input string can be produced with the help of the syntax tree, the input string is found to be in the correct syntax. If not, error is reported by syntax analyzer.



In this chapter, we shall learn the basic concepts used in the construction of a parser. We have seen that a lexical analyzer can identify tokens with the help of regular expressions and pattern rules. But a lexical analyzer cannot check the syntax of a given sentence due to the limitations of the regular expressions. Regular expressions cannot check balancing tokens, such as parenthesis. Therefore, this phase uses context-free grammar (CFG), which is recognized by push-down automata.

A **Syntax Analyzer** creates the syntactic structure (generally a parse tree) of the given source program. Syntax analyzer is also called the parser. Its job is to analyze the source program based on the definition of its syntax. It is responsible for creating a parse-tree of the source code.

Ex: newval:= oldval + 12



- In a parse tree all terminals are at leaves.
- All inner nodes are non-terminals in a context free grammar (CFG).

## Role of syntax analyzer

A syntax analyzer or parser takes the input from a lexical analyzer in the form of token streams. The parser analyzes the source code (token stream) against the production rules to detect any errors in the code. The output of this phase is a parse tree.

The parser obtains a string of tokens from the lexical analyzer and verifies that the string can be generated by the grammar for the source language. It has to report any syntax errors if occurs. The tasks of parser can be expressed as

- Analyzes the context free syntax
- Generates the parse tree
- Provides the mechanism for context sensitive analysis
- Determine the errors and tries to handle them

## Context Free Grammar

Generally most of the programming languages have recursive structures that can be defined by Context free grammar (CFG). Context-free grammars can generate context-free languages. They do this by taking a set of variables which are defined recursively, in terms of one another, by a set of production rules. Context-free grammars are named as such because any of the production rules in the grammar can be applied regardless of context—it does not depend on any other symbols that may or may not be around a given symbol that is having a rule applied to it. Context-free grammars have the following components:

- **A set of terminal symbols** which are the characters that appear in the language/strings generated by the grammar. Terminal symbols never appear on the left-hand side of the production rule and are always on the right-hand side.
- **A set of non-terminal symbols (or variables)** which are placeholders for patterns of terminal symbols that can be generated by the non-terminal symbols. These are the

symbols that will always appear on the left-hand side of the production rules, though they can be included on the right-hand side. The strings that a CFG produces will contain only symbols from the set of non-terminal symbols.

- **A set of production rules** which are the rules for replacing non-terminal symbols. Production rules have the following form: variable → string of variables and terminals.
- **A start symbol** which is a special non-terminal symbol that appears in the initial string generated by the grammar.

Mathematically, CFG can be defined as 4 – tuple (V, T, P, S), where,

- V: finite set of Variables or Non-terminals that are used to define the grammar denoting the combination of terminal or no-terminals or both
- T: Terminals, the basic symbols of the sentences. Terminals are indivisible units
- P: Production rule that defines the combination of terminals or non-terminals or both for particular non-terminal
- S: It is the special non-terminal symbol called start symbol

**Example 1:** Construct a CFG for the regular expression (0+1)*

Solution: The CFG can be given by,

Production rule (P)

$S \rightarrow 0S \mid 1S$

$S \rightarrow \varepsilon$

The rules are in the combination of 0's and 1's with the start symbol. Since (0+1)* indicates {ε, 0, 1, 01, 10, 00, 11 ...}. In this set, ε is a string, so in the rule, we can set the rule S → ε.

**Example 4:** Construct a CFG for the language L = anb2n where n>=1.

**Solution**: The string that can be generated for a given language is,

{abb, aabbbb, aaabbbbbb....}.

The grammar could be:

$S \rightarrow aSbb \mid abb$

## CFG: Notational Conventions

- Terminals are denoted by lower-case letters and symbols (single atoms) and bold strings (tokens)

  *a, b, c,…* $\in T$

  Specific terminals: 0, 1, id, +
- Non-terminals are denoted by *lower-case italicized* letters or upper-case letters symbols

  *A, B, C…* $\in N$

  Specific non-terminals: *expr*, *term*, *stmt*
- Production rules are of the form

  $A \rightarrow \alpha$, that is read as "A can produce α"

- Strings comprising of both terminals and non-terminals are denoted by Greek letters: α, β , etc

## Derivations

A derivation is basically a sequence of production rules, in order to get the input string. During parsing, we take two decisions for some sentential form of input:

- Deciding the non-terminal which is to be replaced.
- Deciding the production rule, by which, the non-terminal will be replaced.

To decide which non-terminal to be replaced with production rule, we can have two options.

### Left-most derivation

If the sentential form of an input is scanned and replaced from left to right, it is called left-most derivation.

### Right-most derivation

If we scan and replace the input with production rules, from right to left, it is known as right-most derivation.

**Example:** Production rules:

E → E + E

E → E * E

E → id

**Input string:** id + id * id

The left-most derivation is:

E → E * E

E → E + E * E

E → id + E * E

E → id + id * E

E → id + id * id

Notice that the left-most side non-terminal is always processed first.

The right-most derivation is:

E → E + E

E → E + E * E

E → E + E * id

E → E + id * id

E → id + id * id

## Parse Trees

A parse tree is a graphical depiction of a derivation. It is convenient to see how strings are derived from the start symbol. In parse tree internal nodes represent non-terminals and the leaves represent terminals. The start symbol of the derivation becomes the root of the parse tree.

**Example: Let's take a CFG with production rules,**

Production rules:

E → E + E

E → E * E

E → id

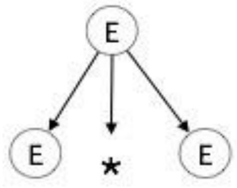
**Input string:** id + id * id

The left-most derivation is:

E → E * E
E → E + E * E
E → id + E * E
E → id + id * E
E → id + id * id

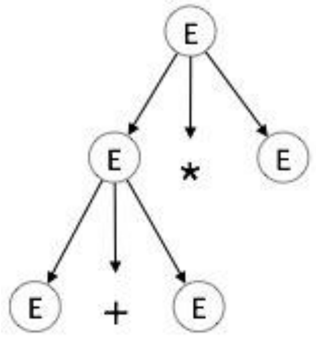
Step 1:

E → E * E



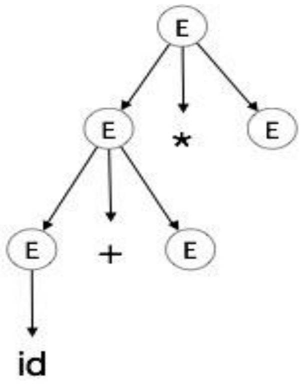
Step 2:

E → E + E * E



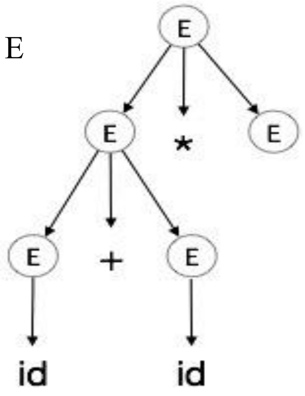
Step 3:

E → id + E * E

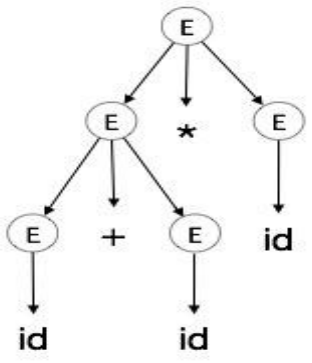


Step 4:

E → id + id * E



Step 5:
E → id + id * id



A parse tree depicts associativity and precedence of operators. The deepest sub-tree is traversed first; therefore the operator in that sub-tree gets precedence over the operator which is in the parent nodes.

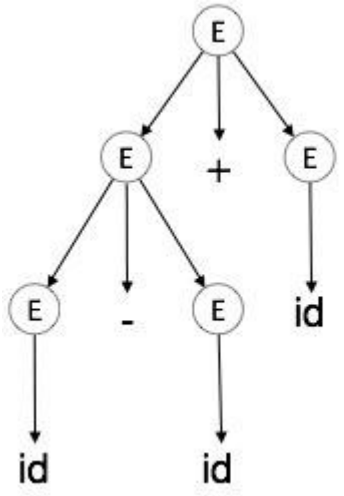
## Ambiguity of a grammar

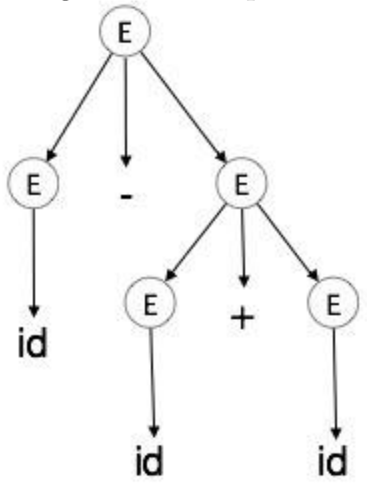
A grammar G is said to be ambiguous if it has more than one parse tree (left or right derivation) for at least one string. Also a grammar G is said to be ambiguous if there is a string w∈L(G) for which we can construct more than one parse tree rooted at start symbol of the production.

**Example: Let's take a grammar with production rules,**

E → E + E
E → E – E
E → id

For the string id + id – id, the above grammar generates two parse trees:

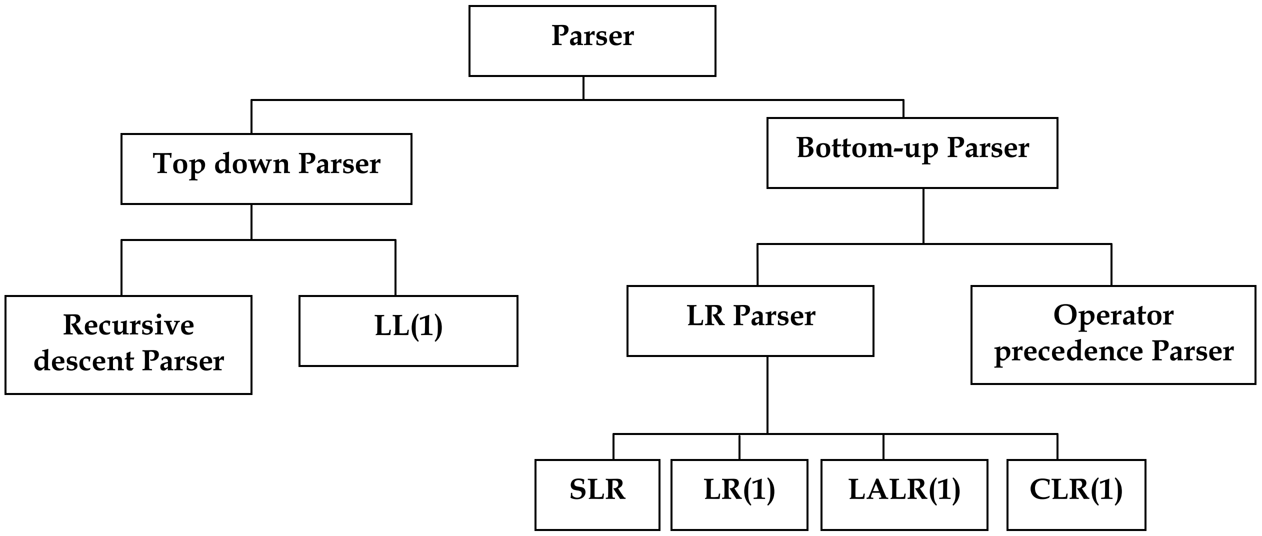




The language generated by an ambiguous grammar is said to be **inherently ambiguous**. Ambiguity in grammar is not good for a compiler construction. No method can detect and remove ambiguity automatically, but it can be removed by either re-writing the whole grammar without ambiguity, or by setting and following associativity and precedence constraints.

## Parsing

Parser is a compiler that is used to break the data into smaller elements coming from lexical analysis phase. A parser takes input in the form of sequence of tokens and produces output in the form of parse tree. Given a stream of input tokens, *parsing* involves the process of reducing them to a non-terminal. Parsing is of two types: top down parsing and bottom up parsing.



### 1. Top-down Parser

Top-down parser is the parser which generates parse for the given input string with the help of grammar productions by expanding the non-terminals i.e. it starts from the start symbol and ends on the terminals. It uses left most derivation. Further Top-down parser is classified into 2 types:

- Recursive descent parser and
- Non-recursive descent parser.

**Recursive descent parser**

It is also known as Brute force parser or the backtracking parser. It basically generates the parse tree by using brute force and backtracking. It is a general inefficient parsing technique, but not widely used.

**Example**: Consider the grammar,

$S \rightarrow aBc$

$B \rightarrow bc|b$

**Input string:** a b c

Parsed using Recursive-Descent parsing

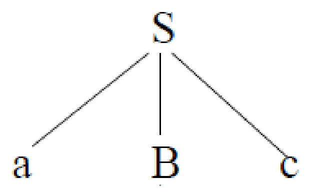
**Solution:**

| Input | Output | Rule used |
|---|---|---|
| abc | S | $S \rightarrow aBc$ |
| abc | aBc | Match symbol a |
| bc | Bc | B→bc |
| bc | bcc | Match symbol b |
| c | cc | Match symbol c |

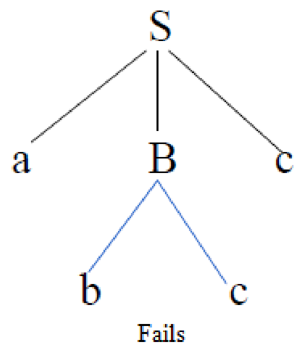
|  |  |  |
|---|---|---|
| Φ | c | Dead end, backtrack |
| bc | Bc | B → b |
| bc | bc | Match symbol b |
| c | c | Match symbol c |
| Φ | Φ | Accepted |

**Graphically,**

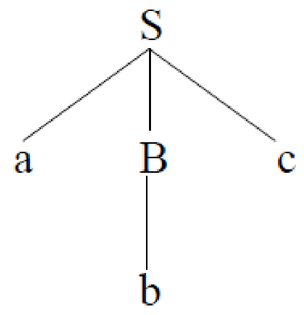
**Step 1:** The first rule $S \rightarrow aBc$ to parse S



Step 2: The next non-terminal is B and is parsed using production $B \rightarrow bc$ as,



Step 3: Which is false and now backtrack and use production $B \rightarrow b$ to parse for B



## Left Recursion

A grammar becomes left-recursive if it has any non-terminal 'A' whose derivation contains 'A' itself as the left-most symbol. Left-recursive grammar is considered to be a problematic situation for top-down parsers. Top-down parsers start parsing from the Start symbol, which in itself is non-terminal. So, when the parser encounters the same non-terminal in its derivation, it becomes hard for it to judge when to stop parsing the left non-terminal and it goes into an infinite loop. A grammar is **left recursive** if it has a non-terminal A such that there is a derivation.

$A \rightarrow A\alpha$ For some string α

Left recursion causes recursive descent parser to go into infinite loop. Top down parsing techniques cannot handle left – recursive grammars. So, we have to convert our left recursive grammar which is not left recursive. The left recursion may appear in a single derivation called immediate left recursion, or may appear in more than one step of the derivation. So, we have to convert our left-recursive grammar into an equivalent grammar which is non left-recursive.

**Example 1:** Immediate Left-Recursion

$A \rightarrow A\alpha \mid \beta$

$$\Big\Downarrow \text{ Eliminate immediate left recursion}$$

$A \rightarrow \beta A'$

$A' \rightarrow \alpha A' \mid \in$

In general,

$A \rightarrow A\alpha_1 \mid A\alpha_2 \mid A\alpha_3 \mid \ldots\ldots \mid A\alpha_n \mid \beta_1 \mid \beta_2 \mid \beta_3 \mid \ldots\ldots \mid \beta_n$

Where, $\beta_1$, $\beta_2$, …. $B_n$ do not start with A

Now eliminate immediate left recursion as,

$A \rightarrow \beta_1 A' \mid \beta_2 A' \mid \ldots. \mid \beta_n A'$

$A' \rightarrow \alpha_1 A' \mid \alpha_2 A' \mid \alpha_3 A' \mid \ldots\ldots \mid \alpha_n A' \mid \varepsilon$

**Example 1:** Eliminate left recursion from following grammar,

$E \rightarrow E+T \mid T$

$E \rightarrow T*F \mid F$

$F \rightarrow id \mid (E)$

**Solution:** Now eliminate left recursion as,

$E \rightarrow TE'$

$E' \rightarrow +TE' \mid \varepsilon$

$T \rightarrow FT'$

$T' \rightarrow *FT' \mid \varepsilon$

$F \rightarrow id \mid (E)$

## Non-Immediate Left-Recursion

By just eliminating the immediate left-recursion, we may not get a grammar which is not left-recursive.

**Example 2:** Let's take a grammar with non-immediate left recursion

$S \rightarrow Aa \mid b$

$A \rightarrow Sc \mid d$

This grammar is not immediately left-recursive but it is still left recursive,

So at first make immediate left recursive as,

$S \rightarrow Sca \mid da \mid b$

$A \rightarrow Sc \mid d$

Now, eliminate left recursion as,

$S \rightarrow daS' \mid bS'$

$S' \rightarrow caS' \mid \varepsilon$

$A \rightarrow Sc \mid d$

**Example 3:** Eliminate left recursion from following grammar,

$S \rightarrow Aa \mid b$

$A \rightarrow Ac \mid Sd \mid f$

Let us order the non terminals as S, A

**For S:**

- Don't enter the inner loop
- Also there is no immediate left recursion to remove in S as outer loop says

**For A:**

Replace, $A \rightarrow Sd$ with $A \rightarrow Aad \mid bd$

Then we have, $A \rightarrow Ac \mid Aad \mid bd \mid f$

Now, remove the immediate left recursions

$A \rightarrow bdA' \mid fA'$

$A' \rightarrow cA' \mid adA' \mid \varepsilon$

So, the resulting equivalent grammar with no left recursion is

$S \rightarrow Aa \mid b$

$A \rightarrow bdA' \mid fA'$

$A' \rightarrow cA' \mid adA' \mid \varepsilon$

## Left-Factoring

If more than one production rules of a grammar have a common prefix string, then the top-down parser cannot make a choice as to which of the production it should take to parse the string in hand. Simply, when a no terminal has two or more productions whose right-hand sides start with the same grammar symbols, then such a grammar is not LL(1) and cannot be used for predictive parsing. This grammar is called left factoring grammar.

**Example 1:** Let's take a grammar with productions are in left factoring,

$A \rightarrow \alpha\, \beta_1 \mid \alpha\, \beta_2 \mid \alpha\, \beta_3 \mid \ldots\ldots\ldots \mid \alpha\, \beta_n \mid \gamma$

Now eliminate left factoring as,

$A \rightarrow \alpha\, (\beta_1 \mid \beta_2 \mid \beta_3 \mid \ldots\ldots\ldots \mid \beta_n) \mid \gamma$

The resulting grammar with left factoring free is,

$A \rightarrow \alpha\, A' \mid \gamma$

$A' \rightarrow \beta_1 \mid \beta_2 \mid \beta_3 \mid \ldots\ldots\ldots \mid \beta_n$

**Example 2:** Eliminate left factorial from following grammar:

$S \rightarrow iEiS \mid iEiSiS \mid a$

$B \rightarrow b$

**Solution**: Now eliminate left factoring as,

$S \to iEiS\ (\varepsilon \mid iS) \mid a$

$B \to b$

The resulting grammar with left factorial free is,

$S \to iEiSS' \mid a$

$S' \to iS$

$B \to b$

## Non-recursive descent parser

A form of recursive-descent parsing that does not require any back-tracking is known as predictive parsing. It is also called as LL(1) parsing table technique since we would be building a table for string to be parsed. It has capability to predict which production is to be used to replace input string. To accomplish its tasks, predictive parser uses a look-ahead pointer, which points to next input symbols. To make parser back-tracking free, predictive parser puts some constraints on grammar and accepts only a class of grammar known as LL(k) grammar. Given an LL(1) grammar G = (N, T, P, S) construct a table M[A, a] for A ∈ N, a ∈ T and use a driver program with a stack. A table driven predictive parser has an input buffer, a stack, a parsing table and an output stream.

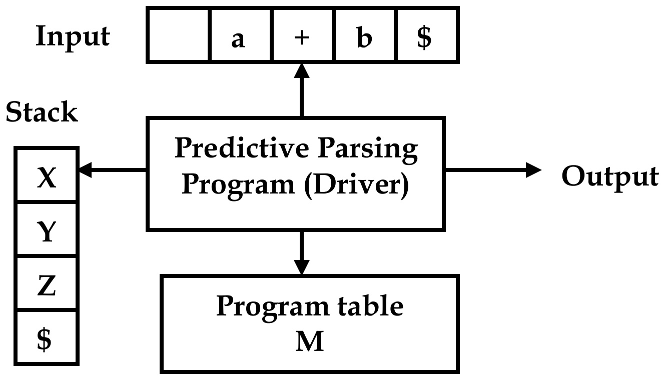


**Figure:** Model of a non-recursive predictive parser

- **Input buffer:** It contains the string to be parsed followed by a special symbol $.
- **Stack:** A stack contains a sequence of grammar symbols with $ on the bottom. Initially it contains the symbol $.
- **Parsing table:** It is a two dimensional array M[A, a] where 'A' is non-terminal and 'a' is a terminal symbol.
- **Output stream:** A production rule representing a step of the derivation sequence of the string in the input buffer.

**Procedure**

1. Initially the stack holds just the start symbol of the grammar.
2. At each step a symbol X is popped from the stack:

- if X is a terminal symbol then it is matched with lookahead and lookahead is advanced,
- if X is a non-terminal, then using lookahead and a parsing table a production is chosen and its right hand side is pushed onto the stack.

3. This process goes on until the stack and the input string become empty. It is useful to have an end_of_stack and an end_of_input symbols. We denote them both by $.

**Algorithm**

```
1. Set ip to the first symbol of input stream
2. Set the stack to $S where S is the start symbol of the grammar
3. Repeat
        Let X be the top stack symbol and 'a' be the symbol pointed by ip
            If X is a terminal or $ then,
                If X==a then,
                    Pop X from the stack and advance ip
                Else
                    Error()
            Else
                If M[X, a] = X→Y1,Y2, Y3,…….,Yk then,
                    Pop X from stack
                    Push Yk, Yk-1, ………..Y2, Y1 onto stack with Y1 on top
                Else
                    Error()
4. Until X=$
```

**Example 1:** Consider the grammar G given by:

$S \rightarrow aAa \mid BAa \mid \varepsilon$

$A \rightarrow cA \mid bA \mid \varepsilon$

$B \rightarrow b$

**Solution**: Parsing table be (we will see how to construct parsing table later)

| | **a** | **b** | **c** | **$** |
|---|---|---|---|---|
| **S** | $S \rightarrow aAa$ | $S \rightarrow BAa$ | | $S \rightarrow \varepsilon$ |
| **A** | $A \rightarrow \varepsilon$ | $A \rightarrow bA$ | $A \rightarrow cA$ | |
| **B** | | $B \rightarrow b$ | | |

Where, the empty slots correspond to error-entries. Consider the parsing of the word w = bcba

| **Stack** | **Remaining input** | **Action** |
|---|---|---|
| $S | bcba$ | choose $S \rightarrow Baa$ |
| $aAB | bcba$ | choose $B \rightarrow b$ |
| $aAb | bcba$ | match b |
| $aA | cba$ | choose $A \rightarrow cA$ |
| $aAc | cba$ | match c |
| $aA | ba$ | choose $A \rightarrow bA$ |

| | | |
|---|---|---|
| $aAb | ba$ | match b |
| $aA | a$ | choose A → ε |
| $a | a$ | match a |
| $ | $ | Accept |

**Example 2**: Given a grammar,

S → aBa

B → bB | ε

**Input**: abba

**Solution**: Parsing table be (we will see how to construct parsing table later)

| | **a** | **b** | **$** |
|---|---|---|---|
| **S** | S → aBa | S → BAa | |
| **B** | B → ε | B → bB | |

Table: LL(1) Parsing table

Where, the empty slots correspond to error-entries. Consider the parsing of the word w = abba

| **Stack** | **Remaining input** | **Action** |
|---|---|---|
| $S | abba$ | choose S → aBa |
| $aBa | abba$ | Mach a |
| $aB | bba$ | Choose B → bB |
| $aBb | bba$ | match b |
| $aB | ba$ | choose B → bB |
| $aBb | ba$ | Match b |
| $aB | a$ | choose B → ε |
| $a | a$ | Match a |
| $ | $ | Accept and successful completion |

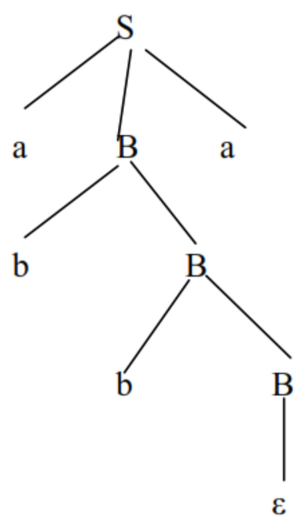
**Output:**

S → aBa
B → bB
B → bB
B → ε

So, the left most derivation is,

S ==> aBa ==> abBa ==> abbBa ==> abba

And the parse tree will be,



**Difference between Recursive Predictive Descent Parser and Non-Recursive Predictive Descent Parser:**

The main difference between recursive descent parsing and predictive parsing is that recursive descent parsing may or may not require backtracking while predictive parsing does not require any backtracking. Major differentiate between them are tabulated below:

| Recursive predictive descent parser | Non-recursive predictive descent parser |
|---|---|
| It is a technique which may or may not require backtracking process. | It is a technique that does not require any kind of back tracking. |
| It uses procedures for every non terminal entity to parse strings. | It finds out productions to use by replacing input string. |
| It is a type of top-down parsing built from a set of mutually recursive procedures where each procedure implements one of non-terminal s of grammar. | It is a type of top-down approach, which is also a type of recursive parsing that does not uses technique of backtracking. |
| It contains several small functions one for each non- terminals in grammar. | The predictive parser uses a look ahead pointer which points to next input symbols to make it parser back tracking free, predictive parser puts some constraints on grammar. |
| It accepts all kinds of grammars. | It accepts only a class of grammar known as LL(k) grammar. |

## Constructing LL(1) Parsing Table

The parse table construction requires two functions: FIRST and FOLLOW. For a string of grammar symbols α, FIRST(α) is the set of terminals that begin all possible strings derived from α. If α ==>* ε, then ε is also in FIRST(α). FOLLOW(A) for non terminal A is the set of terminals that can appear immediately to the right of A in some sentential form. If A can be the last symbol in a sentential form, then $ is also in FOLLOW(A).

A grammar G is suitable for LL(1) parsing table if the grammar is free from left recursion and left factoring.

$$\begin{tikzcd}\text{A grammar} \arrow[r, "\text{Eliminate Left recursion}"] & {} \arrow[r, "\text{Eliminate Left factor}"] & \text{Grammar is suitable for Predictive parsing (LL(1) grammar)}\end{tikzcd}$$

To compute LL (1) parsing table, at first we need to compute FIRST and FOLLW functions.

## Compute FIRST

**FIRST(α)** is a set of the terminal symbols which occur as first symbols in strings derived from α where α is any string of grammar symbols. To compute FIRST(A) for all grammar symbols A, apply the following rules until no more terminals or e can be added to any FIRST set:

1. If 'a' is terminal, then FIRST(a) is {a}.
2. If A→ε is a production, then add ε to FIRST(A).
3. For any non-terminal A with production rules A→ α1|α2|α3|…..|αn then
   FIRST(A) = FIRST(α1) U FIRST(α2) U FIRST(α3) U……….. U FIRST(αn)
4. If the production rule of the form, A→β1β2β3…..βn then,
   FIRST(A) = FIRST(β1β2β3…..βn)

**Example 1**: Find FIRST of following grammar symbols,

R → aS | (R) S

S → +RS |aRS | *S | ε

**Solution:**

FIRST(R) = {FIRST(aS) U FIRST((R)S)} = {a, ( }

FIRST(S) = {FIRST(+RS) U FIRST(aRS) U FIRST(*S) U FIRST(ε)}={+, a, *, ε }

FIRST(aS) = {a} FIRST((R)S) = {( }

FIRST(+RS) = {+} FIRST(aRS) = {a}

FIRST(*S) = {*} FIRST(ε) = {ε}

**Example 2**: Find FIRST of following grammar symbols,

E → TE'
E' → +TE'|ε
T → FT'
T' → *FT'|ε
F → (E)|id

**Solution:**

FIRST(F) = FIRST( ( ) U FIRST(id) = {(, id} FIRST(id) = {id}

FIRST(T') = {*, ε} FIRST(T) = FIRST(F) = {(, id}

FIRST(E') = {+, ε} FIRST(E) = FIRST(T) = {(, id}

FIRST(TE') = FIRST(T) = {(, id} FIRST(+TE') = FIRST(+) = {+}

FIRST(ε) = {ε}

FIRST(*FT′) = FIRST(*) = {*}

FIRST(FT′) = FIRST(F) = {(, id}

FIRST((E)) = FIRST( ( ) = { ( }

## Compute FOLLOW

FOLLOW (A) is the set of the terminals which occur immediately after (follow) the non-terminal A in the strings derived from the starting symbol.

FOLLOW(A) = {the set of terminals that can immediately follow non terminal A except ε}

**Rules:**

1. If A is a starting symbol of given grammar G then FIRST(A)={$}
2. For every production B → α A β, where α and β are any string of grammar symbols and A is non terminal, then everything in FIRST(β) except ε is placed on FOLLOW(A)
3. For every production B → α A, or a production B → α A β, FIRST(β) contains ε, then everything in FOLLOW(B) is added to FOLLOW(A)

**Example 1:** Compute FOLLOW of following grammar,

R → aS | (R) S

S → +RS |aRS | *S | ε

**Solution:**

FOLLOW(R) = {FIRST( )S) U FIRST(S) U FOLLOW(S)} = {$, ), +, a, *}

FOLLOW(S) = {FOLLOW(R)} = {$, ), +, a, *}

**Example 2:** Compute FOLLOW of following grammar,

E → TE′

E′ → +TE′ | ε

T → FT′

T′ → *FT′ | ε

F → (E) | id

**Solution:**

FOLLOW(E) = {FIRST( ) ) U $} = {$, ) }

FOLLOW(E′) = {FOLLOW(E) U FOLLOW(E′)} = {$, )}

FOLLOW(T) = {FIRST(E′) U FOLLOW(E′)} = {+, ), $}

FOLLOW(T′) = { FOLLOW(T)} = {+, ), $}

FOLLOW(F) = FIRST(T′) U FOLLOW(T′)} = {+, *, ), $}

## Constructing LL(1) Parsing Tables

If we can always choose a production uniquely by using FIRST and FOLLOW functions then this is called LL(1) parsing where the first L indicates the reading direction (Left-to –right) and second L indicates the derivation order left most and 1 indicates that there is a one-symbol look-ahead. The grammar that can be parsed using LL(1) parsing is called an LL(1) grammar.

**Algorithm**

For every production $A \rightarrow \alpha$ in the grammar:

1. If $\alpha$ can derive a string starting with a (i.e., for all a in FIRST($\alpha$ ),

   Table [A, a] = $A \rightarrow \alpha$

2. If $\alpha$ can derive the empty string $\varepsilon$, then, for all b that can follow a string derived from A (i.e., for all b in FOLLOW (A),

   Table [A, b] = $A \rightarrow \varepsilon$

**Example 1** Construct LL(1) parsing table of following grammar

$E \rightarrow T E'$
$E' \rightarrow + T E' \mid \varepsilon$
$T \rightarrow F T'$
$T' \rightarrow * F T' \mid \varepsilon$
$F \rightarrow ( E ) \mid id$

**Solution:** At first compute FIRST of above grammar as,

FIRST(F) ={(, id}
FIRST(T) ={(, id}
FIRST(E) ={(, id}
FIRST(+TE′) = {+}
FIRST(FT′) = {(, id}
FIRST(ε) = { ε }
FIRST(id) = {id}

FIRST(T′) ={*, ε }
FIRST(E′) ={+, ε }
FIRST(TE′) = {(, id}
FIRST(ε) = { ε }
FIRST(*FT′) = {*}
FIRST((E)) = {( }

Now compute FOLLOW of given function as,

FOLLOW(E) ={$, )}
FOLLOW(T) ={+, $, )}
FOLLOW(F) ={+, *, $, )}

FOLLOW(E′) ={$, )}
FOLLOW(T′) ={+, $, )}

Now do for every production

| | | |
|---|---|---|
| $E \rightarrow TE'$ | FIRST(TE′) = {(, id} | $E \rightarrow TE'$ into M[E, (] and M[E, id] |
| $E' \rightarrow +TE'$ | FIRST(+TE′) = {+} | $E' \rightarrow +TE'$ into M[E′, +] |
| $E' \rightarrow \varepsilon$ | FIRST(ε) = { ε } | but since ε in $E' \rightarrow \varepsilon$ into M[E′, $] and M[E′, )] |
| | FIRST(ε) and FOLLOW(E′) = {$, )} | |
| $T \rightarrow FT'$ | FIRST(FT′) = {(, id} | $T \rightarrow FT'$ into M[T, (] and M[T, id] |
| $T' \rightarrow *FT'$ | FIRST(*FT′) = {*} | $T' \rightarrow *FT'$ into M[T′, *] |
| $T' \rightarrow \varepsilon$ | FIRST(ε) = { ε } but since ε in | $T' \rightarrow \varepsilon$ into M[T′, $], M[T′, )] and |
| | FIRST(ε) and FOLLOW(T′) = {$, ), +} | |
| $F \rightarrow (E)$ | FIRST((E)) = {( } | $F \rightarrow (E)$ into M[F, ( ] |
| $F \rightarrow id$ | FIRST(id) = {id} | $F \rightarrow id$ into M[F, id] |

Now, the final table looks like

| **Non-terminals** | **Terminal Symbols** | | | | | |
|---|---|---|---|---|---|---|
| | **+** | ***** | **(** | **)** | **id** | **$** |
| E | | | E → TE′ | | E → TE′ | |
| E′ | E′ → +TE′ | | | E′ → ε | | E′ → ε |
| T | | | T → FT′ | | T → FT′ | |
| T′ | T′ → ε | T′ → *FT′ | | T′ → ε | | T′ → ε |
| F | | | F → (E) | | F → id | |

As you can see that all the null productions are put under the follow set of that symbol and all the remaining productions are lie under the first of that symbol.

**Input:** id + id * id

| **Stack** | **Input** | **Output** |
|---|---|---|
| $E | id + id * id $ | E → TE′ [see above M[E, id] = E → TE′] |
| $E′T | id + id * id $ | T → FT′ |
| $E′T′F | id + id * id $ | F → id |
| $E′T′id | id + id * id $ | Match id, pop |
| $E′T′ | + id * id $ | T′ → ε |
| $E′ | + id * id $ | E′ → +TE′ |
| $E′T+ | + id * id $ | Match +, pop |
| $E′T | id * id $ | T → FT′ |
| $E′T′F | id * id $ | F → id |
| $E′T′id | id * id $ | Match id, pop |
| $E′T′ | * id $ | T′ → *FT′ |
| $E′T′F* | * id $ | Match *, pop |
| $E′T′F | id $ | F → id |
| $E′T′id | id $ | Match id, pop |
| $E′T′ | $ | T′ → ε |
| $E′ | $ | E′ → ε |
| $ | $ | Accept |

**Example 2:** Constructing LL(1) Parsing Tables for following grammar

S→iEtSS′ | a

S′→eS | ε

E→b

**Solution:** At first compute FIRST as,

| | |
|---|---|
| FIRST(S) = {i, a} | FIRST(S′) = {e, ∈} |
| FIRST(E) = {b} | FIRST(iEtSS′) = {i} |
| FIRST(a) = {a} | FIRST(eS) = {e} |
| FIRST(b) = {b} | FIRST(∈) = {∈} |

Now Compute FOLLOW as,

FOLLOW(S) = {FIRST(S′)} = {e, $}

FOLLOW(S′) = {FOLLOW(S)} = {e, $}

FOLLOW(E) = {FIRST(tSS′)} = {t}

Now construct LL(1) parsing table as,

| **Non-terminals** | **Terminal Symbols** | | | | | |
|---|---|---|---|---|---|---|
| | **i** | **t** | **a** | **e** | **b** | **$** |
| S | $S \rightarrow iEtSS'$ | | $S \rightarrow a$ | | | |
| S′ | | | | $S' \rightarrow eS$<br>$S' \rightarrow \varepsilon$ | | $S' \rightarrow \varepsilon$ |
| E | | | | | $E \rightarrow b$ | |

Here, we can see that there are two productions into the same cell. Hence, this grammar is not feasible for LL(1) Parser.

**Example 3:** Test whether following grammar is feasible for LL(1) parsing or not??

$S \rightarrow A \mid a$

$A \rightarrow a$

**Solution:** At first compute FIRST as,

FIRST(S) = {a} FIRST(A) = {a}

Now calculate FOLLOW as,

FOLLOW(S) = {$} FOLLOW(A) = FOLLOW(S) = {$}

Now, construct LL(1) parsing table as,

| **Non-terminals** | **Terminal Symbols** | |
|---|---|---|
| | **a** | **$** |
| S | $S \rightarrow A$, $S \rightarrow a$ | |
| A | $A \rightarrow a$ | |

Here, we can see that there are two productions into the same cell. Hence, this grammar is not feasible for LL(1) Parser.

**Example 4:** Consider the following grammar,

$S \rightarrow L = R$

$S \rightarrow R$

$L \rightarrow *R$

$L \rightarrow id$

$R \rightarrow L$

At first compute FIRST and FOLLOW then construct LL(1) parsing table.

Solution: At first compute FIRST as,

FIRST(L) = {*, id} FIRST(R) = FIRST(L) = {*, id}

FIRST(S) = {FIRST(L) U FIRST(R)} = {*, id}

Now compute FOLLOW of each of the non-terminals as,

FOLLOW(S) = {$}

FOLLOW(L) = {FIRST(=R) U FOLLOW(R)} = {= U FOLLOW(S)} = {=, $}

FOLLOW(R) = {FOLLOW(S) U FOLLOW(L)} = {$, =}

Now construct LL(1) parsing table as,

| **Non-terminals** | **Terminal Symbols** | | | |
|---|---|---|---|---|
| | **=** | ***** | **id** | **$** |
| S | | $S \rightarrow L = R$<br>$S \rightarrow R$ | $S \rightarrow L = R$<br>$S \rightarrow R$ | |
| L | | $L \rightarrow *R$ | $L \rightarrow id$ | |
| R | | $R \rightarrow L$ | $R \rightarrow L$ | |

Here, we can see that there are two productions into the same cell. Hence, this grammar is not feasible for LL(1) Parser.

## LL(1) Grammars

A context-free grammar G = (V, T, P, S) whose parsing table has no multiple entries is said to be LL(1) grammar. In the name LL(1),

- the first L stands for scanning the input from left to right,
- the second L stands for producing a leftmost derivation and
- The 1 stands for using one input symbol of lookahead at each step to make parsing action decision.

A left recursive, left factored and ambiguous grammar cannot be a LL(1) grammar (i.e. left recursive, not left factored and ambiguous grammar may have multiply –defined entries in parsing table)

**Properties of LL(1) Grammars**

A grammar G is LL(1) if and only if the following conditions hold for two distinctive production rules $A \rightarrow \alpha$ and $A \rightarrow \beta$

- Both α and β cannot derive strings starting with same terminals.
- At most one of the α and β can drive to ε
- If β can drive to ε, then α cannot drive to any string staring with a terminal in FOLLOW(A)
- No Ambiguity
- No Recursion

**Example:** let's take following production rules,

| Grammar | Not LL(1) Because |
|---|---|
| S → S a \| a | Left Recursive |
| S → a S \| a | FIRST(a S) ∩ FIRST(a) ≠ ∅ |
| S → a R \| ε, R → S \| ε | **For R:** S ⇒* ε and ε ⇒* ε |
| S → a R a, R → S \| ε | **For R:** FIRST(S) ∩ FOLLOW(R) ≠ ∅ |

**Example 5:** Show that given grammar is not LL(1)

S→aA | bAc

A →c | ε

**Solution:** At first calculate FIRST as,

FIRST(S) = {a, b}　　FIRST(A) = {C, ε}

FIRST(aA) = {a}　　FIRST(bAc) = {b}

FIRST(c) = {c}　　FIRST(ε) = {ε}

Now calculate FOLLOW as,

FOLLOW(S) = { $ }　　FOLLOW(A) = {$, c }

Now construct LL(1) parsing table as,

| Non-terminals | Terminal Symbols | | | |
|---|---|---|---|---|
| | a | b | c | $ |
| S | S → aA | S → bAc | | |
| A | | | A → c<br>A → ε | A → ε |

Here, we can see that there are two productions into the same cell. Hence, this grammar is not feasible for LL(1) Parser.

## 2. Bottom-up Parser

Bottom-up parsing constructs a parse tree for an input string beginning at the leaves and working up towards the root. To do so, bottom-up parsing tries to find a rightmost derivation of a given string backwards.

Bottom-up Parser is the parser which generates the parse tree for the given input string with the help of grammar productions by compressing the non-terminals i.e. it starts from non-terminals and ends on the stat symbol. It uses reverse of the right most derivation. Bottom up parsing is classified in to various parsing. These are as follows:

1. Shift-Reduce Parsing
2. Operator Precedence Parsing
3. Table Driven LR Parsing
    a. LR( 1 )
    b. SLR( 0 )

c. CLR ( 1 )
d. LALR( 1 )

**Basic terminologies used in bottom up parsing**

**Reduction**

The process of replacing a substring by a non-terminal in bottom-up parsing is called reduction. It is a reverse process of production. E.g. let's take a production rule S→aA

Here, if replacing aA by S then such a grammar is called reduction.

**Example:** Consider the grammar

S → aABe
A → Abc | b
B → d

Now, the sentence **abbcde** can be reduced to S as follows

abbcde
aAbcde (replacing b by using A → b)
aAde (replacing Abc by using A → Abc)
aABe (replacing d by using B → d)
S this is the starting symbol of the grammar.

**Shift-Reduce Parsing**

A shift reduce parser tries to reduce the given input string into the starting symbol. At each reduction step, a substring of the input matching to the right side of a production rule is replaced by non – terminal at the left side of that production rule. If the substring is chosen correctly, the rightmost derivation of that string is created in reverse order. Simply the process of reducing the given input string into the starting symbol is called shift-reduce parsing.

A string $\xrightarrow[\text{Reduced to}]{\hspace{6em}}$ the starting symbol

**Handle**

A substring that can be replaced by a non-terminal when it matches its right sentential form is called a handle. If the grammar is unambiguous, then every right-sentential form of the grammar has exactly one handle. If the grammar is unambiguous, then every right – sentential form of the grammar has exactly one handle.

**Example 1:** A Shift-Reduce Parser with Handle

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

**String:** id + id * id

| Right-Most Sentential Form | Reduction Production | Handle |
|---|---|---|
| $\underline{id}$ + id * id | F → id | id |
| $\underline{F}$ + id * id | T → F | F |
| $\underline{T}$ + id * id | E → T | T |
| E + $\underline{id}$ * id | F → id | id |
| E + $\underline{F}$ * id | T → F | F |
| E + T * $\underline{id}$ | F → id | id |
| E + $\underline{T * F}$ | T → T * F | T * F |
| $\underline{E + T}$ | E → E + T | E + T |
| E | | |

## Stack Implementation of Shift-Reduce Parser

Like a table-driven predictive parser, a bottom-up parser makes use of a stack to keep track of the position in the parse and a parsing table to determine what to do next. Shift reduce parsing is a process of reducing a string to the start symbol of a grammar. It uses a stack to hold the grammar and an input tape to hold the string. There are mainly following four basic operations are used in shift reduce parser,

- **Shift:** This involves moving of symbols from input buffer onto the stack.
- **Reduce:** If the handle appears on top of the stack then, its reduction by using appropriate production rule is done i.e. RHS of production rule is popped out of stack and LHS of production rule is pushed onto the stack.
- **Accept:** If only start symbol is present in the stack and the input buffer is empty then, the parsing action is called accept. When accept action is obtained, it is means successful parsing is done.
- **Error:** This is the situation in which the parser can neither perform shift action nor reduce action and not even accept action.

**Algorithm**

```
1. Initially stack contains only the sentinel $, and input buffer contains the input string w$.
2. While stack not equal to $S do
    a. While there is no handle at the top of the stack, do shift input buffer and push the
       symbol onto the stack.
    b. If there is a handle on the top of the stack, then pop the handle and reduce the
       handle with its non – terminal and push it onto stack
3. Done
```

**Example 1:** Use the following grammar

S → S+S | S*S| (S) | id

Perform Shift Reduce parsing for input string: id + id + id

| **Stack** | **Input Buffer** | **Parsing Action** |
|---|---|---|
| $ | id+id+id$ | Shift |
| $id | +id+id$ | Reduce by S → id |
| $S | +id+id$ | Shift |
| $S+ | id+id$ | Shift |
| $S+id | +id$ | Reduce by S → id |
| $S+S | +id$ | Shift |
| $S+S+ | id$ | Shift |
| $S+S+id | $ | Reduce by S → id |
| $S+S+S | $ | Reduce by S → S+S |
| $S+S | $ | Reduce by S → S+S |
| $S | $ | Accept |

**Example 2:** Use the following grammar

E → E + T | T

T → T * F | F

F → (E) | id

**Input string:** id + id * id

| **Stack** | **Input Buffer** | **Parsing Action** |
|---|---|---|
| $ | id + id * id $ | Shift id |
| $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 id |
| $E + id | * id $ | Reduce by F → id |
| $E + F | * id $ | Reduce by T → F |
| $E + T | * id $ | Shift * (OR Reduce by E → T) CONFLICT |
| $E + T * | id $ | Shift id |
| $E + T * id | $ | Reduce by F → id |
| $E + T * F | $ | Reduce by T → T * F |
| $E + T | $ | Reduce by E → E + T |
| $E | $ | Accept |

**Example 3:** Consider the following grammar

S → (L) | a

L → L, S | S

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

| **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 |

**Example 4:** 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.

| **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 |

**Example 5:** Considering the string “10201”, design a shift-reduce parser for the following grammar:

S → 0S0 | 1S1 | 2

| 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 |

**Example 6:** Consider the following grammar:

E → E – E

E → E * E

E → id

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

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

## Conflicts in Shift – Reduce Parsing

Some grammars cannot be parsed using shift-reduce parsing and result in conflicts. There are two kinds of shift-reduce conflicts:

**Shift / Reduce Conflict**

Here, the parser is not able to decide whether to shift or to reduce.

**Example:** if A → ab

A→ abcd, and the stack contains $ab, and the input buffer contains cd$, the parser cannot decide whether to reduce $ab to $A or to shift two more symbols before reducing.

**Reduce / Reduce Conflict**

Here, the parser cannot decide which sentential form to use for reduction.

For example: if A → bc and B → abc and the stack contains $abc, the parser cannot decide whether to reduce it to $aA or to $B

## Operator Grammar

Operator precedence grammar is kinds of shift reduce parsing method. It is applied to a small class of operator grammars. A grammar that is used to define mathematical operators is called an operator grammar or operator precedence grammar. Such grammars have the restriction that no production has either an empty right-hand side (null productions) or two adjacent non-terminals in its right-hand side.

**Example:** This is an example of operator grammar:

E → E+E | E*E | id

However, the grammar given below is not an operator grammar because two non-terminals are adjacent to each other:

S → SAS | a

A → bSb | b

We can convert it into an operator grammar, though:

S → SbSbS | SbS | a

A → bSb | b

## Operator precedence parsing

Operator precedence can only established between the terminals of the grammar. It ignores the non-terminal. An operator precedence parser is a bottom-up parser that interprets an operator grammar. This parser is only used for operator grammars. Ambiguous grammars are not allowed in any parser except operator precedence parser. There are two methods for determining what precedence relations should hold between a pair of terminals:

1. Use the conventional associativity and precedence of operator.
2. The second method of selecting operator-precedence relations is first to construct an unambiguous grammar for the language, a grammar that reflects the correct associativity and precedence in its parse trees.

**Defining Precedence Relations:**

The precedence relations are defined using the following rules:

**Rule 1:**

- 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 2:**

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

**Rule 3:** 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 1**: 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 2:**

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

**Step 3:**

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

**Step 4:**

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

**Step 5:**

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

**Example 1:** Consider the following grammar-

E → EAE | id

A → + | *

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

**Solution:**

**Step 1:** We convert the given grammar into operator precedence grammar.

The equivalent operator precedence grammar is-

E → E + E | E * E | id

**Step 2:** The terminal symbols in the grammar are {id, + , * , $ }

We construct the operator precedence table as:

|  | **Id** | **+** | ***** | **$** |
|---|---|---|---|---|
| **Id** |  | > | > | > |
| **+** | < | > | < | > |
| ***** | < | > | > | > |
| **$** | < | < | < |  |

Operator Precedence Table

**Parsing Given String:** id + id * id.

We follow the following steps to parse the given string:

**Step 1:** We insert $ symbol at both ends of the string as-

$ id + id * id $

We insert precedence operators between the string symbols as-

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

**Step 2:** We scan and parse the string as:

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

$ E + < id > * < id > $

$ E + E * < id > $

$ E + E * E $

$ + * $

$ < + < * > $

$ < + > $

$ $

## 3. LR parser

LR parsing is one type of bottom up parsing. It is used to parse the large class of grammars. In the LR parsing, "L" stands for left-to-right scanning of the input. "R" stands for constructing a right most derivation in reverse. "K" is the number of input symbols of the look ahead used to make number of parsing decision. LR parsing is divided into four parts:

- SLR (Simple LR parser)
- LR (Most general LR parser)
- LALR (Intermediate LR parser,
- CLR (Canonical Lookahead)

SLR, LR and LALR work same (they used the same algorithm), only their parsing tables are different.

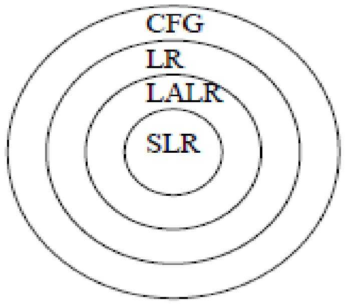


**Figure:** Scope of various types of grammars

**LR Parsers: General Structure**

The LR algorithm requires stack, input, output and parsing table. In all type of LR parsing, input, output and stack are same but parsing table is different.

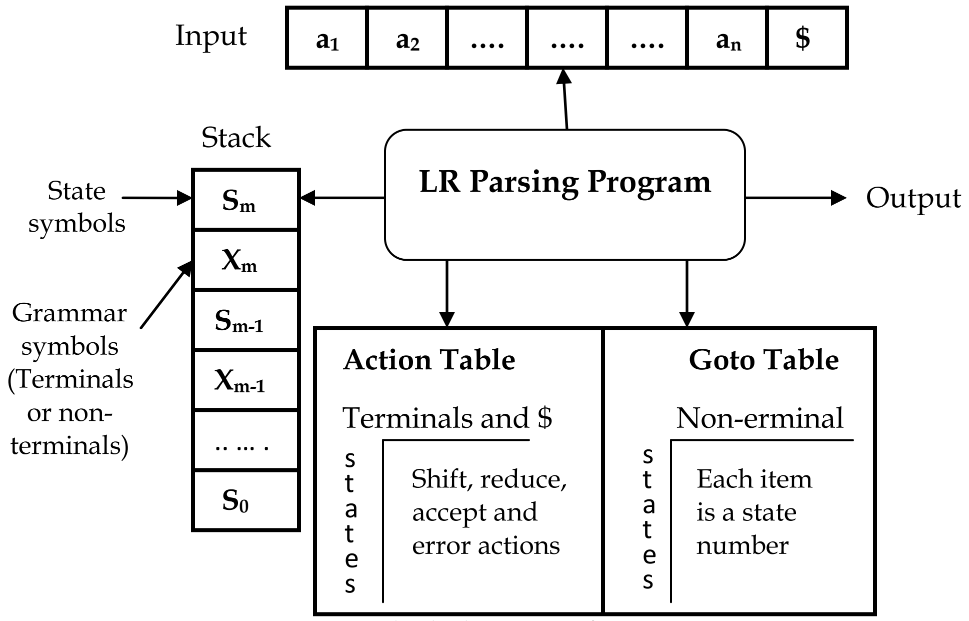


**Figure:** Block diagram of LR parser

The LR algorithm requires stack, input, output and parsing table. In all type of LR parsing, input, output and stack are same but parsing table is different. Input buffer is used to indicate end of input and it contains the string to be parsed followed by a $ Symbol. A stack is used to contain a sequence of grammar symbols with a $ at the bottom of the stack. Parsing table is a two dimensional array. It contains two parts: Action part and Go To part.

**Constructing SLR Parsing Table**

SLR parsers are the simplest class of LR parsers. Constructing a parsing table for action and **goto** involves building a state machine that can identify handles. For building a state machine, we need to define three terms: **Canonical collection of LR item**, **closure** and **goto operations**.

## Basic terminologies used for LR parsing table

### Augmented grammar

If G is a grammar with start symbol S, then the augmented grammar G′ of G is a grammar with a new start symbol S′ and production S′→S.

**Example:** the grammar,

$E \rightarrow E + T \mid T$

$T \rightarrow T * F \mid F$

$F \rightarrow (\, E\, ) \mid \mathbf{id}$

Its augmented grammar is;

$E' \rightarrow E$

$E \rightarrow E + T \mid T$

$T \rightarrow T * F \mid F$

$F \rightarrow (\, E\, ) \mid \mathbf{id}$

### LR(0) Item

An 'item' (LR(0) item) is a production rule that contains a dot (•) somewhere in the right side of the production. For example, the production $A \rightarrow \alpha\, A\, \beta$ has four items:

$A \rightarrow \bullet\alpha\, A\, \beta$

$A \rightarrow \alpha\bullet A\, \beta$

$A \rightarrow \alpha\, A\bullet\beta$

$A \rightarrow \alpha\, A\, \beta\bullet$

The production $A \rightarrow \varepsilon$, generates only one item $A \rightarrow \bullet$. An item represented by a pair of integers, the first giving the production and second the position of the dot. An item indicates how much of a production we have seen at a given point in the parsing process.

### Closure Operation

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

1. Initially, every LR(0) item in I is added to closure(I).
2. If $A \rightarrow \alpha\bullet B\beta$ is in closure(I) and $B \rightarrow \gamma$ is a production rule of G then add $B \rightarrow \bullet\gamma$ in the closure(I) repeat until no more new LR(0) items added to closure(I).

**Example:** Consider a grammar:

$E \rightarrow E + T \mid T$

$T \rightarrow T * F \mid F$

$F \rightarrow (E) \mid id$

Its augmented grammar is;

$E' \rightarrow E$

$E \rightarrow E + T \mid T$

T → T * F | F

F → (E) | id

If I = {[E′ → •E]}, then closure(I) contains the items,

E′ → •E

E → •E + T

E → •T

T → •T * F

T → •F

F → • (E)

F → •id

## Goto Operation

If I is a set of LR(0) items and X is a grammar symbol (terminal or non-terminal), then goto(I, X) is defined as follows:

- If A → 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) = closure({[E′ → E •, E → E • + T]}) = {E′ → E•,  E → E•+T}

goto(I, T) = {E → T•, T → T•*F}

goto(I, F) = {T → F•}

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

goto(I, id) = {F → id•}

## Canonical LR(0) collection

An LR (0) item is a production G with dot at some position on the right side of the production. LR(0) items is useful to indicate that how much of the input has been scanned up to a given point in the process of parsing. In the LR (0), we place the reduce node in the entire row. A collection of sets of LR(0) items is called canonical LR(0) collection. To construct canonical LR(0) collection for a grammar we require augmented grammar and closure & goto functions.

## Algorithm

```
1. Start
2. Augment the grammar by adding production S′ → S
3. C = {closure({S′→.S})}
4. Repeat the followings until no more set of LR(0) 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
```

5. Repeat step 4 until no new sets of items are added to C
6. Stop

**Example 1**: Find canonical collection of LR(0) items of following grammar

$C \rightarrow AB$
$A \rightarrow a$
$B \rightarrow a$

**Solution:** The augmented grammar of given grammar is,

$C' \rightarrow C$
$C \rightarrow AB$
$A \rightarrow a$
$B \rightarrow a$

Next, we obtain the canonical collection of sets of LR(0) items, as follows,

$I_0$ = closure ({C′ → •C}) = {C′ → •C, C → •AB, A → •a}

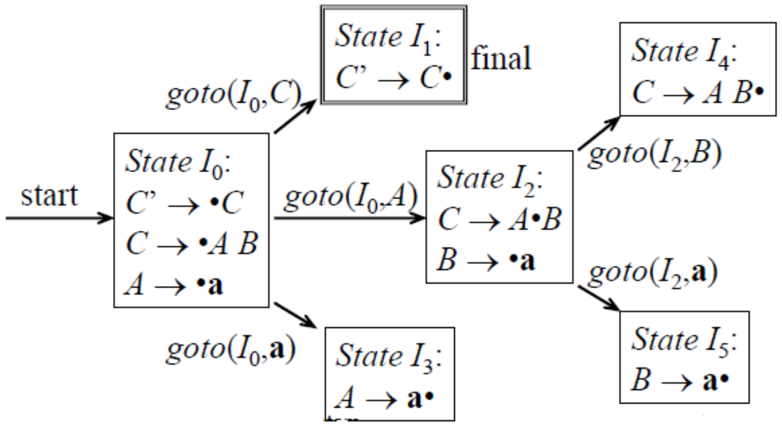
$I_1$ = goto($I_0$, C) = closure(C′ → C•) = { C′ → C•}

$I_2$ = goto($I_0$, A) = closure(C → A•B) = { C → A•B, B → •a}

$I_3$ = goto($I_0$, a) = closure(A → a•) = { A → a• }

$I_4$ = goto($I_2$, B) = closure(C → AB•) = { C → AB•}

$I_5$ = goto($I_2$, a) = closure(B → a•) = { B → a•}


State $I_1$: $C' \rightarrow C\bullet$
final
State $I_4$: $C \rightarrow A\,B\bullet$
goto($I_0$,C)
goto($I_2$,B)
start
State $I_0$: $C' \rightarrow \bullet C$, $C \rightarrow \bullet A\,B$, $A \rightarrow \bullet\mathbf{a}$
goto($I_0$,A)
State $I_2$: $C \rightarrow A\bullet B$, $B \rightarrow \bullet\mathbf{a}$
goto($I_2$,**a**)
goto($I_0$,**a**)
State $I_3$: $A \rightarrow \mathbf{a}\bullet$
State $I_5$: $B \rightarrow \mathbf{a}\bullet$

## Constructing SLR Parsing Tables

### Algorithm

1. Construct the canonical collection of sets of LR(0) items for G′.

   $C \leftarrow \{I_0, I_1 \ldots\ldots I_n\}$

2. Create the parsing action table as follows
   - If A → α•aβ is in $I_i$ and goto($I_i$, a) = $I_j$ then set action[i , a] = shift j.
   - If A → α• is in $I_i$, then set action[i, a] to "reduce A → α" for all 'a' in FOLLOW(A) Where, A≠S′
   - If S′→ S• is in $I_i$, then action[i, $] = 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 $S' \rightarrow \bullet S$

**Example 1**: Construct SLR parsing table of following grammar

$C \rightarrow AB$
$A \rightarrow a$
$B \rightarrow a$

**Solution:** The augmented grammar of given grammar is,

1. $C' \rightarrow C$
2. $C \rightarrow AB$
3. $A \rightarrow a$
4. $B \rightarrow a$

Next, we obtain the canonical collection of sets of LR(0) items, as follows,

$I_0$ = closure ({$C' \rightarrow \bullet C$}) = {$C' \rightarrow \bullet C$, $C \rightarrow \bullet AB$, $A \rightarrow \bullet a$}

$I_1$ = goto($I_0$, C) = closure($C' \rightarrow C\bullet$) = { $C' \rightarrow C\bullet$}

$I_2$ = goto($I_0$, A) = closure($C \rightarrow A\bullet B$) = { $C \rightarrow A\bullet B$, $B \rightarrow \bullet a$}

$I_3$ = goto($I_0$, a) = closure($A \rightarrow a\bullet$) = { $A \rightarrow a\bullet$ }

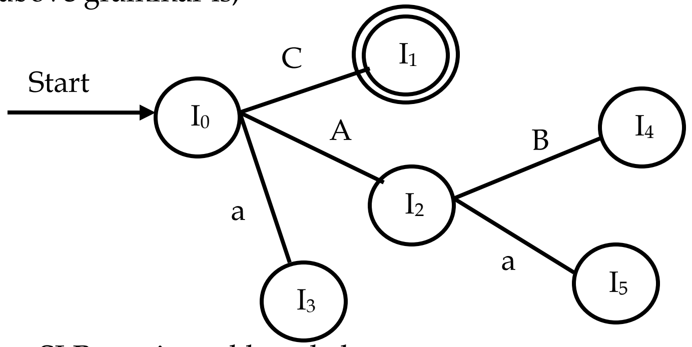
$I_4$ = goto($I_2$, B) = closure($C \rightarrow AB\bullet$) = { $C \rightarrow AB\bullet$}

$I_5$ = goto($I_2$, a) = closure($B \rightarrow a\bullet$) = { $B \rightarrow a\bullet$}

Now calculate FOLLOW function as,

FOLLOW(C′) = {\$}
FOLLOW(C) = {FOLLOW(C′)} = {\$}
FOLLOW(A) = {FIRST(B) U a} = {a}
FOLLOW(B) = {FOLLOW(C)} = {\$}

The DFA of above grammar is,



Now construct SLR parsing table as below,

| States | Action Table | | GOTO Table | | |
|---|---|---|---|---|---|
| | a | \$ | C | A | B |
| $I_0$ | Shift 3 | | State 1 | State 2 | |
| $I_1$ | | Accept | | | |
| $I_2$ | Shift 5 | | | | State 4 |
| $I_3$ | Reduce 3 | | | | |
| $I_4$ | | Reduce 2 | | | |
| $I_5$ | | Reduce 4 | | | |

**Example 2**: Construct the SLR parsing table for the grammar:

$S \rightarrow AA$

$A \rightarrow aA \mid b$

**Solution:** The augment the given grammar is,

0. $S' \rightarrow S$
1. $S \rightarrow AA$
2. $A \rightarrow aA \mid b$
3. $A \rightarrow b$

Next, we obtain the canonical collection of sets of LR(0) items, as follows,

$I_0$ = closure ({$S' \rightarrow \bullet S$}) = {$S' \rightarrow \bullet S$, $S \rightarrow \bullet AA$, $A \rightarrow \bullet aA$, $A \rightarrow \bullet b$}

$I_1$ = goto ({$I_0$, S}) = closure ({$S' \rightarrow S\bullet$}) = {$S' \rightarrow S\bullet$}

$I_2$ = goto ({$I_0$, A}) = closure ({$S \rightarrow A\bullet A$}) = {$S \rightarrow A\bullet A$, $A \rightarrow \bullet aA$, $A \rightarrow \bullet b$}

$I_3$ = goto ({$I_0$, a}) = closure ({$A \rightarrow a\bullet A$}) = {$A \rightarrow a\bullet A$, $A \rightarrow \bullet aA$, $A \rightarrow \bullet b$}

$I_4$ = goto ({$I_0$, b}) = closure ({$A \rightarrow b\bullet$}) = {$A \rightarrow b\bullet$}

$I_5$ = goto ({$I_2$, A}) = closure ({$S \rightarrow AA\bullet$}) = {$S \rightarrow AA\bullet$}

$I_6$ = goto ({$I_2$, a}) = closure ({$A \rightarrow a\bullet A$}) = {$A \rightarrow a\bullet A$, $A \rightarrow \bullet aA$, $A \rightarrow \bullet b$} Same as $I_3$

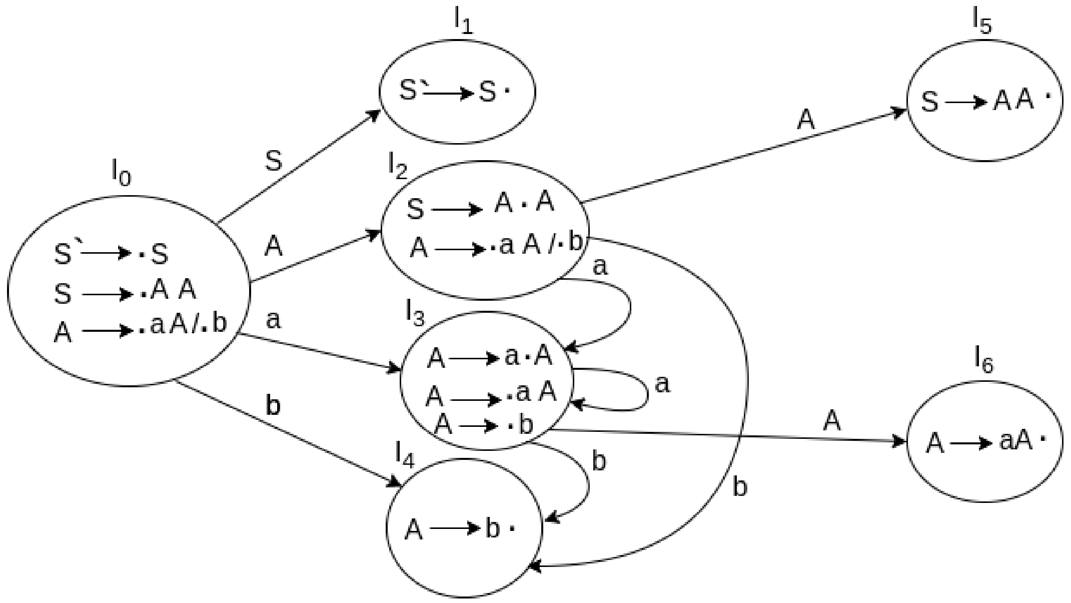
$I_6$ = goto ({$I_2$, b}) = closure ({$A \rightarrow b\bullet$}) = {$A \rightarrow b\bullet$} same as $I_4$

$I_6$ = goto ({$I_3$, A}) = closure ({$A \rightarrow aA\bullet$}) = {$A \rightarrow aA\bullet$}

$I_7$ = goto ({$I_3$, a}) = closure ({$A \rightarrow a\bullet A$}) = {$A \rightarrow a\bullet A$, $A \rightarrow \bullet aA$, $A \rightarrow \bullet b$} Same as $I_3$

$I_7$ = goto ({$I_3$, b}) = closure ({$A \rightarrow b\bullet$}) = {$A \rightarrow b\bullet$} same as $I_4$

**Drawing DFA:** The DFA contains the 7 states $I_0$ to $I_6$.



Now calculate FOLLOW function as,

FOLLOW(S') = {$} FOLLOW(S) = {FOLLOW(S')} = {$}

FOLLOW(A) = {FIRST(S)} = {$}

**SLR table or LR(0) Table of above grammar as below,**

| States | Action Table | | | Goto Table | |
|---|---|---|---|---|---|
| | a | b | $ | A | S |
| $I_0$ | $S_3$ | $S_4$ | | 2 | 1 |
| $I_1$ | | | Accept | | |
| $I_2$ | $S_3$ | $S_4$ | | 5 | |
| $I_3$ | $S_3$ | $S_4$ | | 6 | |
| $I_4$ | $r_3$ | $r_3$ | $r_3$ | | |
| $I_5$ | $r_1$ | $r_1$ | $r_1$ | | |
| $I_6$ | $r_2$ | $r_2$ | $r_2$ | | |

**Explanation**

- $I_0$ on S is going to $I_1$ so write it as 1.
- $I_0$ on A is going to $I_2$ so write it as 2.
- $I_2$ on A is going to $I_5$ so write it as 5.
- $I_3$ on A is going to $I_6$ so write it as 6.
- $I_0$, $I_2$ and $I_3$ on 'a' are going to $I_3$ so write it as $S_3$ which means that shift 3.
- $I_0$, $I_2$ and $I_3$ on 'b' are going to $I_4$ so write it as $S_4$ which means that shift 4.
- $I_4$, $I_5$ and $I_6$ all states contains the final item because they contain • in the right most end. So rate the production as production number.
- $I_1$ contains the final item which drives(S′ → S•), so action {I1, $} = Accept.
- $I_4$ contains the final item which drives A → b• and that production corresponds to the production number 3 so write it as $r_3$ in the entire row.
- $I_5$ contains the final item which drives S → AA• and that production corresponds to the production number 1 so write it as $r_1$ in the entire row.
- $I_6$ contains the final item which drives A → aA• and that production corresponds to the production number 2 so write it as $r_2$ in the entire row.

**Example 3**: Construct the SLR parsing table for the grammar,

E → E + T

E → T

T → T * F

T → F

F → (E)

F → id

And parse the input string id * id + id

**Solution:** The augment the given grammar is,

0. E′ → E

1. E → E + T

2. $E \rightarrow T$

3. $T \rightarrow T * F$

4. $T \rightarrow F$

5. $F \rightarrow (E)$

6. $F \rightarrow id$

Next, we obtain the canonical collection of sets of LR(0) items, as follows,

$I_0$ = closure ({$E' \rightarrow \bullet E$})

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

$I_1$ = goto($I_0$, E) = closure({$E' \rightarrow E\bullet$, $E \rightarrow E \bullet + T$}) = {$E' \rightarrow E\bullet$, $E \rightarrow E \bullet + T$}

$I_2$ = goto($I_0$, T) = closure({$E \rightarrow T\bullet$, $T \rightarrow T \bullet * F$}) = { $E \rightarrow T\bullet$, $T \rightarrow T \bullet * F$ }

$I_3$ = goto($I_0$, F) = closure({$T \rightarrow F\bullet$}) = {$T \rightarrow F\bullet$}

$I_4$ = goto($I_0$, ( ) = closure({$F \rightarrow (\bullet E)$})

= {$F \rightarrow (\bullet E)$, $E \rightarrow \bullet E + T$, $E \rightarrow \bullet T$, $T \rightarrow \bullet T * F$, $T \rightarrow \bullet F$, $F \rightarrow \bullet (E)$, $F \rightarrow \bullet id$}

$I_5$ = goto($I_0$, id ) = closure({$F \rightarrow id\bullet$}) = {$F \rightarrow id\bullet$}

$I_6$ = goto($I_1$, + ) = closure({$E \rightarrow E +\bullet T$}) = {$E \rightarrow E +\bullet T$, $T \rightarrow \bullet T * F$, $T \rightarrow \bullet F$, $F \rightarrow \bullet (E)$, $F \rightarrow \bullet id$}

$I_7$ = goto($I_2$, * ) = closure({$T \rightarrow T * \bullet F$}) = {$T \rightarrow T * \bullet F$, $F \rightarrow \bullet (E)$, $F \rightarrow \bullet id$}

No possible goto for $I_3$ since there is not any member after dot (•)

$I_8$ = goto($I_4$, E) = closure({$F \rightarrow (E\bullet)$, $E \rightarrow E\bullet + T$}) = {$F \rightarrow (E\bullet)$, $E \rightarrow E\bullet + T$ }

$I_9$ = goto($I_4$, T) = closure({$E \rightarrow T\bullet$, $T \rightarrow T\bullet * F$}) = {$E \rightarrow T\bullet$, $T \rightarrow T\bullet * F$ } which is same as $I_2$

Similarly, goto($I_4$, F) = $I_3$

goto($I_4$, id) = $I_5$

goto($I_4$, ( ) = $I_4$

No possible goto for $I_5$ since there is not any member after dot (•)

$I_9$ = goto($I_6$, T) = closure({$E \rightarrow E +T\bullet$, $T \rightarrow T\bullet * F$}) = {$E \rightarrow E +T\bullet$, $T \rightarrow T\bullet * F$}

Similarly, goto($I_6$, F) = $I_3$

goto($I_6$, id) = $I_5$

goto($I_6$, ( ) = $I_4$

$I_{10}$ = goto($I_7$, F ) = closure({$T \rightarrow T * F\bullet$})= {$T \rightarrow T * F\bullet$}

Similarly, goto($I_7$, ( ) = $I_4$

goto($I_7$, id) = $I_5$

$I_{11}$ = goto($I_8$, ) ) = closure({$F \rightarrow (E) \bullet$}) = {$F \rightarrow (E) \bullet$}

Similarly, goto($I_8$, + ) = $I_6$

goto($I_9$, *) = $I_7$

Now, finished stop and total numbers of state = 12

Now calculate FOLLOW function as,

FOLLOW(E′) = {\$}

FOLLOW(E) = {+, ) }

FOLLOW(T) = {FOLLOW(E), * } = {+, ), * }

FOLLOW(F) = { FOLLOW(T)} = {+, ), * }

**SLR table of above grammar as below**

| State | Action table | | | | | | Goto table | | |
|---|---|---|---|---|---|---|---|---|---|
| | id | + | * | ( | ) | $ | E | T | F |
| 0 | s5 | | | s4 | | | 1 | 2 | 3 |
| 1 | | s6 | | | | Accept | | | |
| 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 | | | |

**Parsing the input string id * id + id by using above parsing table**

| Stack | Input buffer | Action table | Goto table | Parsing action |
|---|---|---|---|---|
| $0 | id*id+id$ | [0, id]=$S_5$ | | Shift |
| $0 id5 | *id+id$ | [5, *]=$R_6$ | [0, F]=3 | Reduce by F → id |
| $0 F3 | *id+id$ | [3, *]=$R_4$ | [0, T]=2 | Reduce by T → F |
| $0 T2 | *id+id$ | [2, *]=$S_7$ | | Shift |
| $0 T2 * 7 | id+id$ | [7, id]=$S_5$ | | Shift |
| $0 T2 *id5 | +id$ | [5, +]=$R_6$ | [7, F]=10 | Reduce by F → id |
| $0 T2 *7F10 | +id$ | [10, +]=$R_3$ | [0, T]=2 | Reduce by T → T*F |
| $0 T2 | +id$ | [2, +]=$R_2$ | [0, E]=2 | Reduce by E → T |
| $0 E1 | +id$ | [1, +]=$S_6$ | | Shift |
| $0 E1+6 | +id$ | [6, id]=$S_5$ | | Shift |
| $0 E1+6id5 | $ | [5, $]=$R_6$ | [6, F]=3 | Reduce by F → id |
| $0 E1+6F3 | $ | [3, $]=$R_4$ | [6, T]=9 | Reduce by T → F |
| $0 E1+6T9 | $ | [9, $]=$R_1$ | [0, E]=1 | Reduce by E → E+T |
| $0 E1 | $ | Accept | | Accept |

**Example 4**: Construct the SLR parsing table for the grammar,

S → E

E → E + T | T

T → T * F | F

F → id

**Solution:** The augment the given grammar is,

0. S' → S

1. S → E

2. $E \rightarrow E + T \mid T$

3. $T \rightarrow T * F \mid F$

4. $F \rightarrow id$

$I_0$ = Closure ($S' \rightarrow \bullet E$)

= {$S' \rightarrow \bullet E$, $E \rightarrow \bullet E + T$, $E \rightarrow \bullet T$, $T \rightarrow \bullet T * F$, $T \rightarrow \bullet F$, $F \rightarrow \bullet id$}

$I_1$ = Goto ($I_0$, E) = closure ($S' \rightarrow E\bullet$, $E \rightarrow E\bullet + T$) = {$S' \rightarrow E\bullet$, $E \rightarrow E\bullet + T$}

$I_2$ = Goto ($I_0$, T) = closure ($E \rightarrow T\bullet$, $T \rightarrow T\bullet * F$) = {$E \rightarrow T\bullet$, $T \rightarrow T\bullet * F$}

$I_3$ = Goto ($I_0$, F) = Closure ($T \rightarrow F\bullet$) = {$T \rightarrow F\bullet$}

$I_4$ = Goto ($I_0$, id) = closure ($F \rightarrow id\bullet$) = {$F \rightarrow id\bullet$}

$I_5$ = Goto ($I_1$, +) = Closure ($E \rightarrow E + \bullet T$) = {$E \rightarrow E + \bullet T$, $T \rightarrow \bullet T * F$, $T \rightarrow \bullet F$, $F \rightarrow \bullet id$}

Goto ($I_5$, F) = Closure ($T \rightarrow F\bullet$) = {$T \rightarrow F\bullet$} same as $I_3$

Goto ($I_5$, id) = Closure ($F \rightarrow id\bullet$) = {$F \rightarrow id\bullet$} same as $I_4$

I6 = Goto ($I_2$, *) = Closure ($T \rightarrow T * \bullet F$) = {$T \rightarrow T * \bullet F$, $F \rightarrow \bullet id$}

Goto ($I_6$, id) = Closure ($F \rightarrow id\bullet$) = {$F \rightarrow id\bullet$} same as $I_4$

$I_7$ = Goto ($I_5$, T) = Closure ($E \rightarrow E + T\bullet$) = {$E \rightarrow E + T\bullet$}

$I_8$ = Goto ($I_6$, F) = Closure ($T \rightarrow T * F\bullet$) = {$T \rightarrow T * F\bullet$}

**Compute FOLLOW function as,**

Follow (S′) = {\$}

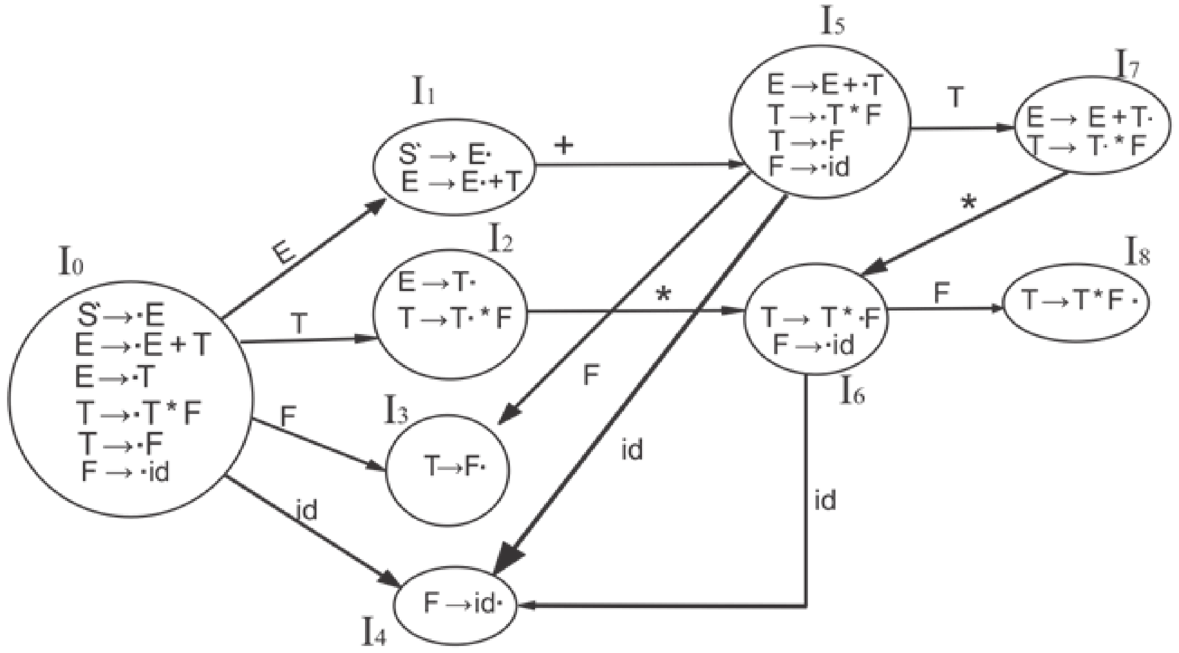
Follow (S) = {FOLLOW(S′)} = {\$}

Follow (E) = {First (+T) ∪ FOLLOW(S)} = {+, \$}

Follow (T) = {First (*F) ∪ First (F)} = {*, +, \$}

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

**Drawing DFA:** The DFA contains the 9 states $I_0$ to $I_8$.



SLR parsing table is,

| States | Action table | | | | Goto table | | |
|---|---|---|---|---|---|---|---|
| | id | + | * | $ | E | T | F |
| $I_0$ | $S_4$ | | | | 1 | 2 | 3 |
| $I_1$ | | $S_5$ | | Accept | | | |
| $I_2$ | | $R_2$ | $S_6$ | $R_2$ | | | |
| $I_3$ | | $R_4$ | $R_4$ | $R_4$ | | | |
| $I_4$ | | $R_5$ | $R_5$ | $R_5$ | | | |
| $I_5$ | $S_4$ | | | | | 7 | 3 |
| $I_6$ | $S_4$ | | | | | | 8 |
| $I_7$ | | $R_1$ | $S_6$ | $R_1$ | | | |
| $I_8$ | | $R_3$ | $R_3$ | $R_3$ | | | |

**Ambiguity in SLR Limitation of SLR**

Every SLR grammar is unambiguous. But there exist certain unambiguous grammars that are not SLR. In such grammar there exist at least one multiply defined entry action[i, a], which contains both a shift and reduce directive. The SLR technique still leaves something to be desired, because we are not using all the information that we have at our disposal. When we have a completed configuration (i.e., dot at the end) such as $A \rightarrow \alpha\bullet$, we know that this corresponds to a situation in which we have $\alpha$ as a handle on top of the stack which we then can reduce, i.e., replacing $\alpha$ by A.

**Example:** Consider a grammar

$S \rightarrow L = R$

$S \rightarrow R$

$L \rightarrow * R$

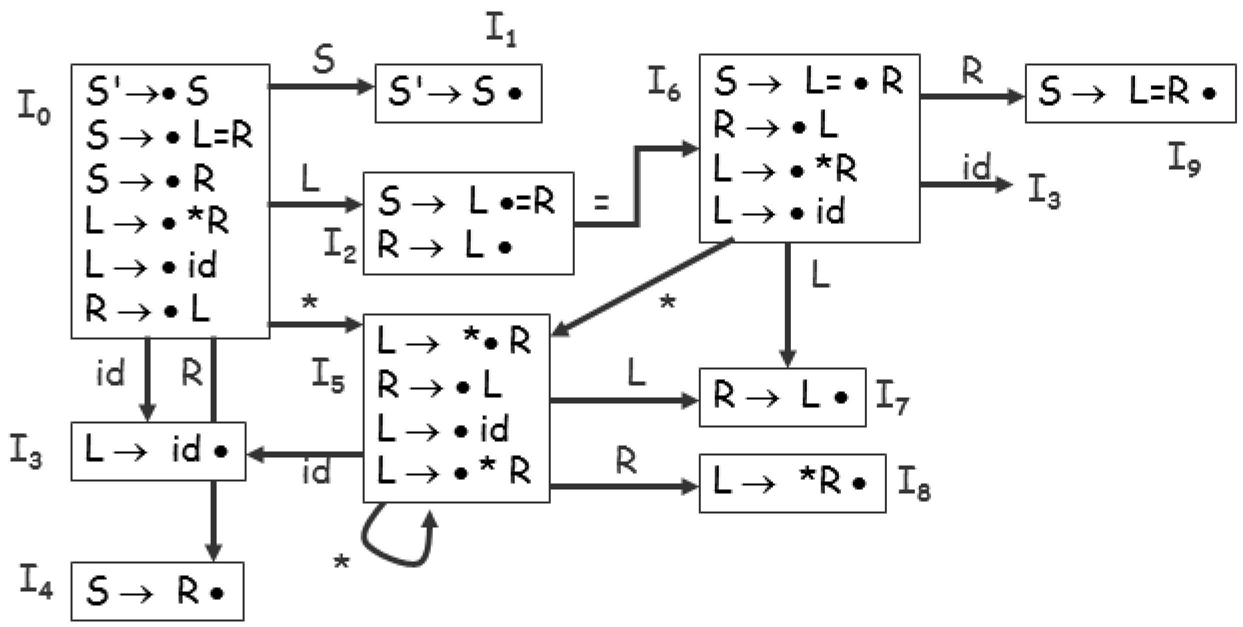
$L \rightarrow id$

$R \rightarrow L$

**Solution:** The augment the given grammar is,

0. $S' \rightarrow S$
1. $S \rightarrow L = R$
2. $S \rightarrow R$
3. $L \rightarrow * R$
4. $L \rightarrow id$
5. $R \rightarrow L$

$I_0$ = Closure ($S' \rightarrow \bullet S$) = {$S' \rightarrow \bullet S$, $S \rightarrow \bullet L = R$, $S \rightarrow \bullet R$, $L \rightarrow \bullet * R$, $L \rightarrow \bullet id$, $R \rightarrow \bullet L$}



Now compute FOLLOW as,

FOLLOW(S') = {$} FOLLOW(S) = {$}

FOLLOW(L) = {$, =} FOLLOW(R) = {$, =}

SLR parsing table is,

| **States** | **Action Table** | | | | **Goto Table** | | |
|---|---|---|---|---|---|---|---|
| | **id** | **=** | ***** | **$** | **S** | **L** | **R** |
| 0 | $S_5$ | | $S_4$ | | 1 | 2 | 3 |
| 1 | | | | Accept | | | |
| 2 | | **$S_5$/ $R_5$** | | $R_5$ | | | |
| 3 | | | | $R_2$ | | | |
| 4 | $S_5$ | | $S_4$ | | | 8 | 9 |
| 5 | | $R_4$ | | $R_4$ | | | |
| 6 | | | $S_4$ | $S_5$ | | 8 | 9 |
| 7 | | $R_3$ | | $R_3$ | | | |
| 8 | | $R_5$ | | $R_5$ | | | |
| 9 | | | | $R_1$ | | | |

In state 2, action [2, =] = $S_6$ and action [2, =] = $R_5$, it is seen that there is shift reduce conflict.

## LR(1) Grammars

SLR is so simple and can only represent the small group of grammar. LR(1) parsing uses look-ahead to avoid unnecessary conflicts in parsing table.

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

| LR(0) item | LR(1) item |
|---|---|
| $[A \to \alpha \bullet \beta]$ | $[A \to \alpha \bullet \beta, a]$ |

**Computation of Closure for LR(1)Items**

The closure of a set S of LR(1) items for augmented grammar G is computed as follows,

1. Start with closure(I) = I (where I is a set of LR(1) items)
2. If $[A \to \alpha \bullet B\beta, a] \in$ closure(I) then

    Add the item $[B \to \bullet \gamma, b]$ to I if not already in I, where $b \in$ FIRST($\beta a$)
3. Repeat 2 until no new items can be added.

**Computation of Goto Operation for LR(1) Items**

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

1. For each item $[A \to \alpha \bullet X\beta, a] \in I$, add the set of items

    closure($\{[A \to \alpha X \bullet \beta, a]\}$) to goto(I,X) if not already there
2. Repeat step 1 until no more items can be added to goto(I, X)

**Construction of The Canonical LR(1) Collection**

**Algorithm**

```
1. Augment the grammar with production S′→S
2. C = {closure({S′→•S, $})} (the start stat of DFA)
3. Repeat the followings until no more set of LR(1) items can be added to C.
        For each I ∈ C and each grammar symbol X ∈ (N∪T)
            Goto(I, X) ≠ Φ and goto(I, X) not ∈ C then
                add goto(I, X) to C
```

**Construction of LR(1) Parsing Table**

SLR used the LR(0) items, that is the items used were productions with an embedded dot, but contained no other (lookahead) information. The LR(1) items contain the same productions with embedded dots, but add a second component, which is a terminal (or \$). This second component becomes important only when the dot is at the extreme right. For LR(1) we do that reduction only if the input symbol is exactly the second component of the item.

**Algorithm**

1. Given the grammar G, construct an augmented grammar G′ by introducing a production of the form $S' \to S$, where S is the star symbol of G.
2. Construct the set $C = \{I_0, I_1, \ldots\ldots, I_n\}$ of LR(1) items.

3. Create the parsing action table as follows
    **a.** If $[A \rightarrow \alpha \bullet a\beta, b]$ in $I_i$ and goto $(Ii, a) = I_j$ then action [i, a] = ***shift j.***
    b. If $[A \rightarrow \alpha \bullet, a]$ is in $I_i$, then action [i, a] = ***reduce A*** $\rightarrow \alpha$ where $A \neq S'$.
    c. If $[S' \rightarrow S \bullet, \$]$ is in $I_i$ , then action[i, \$] = ***accept***.
    d. If any conflicting actions generated by these rules, the grammar is not LR(1).
4. Create the parsing goto table
    a. For all non-terminals A, if goto (Ii, A) = Ij then goto [i, A] = j
5. All entries not defined by (2) and (3) are errors.
6. Initial state of the parser contains $S' \rightarrow .S, \$$

**Example 1:** Construct LR(1) parsing table of following grammar,

$S \rightarrow AA$
$A \rightarrow aA$
$A \rightarrow b$

**Solution:** The augment the given grammar is,

$S' \rightarrow S$
$S \rightarrow AA$
$A \rightarrow aA$
$A \rightarrow b$

$I_0$ = Closure $(S' \rightarrow \bullet S)$ = $\{S' \rightarrow \bullet S, \$$
$S \rightarrow \bullet AA, \$$
$A \rightarrow \bullet aA, a/b$
$A \rightarrow \bullet b, a/b\}$

$I_1$ = Goto $(I_0, S)$ = closure $(S' \rightarrow S\bullet, \$)$ = $\{S' \rightarrow S\bullet, \$\}$

$I_2$ = Goto $(I_0, A)$ = closure $(S \rightarrow A\bullet A, \$)$ = $\{S \rightarrow A\bullet A, \$$
$A \rightarrow \bullet aA, \$$
$A \rightarrow \bullet b, \$\}$

$I_3$ = Goto $(I_0, a)$ = Closure $(A \rightarrow a\bullet A, a/b)$ = $\{A \rightarrow a\bullet A, a/b$
$A \rightarrow \bullet aA, a/b$
$A \rightarrow \bullet b, a/b\}$

$I_4$= Goto $(I_0, b)$ = closure $(A \rightarrow b\bullet, a/b)$ = $\{A \rightarrow b\bullet, a/b\}$

$I_5$= Goto $(I_2, A)$ = Closure $(S \rightarrow AA\bullet, \$)$ = $\{S \rightarrow AA\bullet, \$\}$

$I_6$= Goto $(I_2, a)$ = Closure $(A \rightarrow a\bullet A, \$)$
= $\{A \rightarrow a\bullet A, \$$
$A \rightarrow \bullet aA, \$$
$A \rightarrow \bullet b, \$\}$

$I_7$= Goto $(I_2, b)$ = Closure $(A \rightarrow b\bullet, \$)$ = $\{A \rightarrow b\bullet, \$\}$

$I_8$= Goto $(I_3, A)$ = Closure $(A \rightarrow aA\bullet, a/b)$ = $\{A \rightarrow aA\bullet, a/b\}$

Goto $(I_3, a)$ = Closure $(A \rightarrow a\bullet A, a/b)$ = $\{A \rightarrow a\bullet A, a/b$
$A \rightarrow \bullet aA, a/b$

A → •b, a/b} same as $I_3$

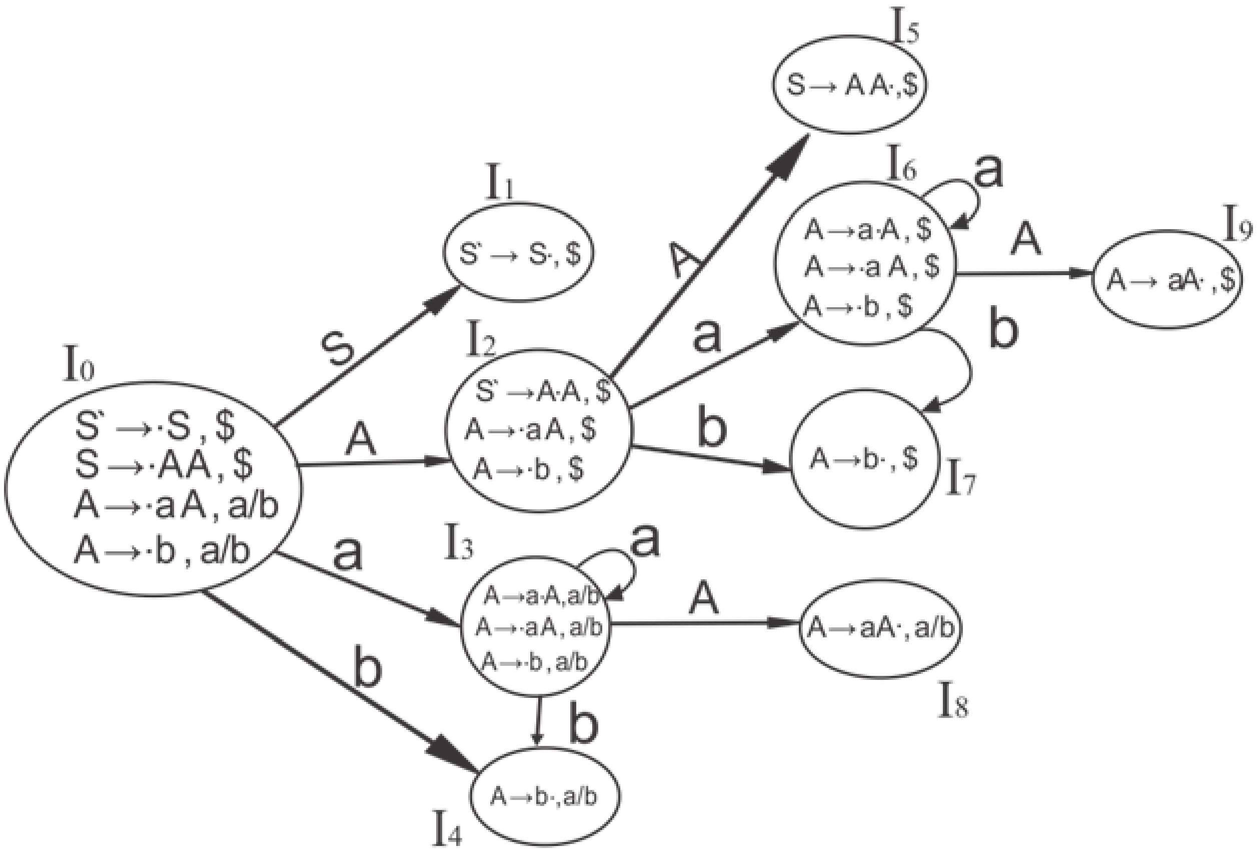
Goto ($I_3$, b) = Closure (A → b•, a/b) same as $I_4$

$I_9$= Goto ($I_6$, A) = Closure (A → aA•, \$) = {A → aA•, \$}

Goto ($I_6$, a) = Closure (A → a•A, \$) same as $I_6$

Goto ($I_6$, b) = Closure (A → b•, \$) same as $I_7$

The DFA is,



The LR(1) parsing table is,

| States | Action table | | | Goto table | |
|---|---|---|---|---|---|
| | a | b | \$ | S | A |
| 0 | $S_3$ | $S_4$ | | 1 | 2 |
| 1 | | | **Accept** | | |
| 2 | $S_6$ | $S_7$ | | | 5 |
| 3 | $S_3$ | $S_4$ | | | 8 |
| 4 | $R_3$ | $R_3$ | | | |
| 5 | | | $R_1$ | | |
| 6 | $S_6$ | $S_7$ | | | 9 |
| 7 | | | $R_3$ | | |
| 8 | $R_2$ | $R_2$ | | | |
| 9 | | | $R_2$ | | |

**Example 2:** Construct LR(1) parsing table for the augmented grammar,

0. S′ → *S*
1. *S* → *L* **=** *R*
2. *S* → *R*
3. *L* → ***** *R*
4. *L* → **id**
5. *R* → *L*

Step 1: At first find the canonical collection of LR(1) items of the given augmented grammar as,

**State $I_0$:**

Closure(S′→•S, $)
{S′→•S, $
*S* → •*L* = *R*, *$*
*S* → •*R*, *$*
*L* → •* *R*, *$*
*L* → •Id, =
*R* → •*L*, *$}*

**State $I_1$:**

closure (goto($I_0$,S))
closure(S′→ S•, $)
{S′→ S•, $
*R* → L•, *$}*

**State $I_2$:**

closure (goto($I_0$, L))
closure((*S* → *L*• = *R*, *$*), ( *R* →L•, *$*))
{*S* → *L*• = *R*, *$*}

**State $I_3$:**

closure (goto($I_0$, R))
closure(*S* → R•, *$*)
*{S* → R•, *$*}

**State $I_4$:**

closure(goto($I_0$, *))
closure(*L* → * •*R*, =)
{(*L* → * •*R*, =), (*R* → •*L*, =), (*L* → •* *R*, =), (*L* → •Id, =)}

**State $I_5$:**

closure (goto($I_0$, id))
closure(*L* → Id• , =)
{*L* → Id•, =}

**State $I_6$:**

closure(goto(($I_2$, =))
closure(*S* → *L* =• *R*, *$*)
{*S* → *L* =• *R*, *$*
*R* →•*L*, *$*
*L* → •* *R*, *$*
*L* →•Id, $}

**State $I_7$:**

closure(goto(($I_4$, R))
closure(*L* → * *R*• ,=)
{*L* → * *R*•, =}

**State $I_8$:**

closure (goto($I_4$, L))
closure(R → L•, =)
{R → L•, =}

**State $I_9$:**

closure(goto(($I_6$, R))
closure(S → L=R• , $)
{S → L=R•, $}

**State $I_{10}$:**

closure(goto(($I_6$, L))
{R → L•, $}

**State $I_{11}$:**

Closure(goto($I_6$, *))
Closure(L→*•R, $)
{L → *•R, $
R → •L, $
L → •*R, $
L → •id, $}

**State $I_{12}$:**

closure(goto(($I_6$, id))
closure(L → id•, $)
{L → id•, $}

**State $I_{13}$:**

closure(goto(($I_{11}$, R))
{L → *R•, $}

**Step 2:** Now construct LR(1) parsing table

| States | Action Table | | | | Goto Table | | |
|---|---|---|---|---|---|---|---|
| | id | * | = | $ | S | L | R |
| 0 | $S_5$ | $S_4$ | | | 1 | 2 | 3 |
| 1 | | | | Accept | | | |
| 2 | | | $S_6$ | $R_5$ | | | |
| 3 | | | | $R_2$ | | | |
| 4 | $S_5$ | $S_4$ | | | | 8 | 7 |
| 5 | | | $R_4$ | $R_4$ | | | |
| 6 | $S_{12}$ | $S_{11}$ | | | | 10 | 9 |
| 7 | | | $R_3$ | $R_3$ | | | |
| 8 | | | $R_5$ | $R_5$ | | | |
| 9 | | | | $R_1$ | | | |
| 10 | | | | $R_5$ | | | |
| 11 | $S_{12}$ | $S_{11}$ | | | | 10 | 13 |
| 12 | | | | $R_4$ | | | |
| 13 | | | | $R_3$ | | | |

**Example 3:** Construct canonical LR(1) collection of the following grammar

$S \rightarrow AaAb$
$S \rightarrow BbBa$
$A \rightarrow \in$
$B \rightarrow \in$

**Solution:** Its augmented grammar is,

0. $S' \rightarrow S$
1. $S \rightarrow AaAb$
2. $S \rightarrow BbBa$
3. $A \rightarrow \in$
4. $B \rightarrow \in$

$I_0$ = closure($S' \rightarrow \bullet S$, \$)
= {$S' \rightarrow \bullet S$, \$
$S \rightarrow \bullet\ AaAb$, \$
$S \rightarrow \bullet BbBa$, \$
$A \rightarrow \bullet$, a
$B \rightarrow \bullet$, b}

$I_1$: goto($I_0$, S) = closure($S' \rightarrow S\bullet$, \$) = {$S' \rightarrow S\bullet$, \$}
$I_2$: goto($I_0$, A) = closure($S \rightarrow A\bullet aAb$, \$) = {$S \rightarrow A\bullet aAb$, \$}
$I_3$: goto($I_0$, B) = closure($S \rightarrow B\bullet bBa$, \$) = {$S \rightarrow B\bullet bBa$, \$}
$I_4$: goto($I_2$, a) = closure($S \rightarrow Aa\bullet Ab$, \$) = {($S \rightarrow Aa\bullet Ab$, \$), ($A \rightarrow \bullet$, b)}
$I_5$: goto($I_3$, b) = closure($S \rightarrow Bb\bullet Ba$, \$) = {($S \rightarrow Bb\bullet Ba$, \$), ($B \rightarrow \bullet$, a)}
$I_6$: goto($I_4$, A) = closure($S \rightarrow AaA\bullet b$, \$) = {$S \rightarrow AaA\bullet b$, \$}
$I_7$: goto($I_5$, B) = closure($S \rightarrow BbB\bullet a$, \$) = {$S \rightarrow BbB\bullet a$, \$}
$I_8$: goto($I_6$, b) = closure($S \rightarrow AaAb\bullet$, \$) = {$S \rightarrow AaAb\bullet$, \$}
$I_9$: goto($I_7$, a) = closure($S \rightarrow BbBa\bullet$, \$) = {$S \rightarrow BbBa\bullet$, \$}

Compute FOLLOW as,

FOLLOW(S′) = {\$}
FOLLOW(S) = {FOLLOW(S′)} = {\$}
FOLLOW(A) = {a, b}
FOLLOW(B) = {a, b}

Now LR(1) parsing table is,

| States | Action Table | | | Goto Table | | |
|---|---|---|---|---|---|---|
| | a | b | \$ | S | A | B |
| 0 | $R_{3,4}$ | $R_{3,4}$ | | 1 | 2 | 3 |
| 1 | | | Accept | | | |
| 2 | $S_4$ | | | | | |
| 3 | | $S_5$ | | | | |
| 4 | $R_3$ | $R_3$ | | | 6 | |
| 5 | $R_4$ | $R_4$ | | | | 7 |
| 6 | | $S_8$ | | | | |
| 7 | $S_9$ | | | | | |
| 8 | | | $R_1$ | | | |
| 9 | | | $R_2$ | | | |

## LALR(1) Grammars

It is an intermediate grammar between the SLR and LR(1) grammar. A typical programming language generates thousands of states for canonical LR parsers while they generate only hundreds of states for LALR parser. 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.

**Example:** Following example show the comparison between LR(1) and LALR parsing

| In LR(1) | In LALR(1) |
|---|---|
| **$I_1$:** $L \rightarrow id.\ , =$ | **$I_{12}$:** $L \rightarrow id.\ , =$ $L \rightarrow id.\ , \$$ |
| **$I_2$:** $L \rightarrow id.\ , \$$ | |

### Constructing LALR Parsing Tables

1. Create the canonical LR(1) collection of the sets of LR(1) items for the given grammar C = {$I_0$, $I_0$…,$I_n$}
1. For each core present in C, find all sets having the core and replace the sets by their union. Let C′ = {$J_0$, $J_1$,…….,$J_m$} be the resulting set of LR(1) items.
2. Construct the action elements of parsing table using the same method as for canonical LR(1) grammars.
3. If J is the union of k LR(1) items, J = $I_1$ U $I_2$ U…… U $I_k$, then the cores of goto[$I_1$, X] goto[$I_2$, X], etc. are all same since $I_1$, $I_2$,…..,$I_k$ have the same core. Let K be the union of all sets of items having the same core as goto[$I_i$, X]. Then goto[J, X] = K.

**Example 1:** Construct LALR parsing table for following grammar,

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

**Solution**: The augmented grammar of above grammar is,

0. $S' \rightarrow S$
1. $S \rightarrow L = R$
2. $S \rightarrow R$
3. $L \rightarrow * R$
4. $L \rightarrow \textbf{id}$
5. $R \rightarrow L$

**Step 1:** At first find the canonical collection of LR(1) items of the given augmented grammar as,

**State $I_0$:**

Closure(S'→•S, \$)
{S'→•S, \$
$S \rightarrow \bullet L = R$, \$
$S \rightarrow \bullet R$, \$
$L \rightarrow \bullet * R$, \$
$L \rightarrow \bullet$Id, =
$R \rightarrow \bullet L$, \$}

**State $I_1$:**

closure (goto($I_0$, S))
closure(S'→ S•, \$)
{S'→ S•, \$
$R \rightarrow$ L•, \$}

**State $I_2$:**

closure (goto($I_0$, L))
closure(($S \rightarrow L \bullet = R$, \$), ( $R \rightarrow$L•, \$))
{$S \rightarrow L \bullet = R$, \$}

**State $I_3$:**

closure (goto($I_0$, R))
closure($S \rightarrow$ R•, \$)
{$S \rightarrow$ R•, \$}

**State $I_4$:**

closure(goto($I_0$, *))
closure($L \rightarrow * \bullet R$, =)
{($L \rightarrow * \bullet R$, =), ($R \rightarrow \bullet L$, =), ($L \rightarrow \bullet * R$, =), ($L \rightarrow \bullet$Id, =)}

**State $I_5$:**

closure (goto($I_0$, id))
closure($L \rightarrow$ Id• , =)
{$L \rightarrow$ Id•, =}

**State $I_6$:**

closure(goto(($I_2$, =))
closure($S \rightarrow L = \bullet R$, \$)
{$S \rightarrow L = \bullet R$, \$
$R \rightarrow \bullet L$, \$
$L \rightarrow \bullet * R$, \$
$L \rightarrow \bullet$Id, \$}

**State $I_7$:**

closure(goto(($I_4$, R))
closure($L \rightarrow * R \bullet$ ,=)
{$L \rightarrow * R \bullet$, =}

**State $I_8$:**

closure (goto($I_4$, L))
closure(R → L•, =)
{R → L•, =}

**State $I_9$:**

closure(goto(($I_6$, R))
closure(S → L=R• , \$)
{S → L=R•, \$}

**State $I_{10}$:**

closure(goto(($I_6$, L))
{R → L•, \$}

**State $I_{11}$:**
Closure(goto($I_{6,*}$))
Closure(L→*•R, $)
{L → *•R, $
R → •L, $
L → •*R, $
L → •id, $}

**State $I_{12}$:**
closure(goto(($I_{6,}$ id))
closure(L → id•, $)
{L → id•, $}

**State $I_{13}$:**
closure(goto(($I_{11,}$ R))
{L → *R•, $}

**State 2:** Combine LR(1) sets with sets of items that share the same first part Combine LR(1) sets with sets of items that share the same first part i.e. core part.

**Combine state 4 and 11 as,**

$I_4$: {(*L* → * •*R*, =), (*R* → •*L*, =), (*L* → •* *R*, =), (*L* → •Id, =)}
$I_{11}$: {(L → *•R, $), (R → •L, $ ), (L → •*R, $), (L → •id, $)}
$I_{4,11}$: {(*L* → * •*R*, =/$), (*R* → •*L*, =/$), (*L* → •* *R*, =/$), (*L* → •Id, =/$)}

**Combine state 5 and 12 as,**

$I_5$: {*L* → Id•, =}

$I_{12}$: {*L* → Id•, $}

$I_{5,12}$: {*L* → Id•, =/$}

**Combine state 7 and 13 as,**

*$I_7$: {L → * R•, =}*

*$I_{13}$: {L → * R•, $}*

*$I_{7,13}$: {L → * R•, =/$}*

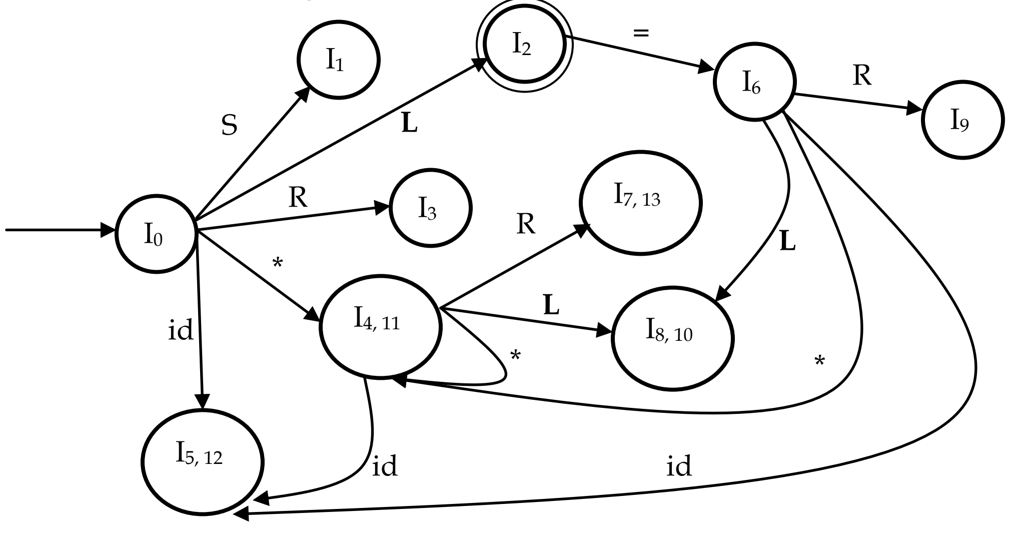
**Combine state 8 and 10 as,**

*$I_8$:* {R → L•, =}

*$I_{10}$:* {R → L•, $}

*$I_{8,10}$:* {R → L•, =/$}

**Step 3:** The DFA of LALR parsing is,



**Step 4:** The LALR parsing table is,

| States | Action Table | | | | Goto Table | | |
|---|---|---|---|---|---|---|---|
| | id | * | = | $ | S | L | R |
| 0 | $S_5$ | $S_4$ | | | 1 | 2 | 3 |
| 1 | | | | Accept | | | |
| 2 | | | $S_6$ | $R_5$ | | | |
| 3 | | | | $R_2$ | | | |
| 4 | $S_5$ | $S_4$ | | | | 8 | 9 |
| 5 | | | $R_4$ | $R_4$ | | | |
| 6 | $S_{12}$ | $S_{11}$ | | | | 10 | 9 |
| 7 | | | $R_3$ | $R_3$ | | | |
| 8 | | | $R_5$ | $R_5$ | | | |
| 9 | | | | $R_1$ | | | |

**Example 2:** Construct LALR parsing table for following grammar,

$S \rightarrow AA$
$A \rightarrow aA$
$A \rightarrow b$

**Solution**: The augmented grammar of above grammar is,

0. $S' \rightarrow S$
1. $S \rightarrow AA$
2. $A \rightarrow aA$
3. $A \rightarrow b$

**Step 1:** At first find the canonical collection of LR(1) items of the given augmented grammar as,

$I_0$ = Closure ($S' \rightarrow \bullet S$)
= {($S' \rightarrow \bullet S$, \$), ($S \rightarrow \bullet AA$, \$), ($A \rightarrow \bullet aA$, a/b), ($A \rightarrow \bullet b$, a/b)}

$I_1$ = Goto ($I_0$, S) = Closure ($S' \rightarrow S\bullet$)
= {$S' \rightarrow S\bullet$, \$}

$I_2$= Goto ($I_0$, A) = closure ($S \rightarrow A\bullet A$, \$)
= {($S \rightarrow A\bullet A$, \$), ($A \rightarrow \bullet aA$, \$), ($A \rightarrow \bullet b$, \$)}

$I_3$= Goto ($I_0$, a) = Closure ($A \rightarrow a\bullet A$, a/b)
= {($A \rightarrow a\bullet A$, a/b), ($A \rightarrow \bullet aA$, a/b), ($A \rightarrow \bullet b$, a/b)}

$I_4$= Goto ($I_0$, b) = closure ($A \rightarrow b\bullet$, a/b) = {$A \rightarrow b\bullet$, a/b}

$I_5$= Goto ($I_2$, A) = Closure ($S \rightarrow AA\bullet$, \$) = {$S \rightarrow AA\bullet$, \$}

$I_6$= Goto ($I_2$, a) = Closure ($A \rightarrow a\bullet A$, \$) = {($A \rightarrow a\bullet A$, \$), ($A \rightarrow \bullet aA$, \$), ($A \rightarrow \bullet b$, \$)}

$I_7$= Goto ($I_2$, b) = Closure ($A \rightarrow b\bullet$, \$) = {$A \rightarrow b\bullet$, \$}

Goto ($I_3$, a) = Closure ($A \rightarrow a\bullet A$, a/b) = same as $I_3$

Goto ($I_3$, b) = Closure ($A \rightarrow b\bullet$, a/b) = same as $I_4$

$I_8$= Goto ($I_3$, A) = Closure ($A \rightarrow aA\bullet$, a/b) = {$A \rightarrow aA\bullet$, a/b}

Goto ($I_6$, a) = Closure ($A \rightarrow a\bullet A$, \$) = same as $I_6$

Goto ($I_6$, b) = Closure ($A \rightarrow b\bullet$, \$) = same as $I_7$

$I_9$= Goto ($I_6$, A) = Closure ($A \rightarrow aA\bullet$, \$) = {$A \rightarrow aA\bullet$, \$}

**Step 2:** Combine LR(1) sets with sets of items that share the same first part Combine LR(1) sets with sets of items that share the same first part i.e. core part.

**Combine state 3 and 6 as,**

$I_{3,6}$ = {(A → a•A, a/b/\$), (A → •aA, a/b/\$), (A → •b, a/b/\$)}

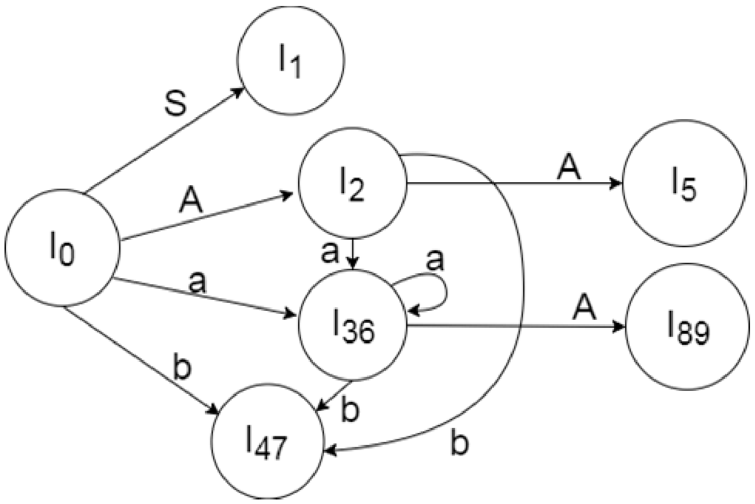
**Combine state 4 and 7 as,**

$I_{4,7}$ = {A → b•, a/b/\$}

**Combine state 8 and 9 as,**

$I_{8,9}$ = {A → aA•, a/b/\$}

**Step 3:** The DFA of LALR parsing is,



**Step 4:** The LALR parsing table is,

| States | Action Table | | | Goto Table | |
|---|---|---|---|---|---|
| | a | b | \$ | S | A |
| $I_0$ | $S_{3,6}$ | $S_{4,7}$ | | 1 | 2 |
| $I_1$ | | | Accept | | |
| $I_2$ | $S_{3,6}$ | $S_{4,7}$ | | | 5 |
| $I_{3,6}$ | $S_{3,6}$ | $S_{4,7}$ | | | 8,9 |
| $I_{4,7}$ | $R_3$ | $R_3$ | $R_3$ | | |
| $I_5$ | | | $R_1$ | | |
| $I_{8,9}$ | $R_2$ | $R_2$ | $R_2$ | | |

## Kernel and Non-Kernel Items

In order to devise a more efficient way of building LALR parsing tables, we define the terms kernel items and non-kernel items. Other than the initial item [S′ → •S, \$] no other item generated by a goto has a dot at the left end of the production. Such items (i.e. the initial item and all other items generated by goto) are called kernel items. Items that are generated by closure over kernel items have a dot at the beginning of the production. These items are called non-kernel items.

In brief, kernel item includes the initial items, S′→ .S and all items whose dot are not at the left end. Similarly non-kernel items are those items which have their dots at the left end except S′→ .S

**Example 1:** Find the kernel and non-kernel items of following grammar,

C → AB
A → a
B → a

**Solution:** The augmented grammar of given grammar is,

1. C′→ C
2. C → AB
3. A → a
4. B → a

Next, we obtain the canonical collection of sets of LR(0) items, as follows,

$I_0$ = closure ({C′ → •C}) = {C′ → •C, C → •AB, A → •a}

$I_1$ = goto($I_0$, C) = closure(C′ → C•) = {C′ → C•}

$I_2$ = goto($I_0$, A) = closure(C → A•B) = {C → A•B, B → •a}

$I_3$ = goto($I_0$, a) = closure(A → a•) = {A → a•}

$I_4$ = goto($I_2$, B) = closure(C → AB•) = {C → AB•}

$I_5$ = goto($I_2$, a) = closure(B → a•) = {B → a•}

List of kernel and non-kernel items are listed below;

| States | Kernel Items | Non-kernel items |
|---|---|---|
| $I_0$ | C′ → •C | C → •AB<br>A → •a |
| $I_1$ | C′ → C• | |
| $I_2$ | C → A•B | B → •a |
| $I_3$ | A → a• | |
| $I_4$ | C → AB• | |
| $I_5$ | B → a• | |

## Difference between Top down parsing and Bottom up parsing

op-down Parsing is a parsing technique that first looks at the highest level of the parse tree and works down the parse tree by using the rules of grammar while Bottom-up Parsing is a parsing technique that first looks at the lowest level of the parse tree and works up the parse tree by using the rules of grammar. There are some differences present to differentiate these two parsing techniques, which are given below:

| S. no | Top down parsing | Bottom up parsing |
|---|---|---|
| 1. | It is a parsing strategy that first looks at the highest level of the parse tree and works down the parse tree by using the rules of grammar. | It is a parsing strategy that first looks at the lowest level of the parse tree and works up the parse tree by using the rules of grammar. |

|  |  |  |
|---|---|---|
| 2. | Top-down parsing attempts to find the left most derivations for an input string. | Bottom-up parsing can be defined as an attempt to reduce the input string to start symbol of a grammar. |
| 3. | In this parsing technique we start parsing from top (start symbol of parse tree) to down (the leaf node of parse tree) in top-down manner. | In this parsing technique we start parsing from bottom (leaf node of parse tree) to up (the start symbol of parse tree) in bottom-up manner. |
| 4. | This parsing technique uses Left Most Derivation. | This parsing technique uses Right Most Derivation. |
| 5. | Its main decision is to select what production rule to use in order to construct the string. | Its main decision is to select when to use a production rule to reduce the string to get the starting symbol. |
| 6. | Error detection is easy | Error detection is difficult |
| 7. | Less power | High power |
| 8. | Parsing table size is small | Parsing table size is bigger than TDP |
| 9. | It uses left most derivation | It uses reverse of right most derivation |

## Parser Generators

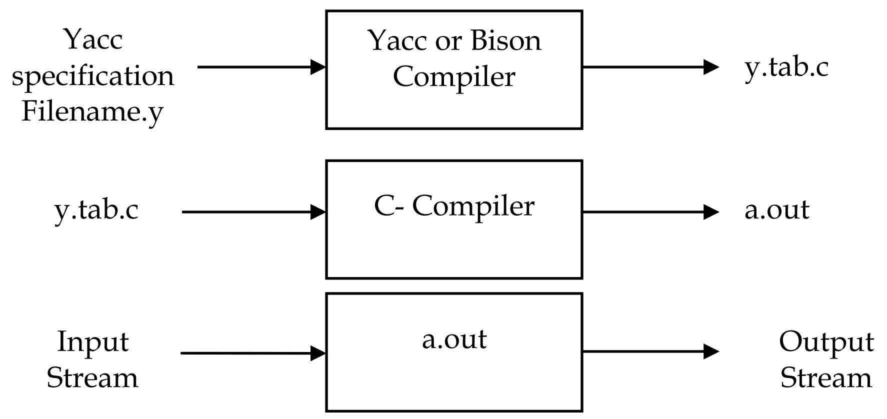
### Introduction to Bison

Bison is a general purpose parser generator that converts a description for an LALR(1) context-free grammar into a C program file. The job of the Bison parser is to group tokens into groupings according to the grammar rules—for example, to build identifiers and operators into expressions. The tokens come from a function called the lexical analyzer that must supply in some fashion (such as by writing it in C).

YACC is an automatic tool for generating the parser program. YACC stands for Yet Another Compiler which is basically the utility available from UNIX. Basically YACC is LALR parser generator. It can report conflict or ambiguities in the form of error messages.

The Bison parser calls the lexical analyzer each time it wants a new token. It doesn't know what is inside the tokens. Typically the lexical analyzer makes the tokens by parsing characters of text, but Bison does not depend on this. The Bison parser file is C code which defines a function named **yyparse** which implements that grammar. This function does not make a complete C program: you must supply some additional functions.

**Bison program specification,**



## Stages in Writing Bison program

- Formally specify the grammar in a form recognized by Bison
- Write a lexical analyzer to process input and pass tokens to the parser.
- Write a controlling function that calls the Bison produced parser.
- Write error-reporting routines.

## Bison Program structure

A bison specification consists of four parts:

```
%{
        C declarations
%}
Bison declarations
%%
        Grammar rules
%%
        Additional C codes
        Productions in Bison are of the form
        Non-terminal:   tokens/non-terminals {action}
        |Tokens/non | terminals {action}
        ……………………………….. ;
```

**Programming Example**

```
/* Mini Calculator */
/* calc.lex */
%{
        #include "heading.h"
        #include "tok.h"
        int yyerror(char *s);
        int yylineno = 1;
%}
        digit           [0-9]
        int_const       {digit}+
%%
        {int_const}     {yylval.int_val = atoi(yytext); return INTEGER_LITERAL; }
        "+"             {yylval.op_val = new std::string(yytext); return PLUS; }
        "*"             {yylval.op_val = new std::string(yytext); return MULT; }
        [\t]*           { }
        [\n]            {yylineno++;   }

        .               {std::cerr << "SCANNER "; yyerror(""); exit(1);        }
%%
/* Mini Calculator */
/* calc.y */
%{
        #include "heading.h"
        int yyerror(char *s);
        int yylex(void);
%}
%union{
                int     int_val;
                string* op_val;
        }
        %start          input
        %token          <int_val> INTEGER_LITERAL
        %type           <int_val> exp
        %left           PLUS
        %left           MULT
%%
input:          /* empty */
                | exp   { cout << "Result: " << $1 << endl;}
                ;
exp:            INTEGER_LITERAL  { $$ = $1; }
                | exp PLUS exp   { $$ = $1 + $3; }
                | exp MULT exp   { $$ = $1 * $3; }
                ;
%%
int yyerror(string s)
{
        extern int yylineno;    // defined and maintained in lex.c
        extern char *yytext;    // defined and maintained in lex.c
        cerr << "ERROR: " << s << " at symbol \"" << yytext;
        cerr << "\" on line " << yylineno << endl;
        exit(1);
}
int yyerror(char *s)
{
        return yyerror(string(s));
}
```

**Exercise:**

Q. For the grammar,

$S \rightarrow [C]S \mid \in$

$C \rightarrow \{A\}C \mid \in$

$A \rightarrow A(\,) \mid \in$

Construct the predictive top down parsing table (LL (1) parsing table)

**[1]. Construct the SLR parsing table for the following grammar**

$X \rightarrow S\,S + \mid S\,S * \mid a$

**[2]. Construct the SLR parsing table for the following grammar**

$S' \rightarrow S$

$S \rightarrow aABe$

$A \rightarrow Abc$

$A \rightarrow b$

$B \rightarrow d$

3. Construct LR(1) parsing table for given grammar:

$S' \rightarrow S$

$S \rightarrow CC$

$C \rightarrow cC$

$C \rightarrow d$

**Example 3**: Show that the following grammar is LR(1) but not LALR(1)

$S \rightarrow Aa \mid bAc \mid Bc \mid bBa$

$A \rightarrow d$

$B \rightarrow d$