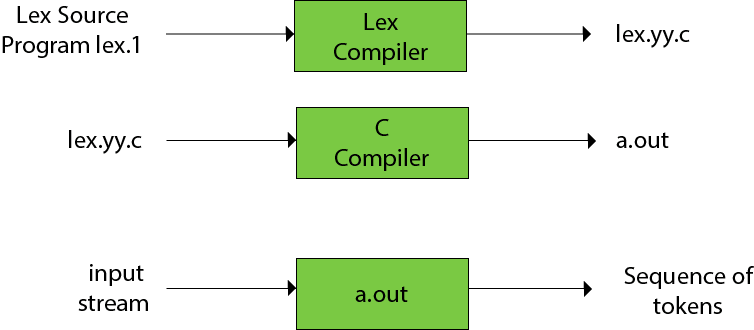
**LEX**

* **Lex is a program that generates a lexical analyser. It is used with the YACC parser generator.**
* **The lexical analyser is a program that transforms an input stream into a sequence of tokens.**
* **It reads the input stream and produces the source code as output through implementing the lexical analyser in the C program.**

**The function of Lex is as follows:**

* **Firstly lexical analyser creates a program lex.1 in the Lex language. Then Lex compiler runs the lex.1 program and produces a C program lex.yy.c.**
* **Finally C compiler runs the lex.yy.c program and produces an object program a.out.**
* **a.out is lexical analyser that transforms an input stream into a sequence of tokens.**

****

**Lex file format**

**A Lex program is separated into three sections by %% delimiters. The formal of Lex source is as follows:**

1. **{ definitions }**
2. **%%**
3. **{ rules }**
4. **%%**
5. **{ user subroutines }**

**Definitions include declarations of constant, variable, and regular definitions.**

**Rules define the statement of form p1 {action1} p2 {action2}....pn {action}.**

**Where pi describes the regular expression and action1 describes the actions and what action the lexical analyzer should take when pattern pi matches a lexeme.**

**User subroutines are auxiliary procedures needed by the actions. The subroutine can be loaded with the lexical analyzer and compiled separately.**

**Formal grammar**

* **Formal grammar is a set of rules. It is used to identify correct or incorrect strings of tokens in a language. The formal grammar is represented as G.**
* **Formal grammar is used to generate all possible strings over the alphabet that is syntactically correct in the language.**
* **Formal grammar is used mostly in the syntactic analysis phase (parsing), particularly during the compilation.**

**Formal grammar G is written as follows:**

1. **G = <V, N, P, S>**

**Where:**

**N describes a finite set of non-terminal symbols.  
V describes a finite set of terminal symbols.  
P describes a set of production rules  
S is the start symbol.**

**Example:**

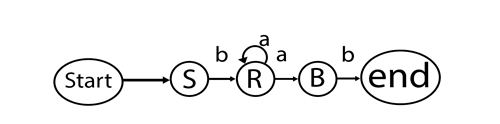
1. **L = {a, b}, N = {S, R, B}**

**Production rules:**

1. **S = bR**
2. **R = aR**
3. **R = aB**
4. **B = b**

**Through this production, we can produce some strings like bab, baab, baaab etc.**

**This production describes the string of shape banab.**

****

**Fig: Formal grammar**

**BNF Notation**

**BNF stands for Backus-Naur Form. It is used to write a formal representation of a context-free grammar. It is also used to describe the syntax of a programming language.**

**BNF notation is basically just a variant of a context-free grammar.**

**In BNF, productions have the form:**

1. **Left side → definition**

**Where leftside ∈ (Vn∪ Vt)+ and definition ∈ (Vn∪ Vt)\*. In BNF, the leftside contains one non-terminal.**

**We can define the several productions with the same leftside. All the productions are separated by a vertical bar symbol "|".**

**There is the production for any grammar as follows:**

1. **S → aSa**
2. **S → bSb**
3. **S → c**

**In BNF, we can represent above grammar as follows:**

1. **S → aSa| bSb| c**

**Context-free grammar**

A context-free grammar is a formal grammar which is used to generate all possible strings in a given formal language.

Four tuples can define context-free grammar G as:

1. G= (V, T, P, S)

Where,

**G** describes the grammar

**T** describes a finite set of terminal symbols.

**V** describes a finite set of non-terminal symbols

**P** describes a set of production rules

**S** is the start symbol.

In CFG, the start symbol is used to derive the string. You can derive the string by repeatedly replacing a non-terminal by the right-hand side of the production until terminal symbols have replaced all non-terminal.

**Example:**

L= {wcwR | w € (a, b)\*}

**Production rules:**

1. S → aSa
2. S → bSb
3. S → c

Now check that abbcbba string can be derived from the given CFG.

1. S ⇒ aSa
2. S ⇒ abSba
3. S ⇒ abbSbba
4. S ⇒ abbcbba

By applying the production S → aSa, S → bSb recursively and finally applying the production S → c, we get the string abbcbba.

**Capabilities of CFG**

There are the various capabilities of CFG:

* Context-free grammar is useful to describe most of the programming languages.
* If the grammar is properly designed then an efficient parser can be constructed automatically.
* Using the features of associatively & precedence information, suitable grammars for expressions can be constructed.
* Context-free grammar is capable of describing nested structures like balanced parentheses, matching begin-end, corresponding if-then-else's & so on.

**Derivation**

The derivation is a sequence of production rules. It is used to get the input string through these production rules. During parsing, we have to make two decisions. These are as follows:

* We have to decide the non-terminal which is to be replaced.
* We must decide the production rule by which the non-terminal will be replaced.

We have two options to decide which non-terminal to replace with the production rule.

Left-most Derivation

In the leftmost derivation, the input is scanned and replaced with the production rule from left to right. So in left-most derivatives, we read the input string from left to right.

Example:

**Production rules:**

1. S = S + S
2. S = S - S
3. S = a | b |c

Input:

a - b + c

**The left-most derivation is:**

1. S = S + S
2. S = S - S + S
3. S = a - S + S
4. S = a - b + S
5. S = a - b + c

Right-most Derivation

In the rightmost derivation, the input is scanned and replaced with the production rule from right to left. So in right-most derivatives, we read the input string from right to left.

Example:

1. S = S + S
2. S = S - S
3. S = a | b |c

Input:

a - b + c

**The right-most derivation is:**

1. S = S - S
2. S = S - S + S
3. S = S - S + c
4. S = S - b + c
5. S = a - b + c

**Parse tree**

* Parse tree is the graphical representation of a symbol. The symbol can be terminal or non-terminal.
* In parsing, the string is derived using the start symbol. The root of the parse tree is that start symbol.
* It is the graphical representation of symbols that can be terminals or non-terminals.
* Parse tree follows the precedence of operators. The deepest sub-tree traversed first. So, the operator in the parent node has less precedence over the operator in the sub-tree.

The parse tree follows these points:

* All leaf nodes have to be terminals.
* All interior nodes have to be non-terminals.
* In-order traversal gives the original input string.

Example:

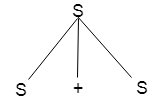
**Production rules:**

1. T= T + T | T \* T
2. T = a|b|c

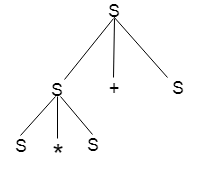
Input:

a \* b + c

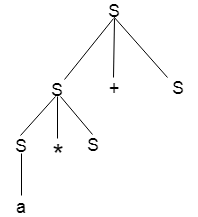
Step 1:



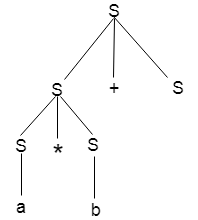
Step 2:



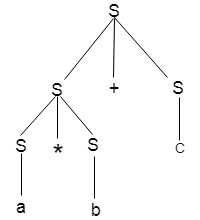
Step 3:



Step 4:



Step 5:



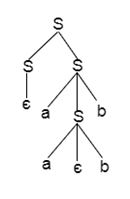
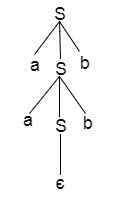
**Ambiguity**

A grammar is said to be ambiguous if there exists more than one leftmost derivation or more than one rightmost derivative or more than one parse tree for the given input string. If the grammar is not ambiguous then it is called unambiguous.

Example:

1. S = aSb | SS
2. S = ∈

For the string aabb, the above grammar generates two parse trees:

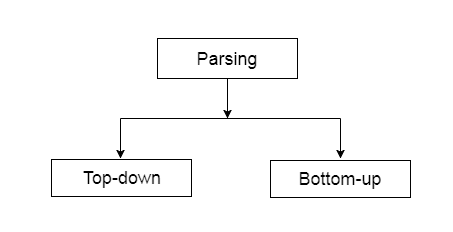
If the grammar has ambiguity then it is not good for a compiler construction. No method can automatically detect and remove the ambiguity but you can remove ambiguity by re-writing the whole grammar without ambiguity.

**Parser**

A parser is a compiler that is used to break the data into smaller elements coming from the lexical analysis phase.

A parser takes input in the form of a sequence of tokens and produces output in the form of the parse tree.

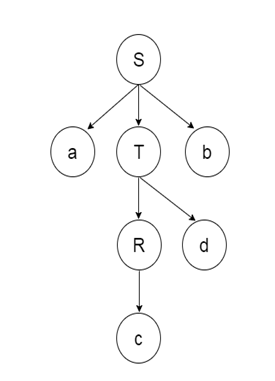
Parsing is of two types: top-down parsing and bottom-up parsing.



Top-down parsing

* The top-down parsing is known as **recursive parsing or predictive parsing**.
* Bottom-up parsing is used to construct a parse tree for an input string.
* In top-down parsing, the parsing starts from the start symbol and transforms it into the input symbol.

Parse Tree representation of input string "acdb" is as follows:



Bottom-up parsing

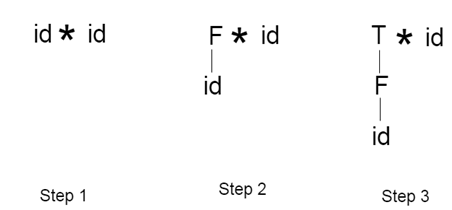
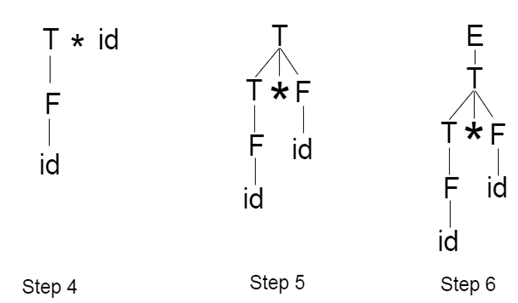
* Bottom-up parsing is also known as **shift-reduce parsing.**
* Bottom-up parsing is used to construct a parse tree for an input string.
* In the bottom-up parsing, the parsing starts with the input symbol and constructs the parse tree up to the start symbol by tracing out the rightmost derivations of the string in reverse.

Example

**Production**

1. E → T
2. T → T \* F
3. T → id
4. F → T
5. F → id

Parse Tree representation of input string "id \* id" is as follows:

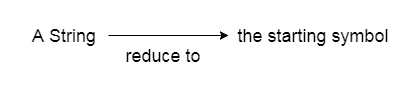
  


Bottom-up parsing is classified into various parsing. These are as follows:

1. Shift-Reduce Parsing
2. Operator Precedence Parsing
3. Table Driven LR Parsing
4. LR( 1 )
5. SLR( 1 )
6. CLR ( 1 )
7. LALR( 1 )

**Shift reduce parsing**

* Shift reduce parsing is a process of reducing a string to the start symbol of a grammar.
* Shift reduce parsing uses a stack to hold the grammar and an input tape to hold the string.



* Sift reduce parsing performs the two actions: shift and reduce. That's why it is known as shift reduces parsing.
* At the shift action, the current symbol in the input string is pushed to a stack.
* At each reduction, the symbols will be replaced by the non-terminals. The symbol is the right side of the production and the non-terminal is the left side of the production.

Example:

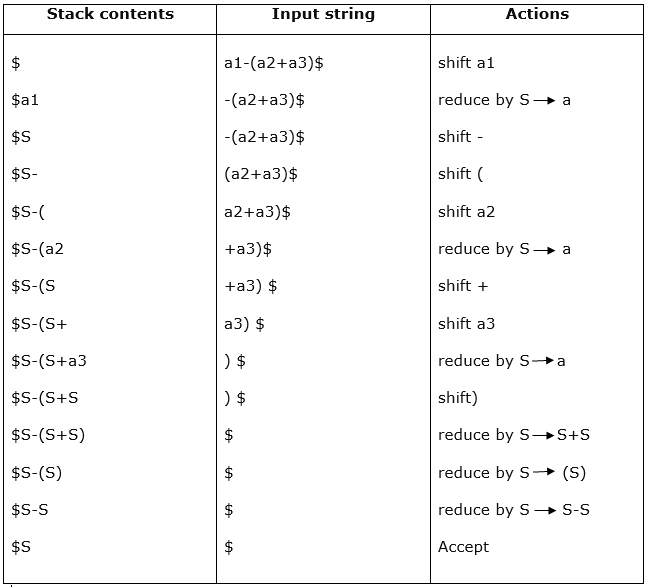
**Grammar:**

1. S → S+S
2. S → S-S
3. S → (S)
4. S → a

**Input string:**

1. a1-(a2+a3)

**Parsing table:**



There are two main categories of shift-reduce parsing as follows:

1. **Operator-Precedence Parsing**
2. LR-Parser

**Operator precedence parsing**

Operator precedence grammar is a kind of shift-reducing parsing method. It is applied to a small class of operator grammars.

A grammar is said to be operator precedence grammar if it has two properties:

No R.H.S. of any production has a∈.

* No two non-terminals are adjacent.

Operator precedence can only established between the terminals of the grammar. It ignores the non-terminal.

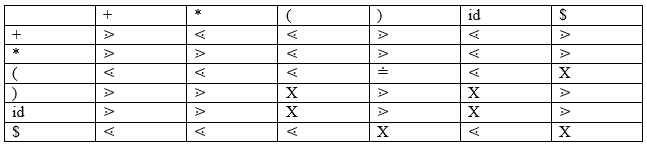
There are three operator precedence relations:

a ⋗ b means that terminal "a" has a higher precedence than terminal "b".

a ⋖ b means that terminal "a" has lower precedence than terminal "b".

a ≐ b means that the terminal "a" and "b" both have the same precedence.

Precedence table:



Parsing Action

* Both ends of the given input string, add the $ symbol.
* Now scan the input string from left to right until the ⋗ is encountered.
* Scan towards leftover all the equal precedence until the first leftmost ⋖ is encountered.
* Everything between left-most ⋖ and right most ⋗ is a handle.
* $ on $ means parsing is successful.

Example

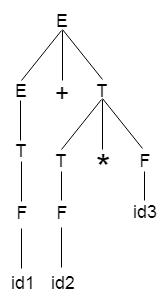
**Grammar:**

1. E → E+T/T
2. T → T\*F/F
3. F → id

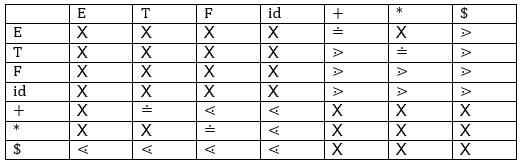
**Given string:**

1. w = id + id \* id

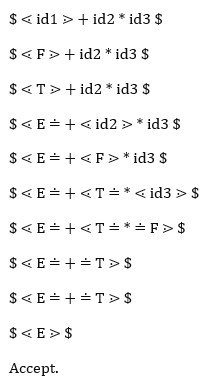
Let us consider a parse tree for it as follows:



Based on the above tree, we can design the following operator precedence table:



Now let us process the string with the help of the above precedence table:



LR Parser

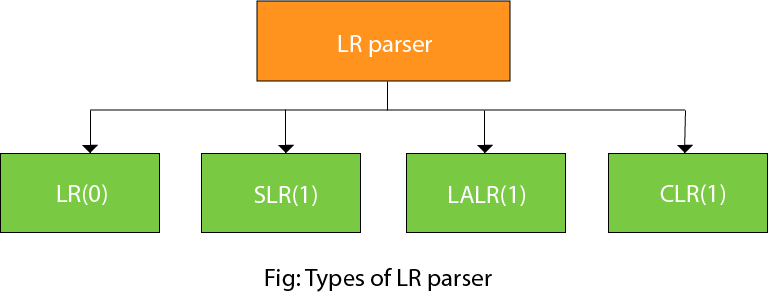
LR parsing is one type of bottom-up parsing. It is used to parse the large class of grammars.

In the LR parsing, "L" stands for left-to-right scanning of the input.

"R" stands for constructing a rightmost derivation in reverse.

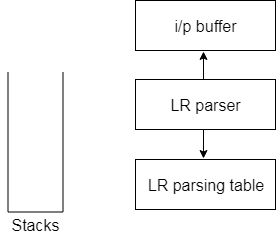
"K" is the number of input symbols of the look-ahead used to make a number of parsing decisions.

LR parsing is divided into four parts: LR (0) parsing, SLR parsing, CLR parsing, and LALR parsing.



LR algorithm:

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



**Fig: Block diagram of LR parser**

The input buffer is used to indicate the 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.

A parsing table is a two-dimensional array. It contains two parts: The action part and the go-to part.

LR (1) Parsing

Various steps involved in the LR (1) Parsing:

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

Augment Grammar

Augmented grammar G` will be generated if we add one more production in the given grammar G. It helps the parser to identify when to stop the parsing and announce the acceptance of the input.

Example

Given grammar

1. S → AA
2. A → aA | b

The Augment grammar G` is represented by

1. S`→ S
2. S → AA
3. A → aA | b

Canonical Collection of LR(0) items

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.

Example

Given grammar:

1. S → AA
2. A → aA | b

Add Augment Production and insert '•' symbol at the first position for every production in G

1. S` → •S
2. S → •AA
3. A → •aA
4. A → •b

I0 State:

Add Augment production to the I0 State and Compute the Closure

I0 = Closure (S` → •S)

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

**I0 =** S` → •S  
       S → •AA

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

**I0=** S` → •S  
       S → •AA  
       A → •aA  
       A → •b

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

Here, the Production is reduced so close the State.

**I1=** S` → S•

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

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

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

Go to (I2,a) = Closure (A → a•A) = (same as I3)

Go to (I2, b) = Closure (A → b•) = (same as I4)

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

Add productions starting with A in I3.

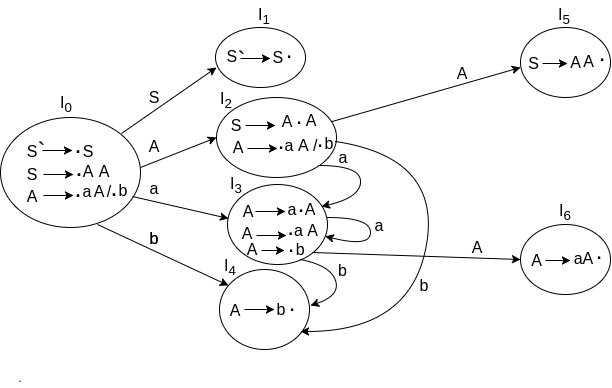
A → a•A  
A → •aA  
A → •b

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

**I4=** Go to (I0, b) = closure (A → b•) = A → b•  
**I5=** Go to (I2, A) = Closure (S → AA•) = SA → A•  
**I6=** Go to (I3, A) = Closure (A → aA•) = A → aA•

Drawing DFA:

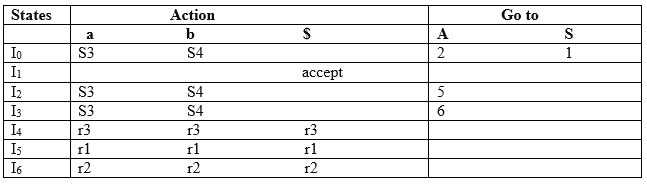
The DFA contains the 7 states I0 to I6.



LR(0) Table

ADVERTISEMENT

* If a state is going to some other state on a terminal then it corresponds to a shift move.
* If a state is going to some other state on a variable then it corresponds to go to move.
* If a state contains the final item in the particular row then write the reduce node completely.



**Explanation:**

* I0 on S is going to I1 so write it as 1.
* I0 on A is going to I2 so write it as 2.
* I2 on A is going to I5 so write it as 5.
* I3 on A is going to I6 so write it as 6.
* I0, I2and I3on a are going to I3 so write it as S3 which means that shift 3.
* I0, I2 and I3 on b are going to I4 so write it as S4 which means that shift 4.
* I4, I5 and I6 all states contains the final item because they contain • in the right most end. So rate the production as the production number.

Productions are numbered as follows:

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

* I1 contains the final item which drives(S` → S•), so action {I1, $} = Accept.
* I4 contains the final item which drives A → b• and that production corresponds to the production number 3 so write it as r3 in the entire row.
* I5 contains the final item which drives S → AA• and that production corresponds to the production number 1 so write it as r1 in the entire row.
* I6 contains the final item which drives A → aA• and that production corresponds to the production number 2 so write it as r2 in the entire row.

SLR (1) Parsing

SLR (1) refers to simple LR Parsing. It is the same as LR(0) parsing. The only difference is in the parsing table. To construct the SLR (1) parsing table, we use a canonical collection of LR (0) items.

In the SLR (1) parsing, we place the reduced move only in the following of the left-hand side.

Various steps involved in the SLR (1) Parsing:

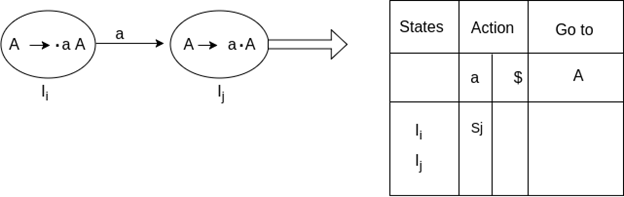
For the given input string write a context-free grammar

* Check the ambiguity of the grammar
* Add Augment production in the given grammar
* Create a Canonical collection of LR (0) items
* Draw a data flow diagram (DFA)
* Construct an SLR (1) parsing table

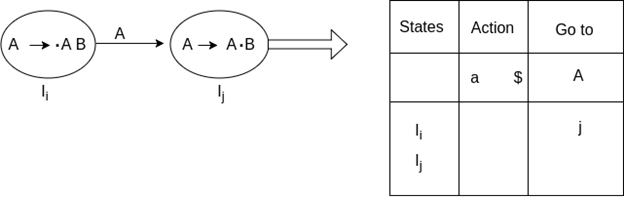
SLR (1) Table Construction

The steps use to construct SLR (1) Table is given below:

If a state (Ii) is going to some other state (Ij) on a terminal then it corresponds to a shift move in the action part.



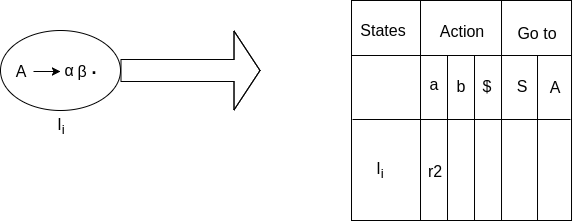
If a state (Ii) is going to some other state (Ij) on a variable then it corresponds to go to move in the Go to part.



If a state (Ii) contains the final item like A → ab• which has no transitions to the next state then the production is known as reduced production. For all terminals X in FOLLOW (A), write the reduced entry along with their production numbers.

Example

1. S -> •Aa
2. A->αβ•
3. Follow(S) = {$}
4. Follow (A) = {a}



SLR ( 1 ) Grammar

S → E  
E → E + T | T  
T → T \* F | F  
F → id

Add Augment Production and insert '•' symbol at the first position for every production in G

S` → •E  
E → •E + T  
E → •T  
T → •T \* F  
T → •F  
F → •id

**I0 State:**

Add Augment production to the I0 State and Compute the Closure

**I0 =** Closure (S` → •E)

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

**I0 =** S` → •E  
        E → •E + T  
        E → •T

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

**I0=** S` → •E  
       E → •E + T  
       E → •T  
       T → •T \* F  
       T → •F  
       F → •id

**I1=** Go to (I0, E) = closure (S` → E•, E → E• + T)  
**I2=** Go to (I0, T) = closure (E → T•T, T• → \* F)  
**I3=** Go to (I0, F) = Closure ( T → F• ) = T → F•  
**I4=** Go to (I0, id) = closure ( F → id•) = F → id•  
**I5=** Go to (I1, +) = Closure (E → E +•T)

Add all productions starting with T and F in I5 State because "." is followed by the non-terminal. So, the I5 State becomes

**I5 =** E → E +•T  
       T → •T \* F  
       T → •F  
       F → •id

Go to (I5, F) = Closure (T → F•) = (same as I3)  
Go to (I5, id) = Closure (F → id•) = (same as I4)

**I6=** Go to (I2, \*) = Closure (T → T \* •F)

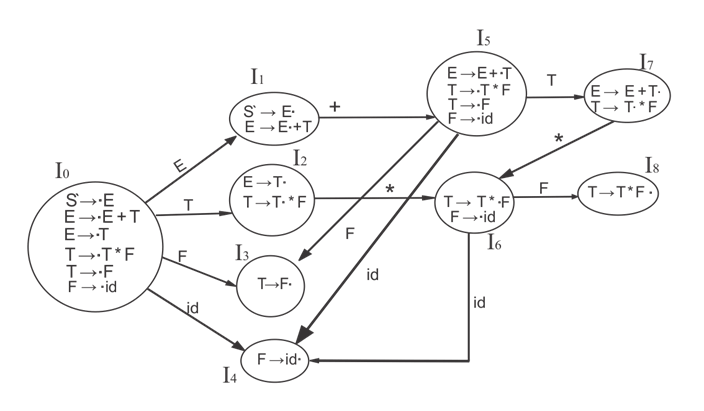
Add all productions starting with F in I6 State because "." is followed by the non-terminal. So, the I6 State becomes

**I6 =** T → T \* •F  
         F → •id

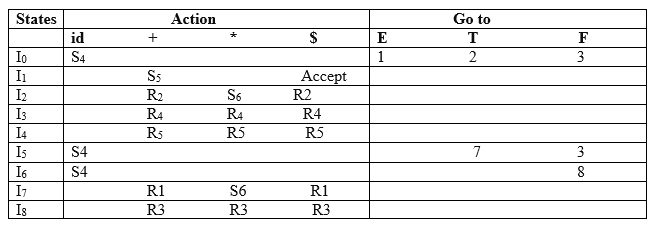
Go to (I6, id) = Closure (F → id•) = (same as I4)

**I7=** Go to (I5, T) = Closure (E → E + T•) = E → E + T•  
**I8=** Go to (I6, F) = Closure (T → T \* F•) = T → T \* F•

Drawing DFA:



SLR (1) Table



Explanation:

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

* I1 contains the final item which drives S → E• and follow (S) = {$}, so action {I1, $} = Accept
* I2 contains the final item which drives E → T• and follow (E) = {+, $}, so action {I2, +} = R2, action {I2, $} = R2
* I3 contains the final item which drives T → F• and follow (T) = {+, \*, $}, so action {I3, +} = R4, action {I3, \*} = R4, action {I3, $} = R4
* I4 contains the final item which drives F → id• and follow (F) = {+, \*, $}, so action {I4, +} = R5, action {I4, \*} = R5, action {I4, $} = R5
* I7 contains the final item which drives E → E + T• and follow (E) = {+, $}, so action {I7, +} = R1, action {I7, $} = R1
* I8 contains the final item which drives T → T \* F• and follow (T) = {+, \*, $}, so action {I8, +} = R3, action {I8, \*} = R3, action {I8, $} = R3.

CLR (1) Parsing

CLR refers to canonical lookahead. CLR parsing uses the canonical collection of LR (1) items to build the CLR (1) parsing table. CLR (1) parsing table produces a larger number of states as compared to the SLR (1) parsing.

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

Various steps involved in the CLR (1) Parsing:

For the given input string write a context-free grammar

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

**LR (1) item**

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

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

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

The look-ahead always adds $ symbol for the argument production.

Example

**CLR ( 1 ) Grammar**

1. S → AA
2. A → aA
3. A → b

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

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

**I0 State:**

Add Augment production to the I0 State and Compute the Closure

**I0 =** Closure (S` → •S)

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

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

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

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

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

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

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

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

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

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

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

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

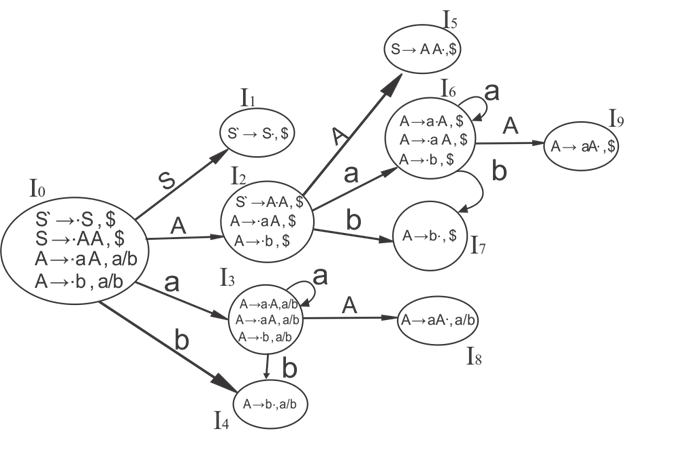
Add all productions starting with A in I6 State because "." is followed by the non-terminal. So, the I6 State becomes

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

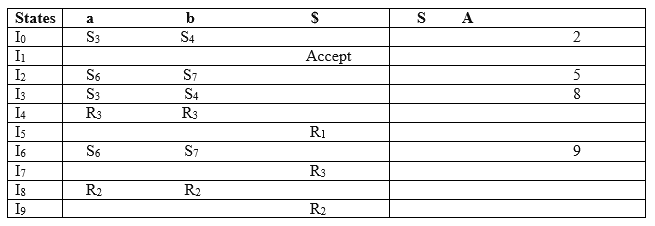
Go to (I6, a) = Closure (A → a•A, $) = (same as I6)  
Go to (I6, b) = Closure (A → b•, $) = (same as I7)

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

Drawing DFA:



CLR (1) Parsing table:



Productions are numbered as follows:

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

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

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

LALR (1) Parsing:

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

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

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

Example

**LALR ( 1 ) Grammar**

1. S → AA
2. A  → aA
3. A → b

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

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

**I0 State:**

Add Augment production to the I0 State and Compute the ClosureL

**I0 =** Closure (S` → •S)

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

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

ADVERTISEMENT

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

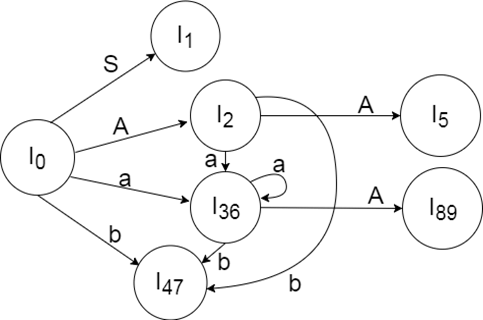
The I4 and I7 are the same but they differ only in their look ahead, so we can combine them and call as I47.

**I47 =** {A → b•, a/b/$}

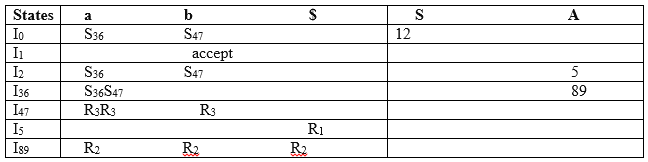
The I8 and I9 are the same but they differ only in their look ahead, so we can combine them and call as I89.

**I89 =** {A → aA•, a/b/$}

Drawing DFA:



LALR (1) Parsing table:



YACC

* YACC stands for **Yet Another Compiler Compiler**.
* YACC provides a tool to produce a parser for a given grammar.
* YACC is a program designed to compile a LALR (1) grammar.
* It is used to produce the source code of the syntactic analyzer of the language produced by LALR (1) grammar.
* The input of YACC is the rule or grammar and the output is a C program.

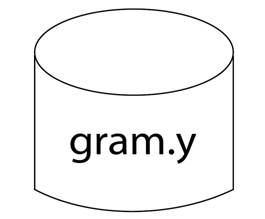
These are some points about YACC:

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

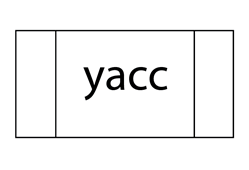
**Output: A parser y.tab.c (yacc)**

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

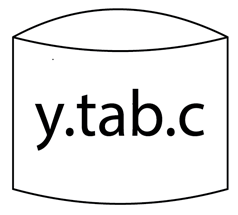
The basic operational sequence is as follows:



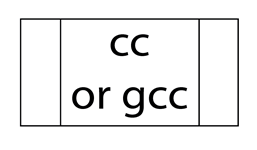
This file contains the desired grammar in YACC format.



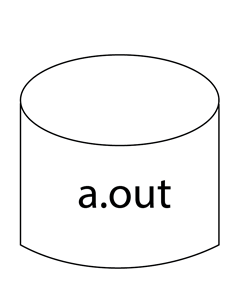
It shows the YACC program.



It is the c source program created by YACC.



C Compiler



Executable file that will parse grammar given in gram.Y