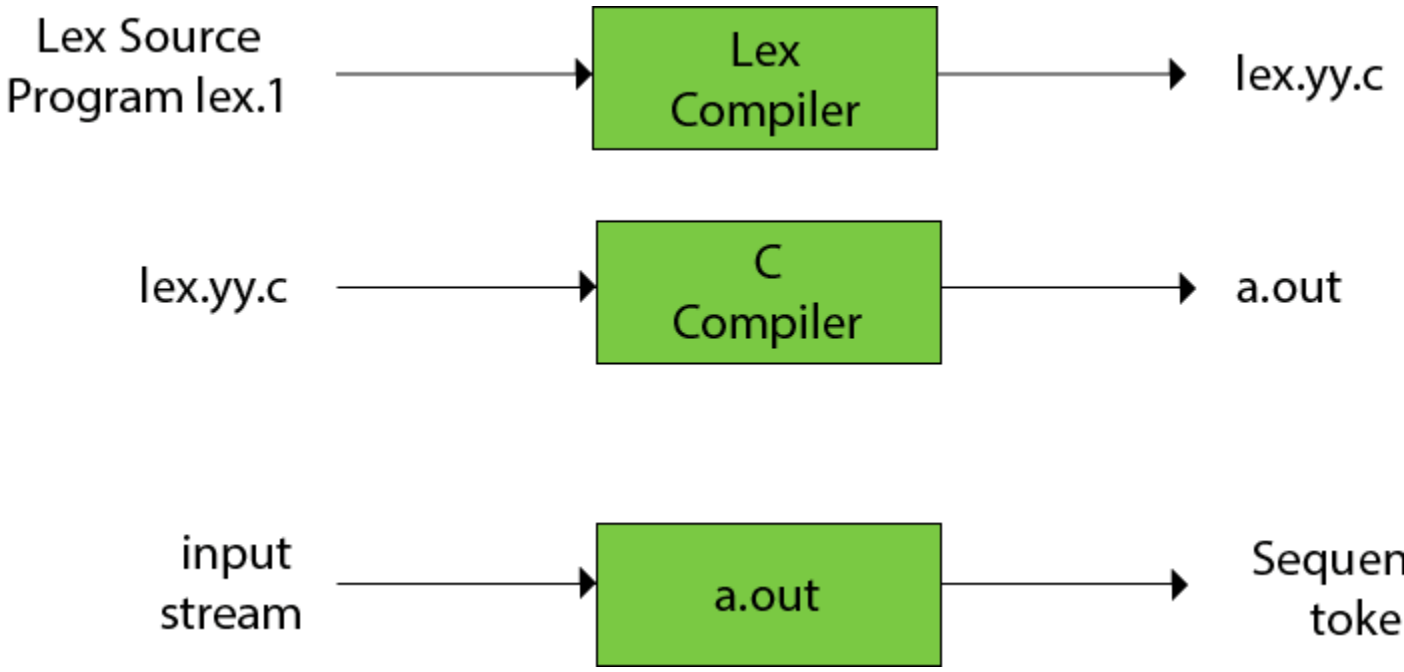
LEX

- Lex is a program that generates lexical analyzer. It is used with YACC parser generator.
- The lexical analyzer 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 analyzer in the C program.

The function of Lex is as follows:

- Firstly lexical analyzer 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 analyzer that transforms an input stream into a sequence of tokens.



Introduction to Compiler

- A compiler is a translator that converts the high-level language into the machine language.
- High-level language is written by a developer and machine language can be understood by the processor.
- Compiler is used to show errors to the programmer.
- The main purpose of compiler is to change the code written in one language without changing the meaning of the program.
- When you execute a program which is written in HLL programming language then it executes into two parts.
- In the first part, the source program compiled and translated into the object program (low level language).
- In the second part, object program translated into the target program through the assembler.

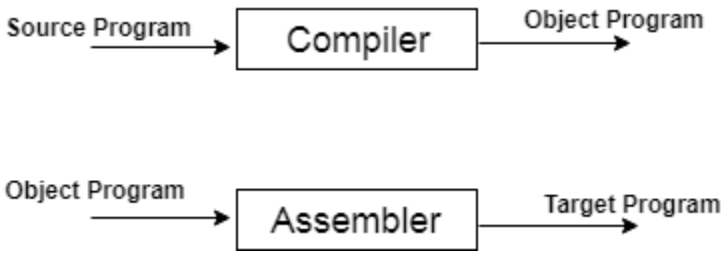


Fig: Execution process of source program in Compiler

next →← prev

Compiler Phases

The compilation process contains the sequence of various phases. Each phase takes source program in one representation and produces output in another representation. Each phase takes input from its previous stage.

There are the various phases of compiler:

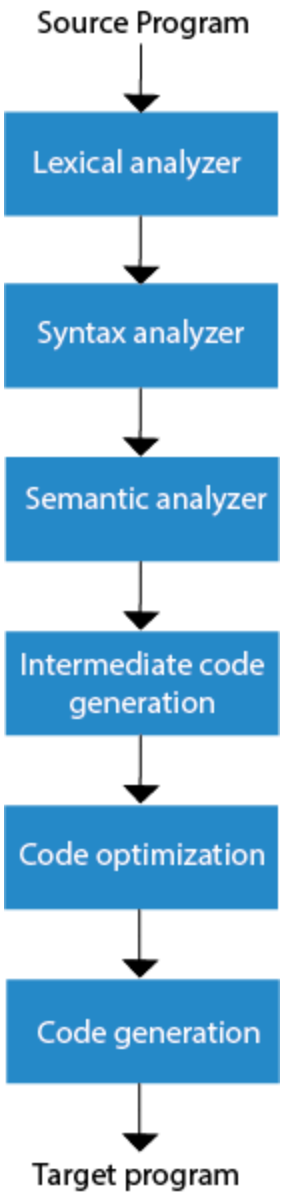


Fig: phases of compiler

next →← prev

Compiler Passes

Pass is a complete traversal of the source program. Compiler has two passes to traverse the source program.

Multi-pass Compiler

- Multi pass compiler is used to process the source code of a program several times.
- In the first pass, compiler can read the source program, scan it, extract the tokens and store the result in an output file.
- In the second pass, compiler can read the output file produced by first pass, build the syntactic tree and perform the syntactical analysis. The output of this phase is a file that contains the syntactical tree.
- In the third pass, compiler can read the output file produced by second pass and check that the tree follows the rules of language or not. The output of semantic analysis phase is the annotated tree syntax.
- This pass is going on, until the target output is produced.

One-pass Compiler

- One-pass compiler is used to traverse the program only once. The one-pass compiler passes only once through the parts of each compilation unit. It translates each part into its final machine code.
- In the one pass compiler, when the line source is processed, it is scanned and the token is extracted.
- Then the syntax of each line is analyzed and the tree structure is build. After the semantic part, the code is generated.
- The same process is repeated for each line of code until the entire program is compiled.

next →← prev

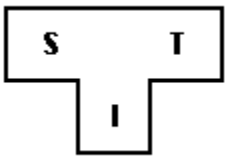
Bootstrapping

- Bootstrapping is widely used in the compilation development.
- Bootstrapping is used to produce a self-hosting compiler. Self-hosting compiler is a type of compiler that can compile its own source code.
- Bootstrap compiler is used to compile the compiler and then you can use this compiled compiler to compile everything else as well as future versions of itself.

A compiler can be characterized by three languages:

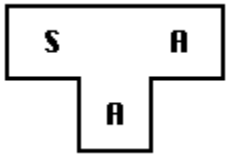
1. Source Language
2. Target Language
3. Implementation Language

The T- diagram shows a compiler ${}^{S}C_{I}^{T}$ for Source S, Target T, implemented in I.

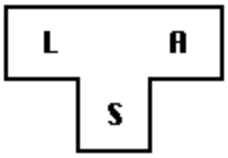


Follow some steps to produce a new language **L** for machine **A**:

1. Create a compiler ${}^{S}C_{A}^{A}$ for subset, S of the desired language, L using language "A" and that compiler runs on machine A.

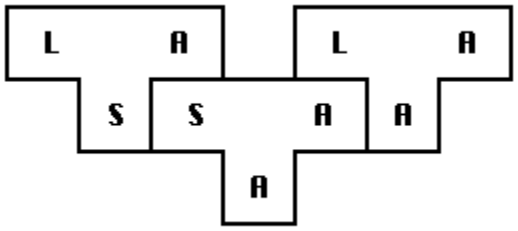


2. Create a compiler ${}^{L}C_{S}^{A}$ for language L written in a subset of L.



3. Compile ${}^{L}C_{S}^{A}$ using the compiler ${}^{S}C_{A}^{A}$ to obtain ${}^{L}C_{A}^{A}$. ${}^{L}C_{A}^{A}$ is a compiler for language L, which runs on machine A and produces code for machine A.

$$ {}^{L}C_{S}^{A} \rightarrow {}^{S}C_{A}^{A} \rightarrow {}^{L}C_{A}^{A} $$



The process described by the T-diagrams is called bootstrapping

Finite state machine

- Finite state machine is used to recognize patterns.
- Finite automata machine takes the string of symbol as input and changes its state accordingly. In the input, when a desired symbol is found then the transition occurs.
- While transition, the automata can either move to the next state or stay in the same state.
- FA has two states: accept state or reject state. When the input string is successfully processed and the automata reached its final state then it will accept.

A finite automata consists of following:

Q: finite set of states
Σ: finite set of input symbol
q0: initial state
F: final state
δ: Transition function

Transition function can be define as

1. δ: Q x Σ →Q

FA is characterized into two ways:

1. DFA (finite automata)
2. NDFA (non deterministic finite automata)

next →‹– prev

Regular expression

- Regular expression is a sequence of pattern that defines a string. It is used to denote regular languages.
- It is also used to match character combinations in strings. String searching algorithm used this pattern to find the operations on string.
- In regular expression, x* means zero or more occurrence of x. It can generate {e, x, xx, xxx, xxxx,.....}
- In regular expression, x+ means one or more occurrence of x. It can generate {x, xx, xxx, xxxx,.....}

Operations on Regular Language

next →‹– prev

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>

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:

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 efficientparser 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

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

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

We have two options to decide which non-terminal to be replaced with production rule

Parse tree

- Parse tree is the graphical representation of 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 symbol 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 original input string.

Ambiguity

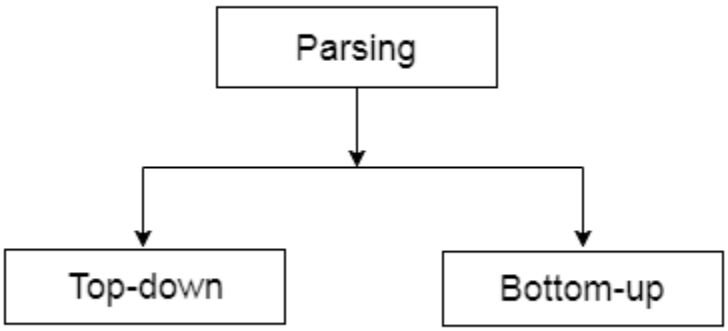
A grammar is said to be ambiguous if there exists more than one leftmost derivation or more than one rightmost derivative or more than one parse tree for the given input string. If the grammar is not ambiguous then it is called unambiguous.

Parser

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.

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



Top down paring

- 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 the top down parsing, the parsing starts from the start symbol and transform it into the input symbol.

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

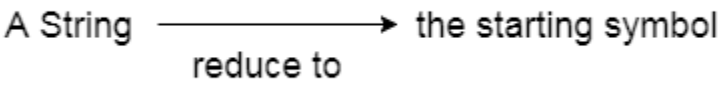
Bottom up parsing

- 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 construct the parse tree up to the start symbol by tracing out the rightmost derivations of string in reverse.

Shift reduce parsing

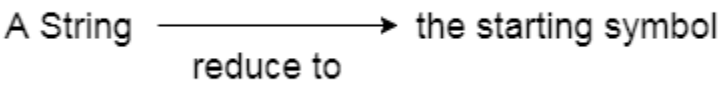
- Shift reduce parsing is a process of reducing a string to the start symbol of a grammar.
- Shift reduce parsing uses a stack to hold the grammar and an input tape to hold the string.



- 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 replaced by the non-terminals. The symbol is the right side of the production and non-terminal is the left side of the production.

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 replaced by the non-terminals. The symbol is the right side of the production and non-terminal is the left side of the production.

Operator precedence parsing

Operator precedence grammar is kinds of shift reduce 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 the three operator precedence relations:

a ⩾ b means that terminal "a" has the higher precedence than terminal "b".

a ⩽ b means that terminal "a" has the lower precedence than terminal "b".

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

Precedence table:

	+	*	(	)	id	$
+	⩾	⩽	⩽	⩾	⩽	⩾
*	⩾	⩾	⩽	⩾	⩽	⩾
(	⩽	⩽	⩽	≐	⩽	X
)	⩾	⩾	X	⩾	X	⩾
id	⩾	⩾	X	⩾	X	⩾
$	⩽	⩽	⩽	X	⩽	X

Parsing Action

- Both end of the given input string, add the $ symbol.
- Now scan the input string from left right until the ⩾ is encountered.
- Scan towards left over all the equal precedence until the first left most ⩽ is encountered.
- Everything between left most ⩽ and right most ⩾ is a handle.

- $ on $ means parsing is successful

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: LR (0) parsing, SLR parsing, CLR parsing and LALR parsing.

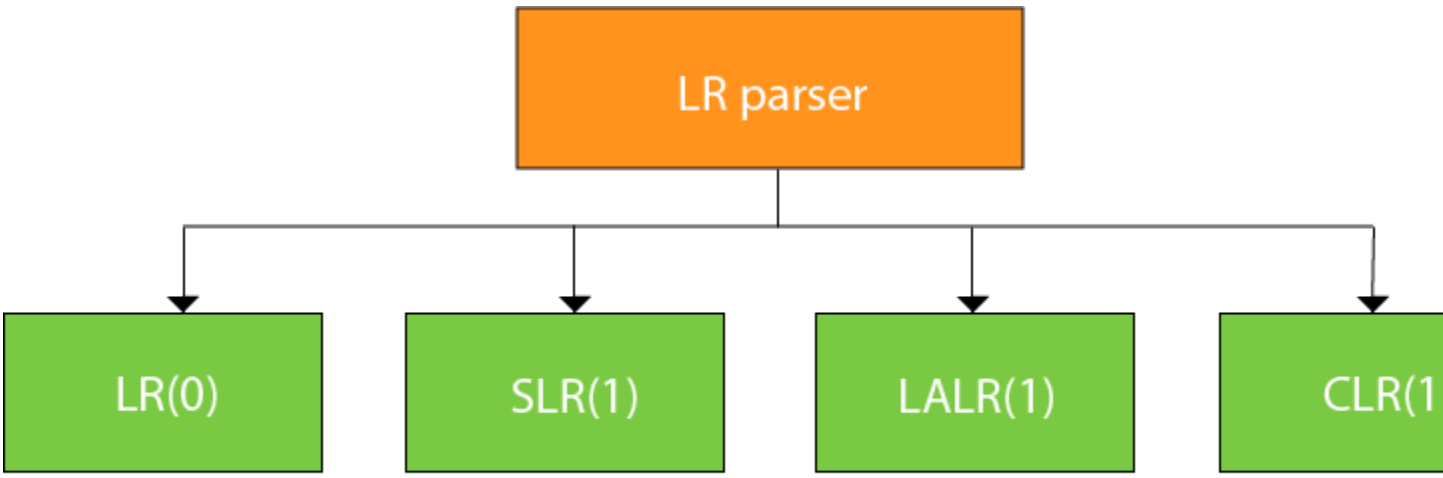


Fig: Types of LR parser

LR algorithm:

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.

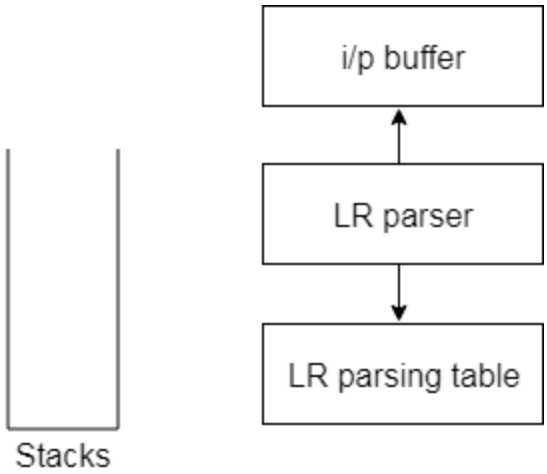


Fig: Block diagram of LR parser

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.

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 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.

next →← prev

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.

next →← prev

SLR (1) Parsing

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

In the SLR (1) parsing, we place the reduce move only in the follow of left hand side.

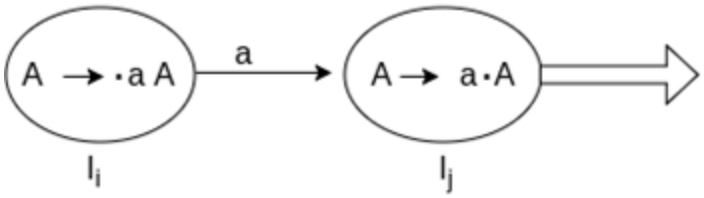
Various steps involved in the SLR (1) Parsing:

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

SLR (1) Table Construction

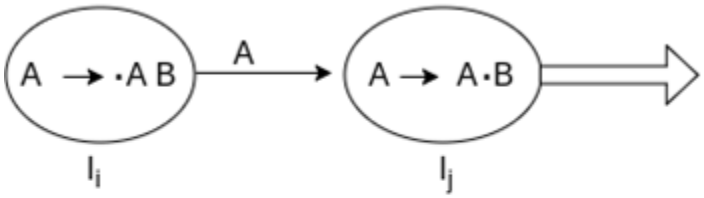
The steps which use to construct SLR (1) Table is given below:

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



States	Action		Go to
	a	$	A
I_i I_j	Sj		

If a state (I_i) is going to some other state (I_j) on a variable then it correspond to go to move in the Go to part.



States	Action		Go to
	a	$	A
I_i I_j			j

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

next →← prev

CLR (1) Parsing

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

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

Various steps involved in the CLR (1) Parsing:

- For the given input string write a context free grammar

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

LR (1) item

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

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

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

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

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 which have same productions but different look ahead are combined to form a single set of items

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

Automatic Parser Generator

YACC is an automatic tool that generates the parser program.

As we have discussed YACC in the first unit of this tutorial so you can go through the concepts again to make things more clear.

- YACC