

# COMPILER DESIGN

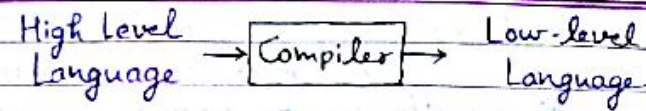
[www.ankurgupta.net](http://www.ankurgupta.net)

\* These notes have been prepared from the following book:-

Principles of Compiler Design  
by Aho, Ullman

Kindly refer the above book for more details.

## Compiler Design



Major parts of a compiler:-

Two major parts:-

(1) Analysis Phase:-

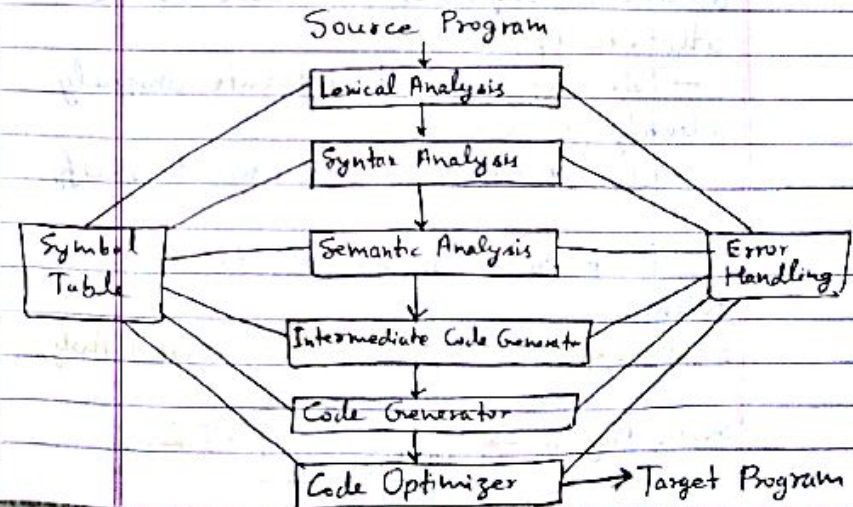
An intermediate representation is created from the given source program.

- (i) Lexical Analyzer,
- (ii) Syntax Analyzer,
- (iii) Semantic Analyzer are phases in it.

(2) Synthesis Phase:-

Equivalent target program is created from this intermediate representation.

- (i) Intermediate Code Generator,
- (ii) Code Generator
- (iii) Code Optimizer are phases in it.





### Phases vs Passes :-

Passes refer to the number of times the compiler has to traverse through the entire program. Several phases are grouped into one pass.

### Cross Compiler :-

A compiler that runs on one machine (A) and produces a code for another machine (B).

### Lexical Analysis :-

→ Lexical Analyzer reads a source program character by character to produce tokens.

→ A token can represent more than one lexeme, additional information should be held for that specific lexeme. This additional information is called as the attribute of the token.

→ Token type and its attribute uniquely identify a lexeme.

→ Regular expressions are used to specify patterns.

→ Finite Automata is used to recognise tokens.

Regular Expression → Lexical Analyzer Generator → Lexical Analyzer

Source Program → Lexical Analyzer → Tokens.

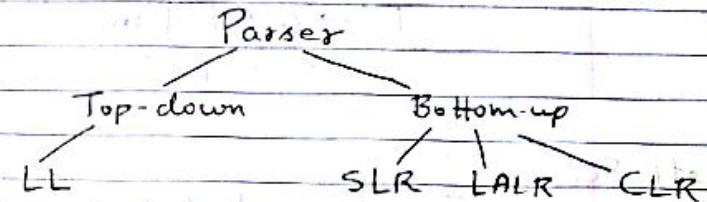
## Syntax Analysis

→ Syntax Analyzer is also known as parser.

→ The syntax of a programming language is described by a CFG.

→ A CFG gives a precise syntactic specification of a programming language.

→ Parser works on a stream of tokens.



→ Top-down parsers try to find the left-most derivation of the given source program.

→ Bottom-up parsers try to find the right-most derivation of the given source program in reverse order.

→ Both top down and bottom-up parsers scan the input from left-to-right.

→ Bottom-up parsers are called as Shift-Reduce Parsers.



### Ambiguity:-

→ An unambiguous grammar should be written to eliminate the ambiguity.

Example:- Disambiguate the grammar:-

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

$\wedge$  (Right to left)

$*, +$  (Left to right).

Solution:-

$$E \rightarrow E + T \mid T$$

$$T \rightarrow T * F \mid F$$

$$F \rightarrow G \wedge F \mid G$$

$$G \rightarrow id \mid (E)$$

### Left Recursion:-

Top-down techniques can not handle left-recursive grammars.

### Left Factoring:-

A predictive parser (a topdown parser without backtracking) insists that the grammar must be left-factored.

Example:-  $A \rightarrow abB \mid aB \mid cdg \mid cdeB \mid cdfB$

↓

$$A \rightarrow aA' \mid cdg \mid cdeB \mid cdfB$$

$$A' \rightarrow bB \mid B$$

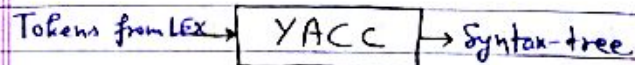
↓

$$A \rightarrow aA' \mid cdA'', \quad A' \rightarrow bB \mid B,$$

$$A'' \rightarrow g \mid eB \mid fB$$

### YACC:-

YACC generates C-code for a syntax analyzer, or parser.



### Top-Down Parsing:-

#### (1) Recursive-Descent Parsing:-

- Backtracking is needed, Not efficient.
- General purpose, not widely used.

#### (2) Predictive-Parsing:-

- No backtracking, efficient
- Uses LL(1) grammars.

### Predictive Parser:-

→ It can uniquely choose a production rule by just looking the current symbol.

→ Non-Recursive (Table Driven) Predictive parser is also known as LL(1) parser.

→ We eliminate the left recursion in the grammar, and left-factor it. But it may not be suitable for predictive parsing (not LL(1) grammar).

### Two types:-

- (1) Recursive Predictive Parsing
- (2) Non-Recursive Predictive Parsing LL(1)



## Recursive Predictive Parsing:-

Each non-terminal corresponds to a procedure.

Example:-  $A \rightarrow aBb \mid bAB$

proc A {

case of the current token {

'a':-match the current token with a,  
and move to next token.

- call B

- match the current token with  
b, and move to the next token.

'b' - match the current token with  
b and move to the next token.

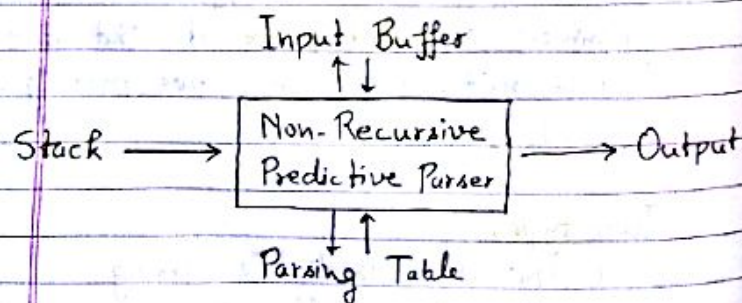
- call 'A'

- call 'B'

## Non-Recursive Predictive Parsing:-

→ Table-driven parser

→ Also known as LL(1) parser.



Example:-  $S \rightarrow aBa$ ,  $B \rightarrow bB \mid \Lambda$

	a	b	\$	
S	$S \rightarrow aBa$			← LL(1) Parsing Table.
B	$B \rightarrow \Lambda$	$B \rightarrow bB$		

Solution:-

Stack	Input	Output
\$S	abba\$	$S \rightarrow aBa$
\$aBa	abba\$	
\$aB	bba\$	$B \rightarrow bB$
\$aBb	bba\$	
\$aB	ba\$	$B \rightarrow bB$
\$aBb	ba\$	
\$aB	a\$	$B \rightarrow \Lambda$
\$a	a\$	
\$	\$	Accept.

## Constructing LL(1) Parsing Tables:-

→ Two functions are used:-

FIRST & FOLLOW

→  $FIRST(\alpha)$  is a set of the terminal symbols which occur as first symbols in strings derived from  $\alpha$ .

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



### Computing FIRST for any string $X$ :-

→ If  $X$  is a terminal symbol:-  
 $FIRST(X) = \{X\}$

→ If  $X$  is a non-terminal symbol,  
and  $X \rightarrow A$  is a production rule, then  
 $A$  is in  $FIRST(X)$ .

→ If  $X$  is a non-terminal symbol,  
and  $X \rightarrow X_1 X_2 \dots X_n$  is a production rule, then

(i) If a terminal symbol  $a$  in  
 $FIRST(X_i)$  and  $A$  is in all  $FIRST(X_j)$

for  $j = 1, 2, \dots, i-1$ , then  $a$  is in  $FIRST(X)$ .

(ii) If  $A$  is in all  $FIRST(X_i)$  for  
 $i = 1, \dots, n$ , then  $A$  is in  $FIRST(X)$ .

### Computing FOLLOW (for non-terminals):-

→ If  $S$  is the start symbol, then:-  
 $\$$  is in  $FOLLOW(S)$ .

→ If  $A \rightarrow \alpha B \beta$  is a production rule, then  
everything in  $FIRST(\beta)$  is  $FOLLOW(B)$   
except  $A$ .

→ If  $(A \rightarrow \alpha B)$  or  $(A \rightarrow \alpha B \beta)$  and  $A$  is  
in  $FIRST(\beta)$ , then:-  
everything in  $FOLLOW(A)$  is in  
 $FOLLOW(B)$ .

### Algorithm for constructing LL(1) Parsing Table:-

For each production rule  $A \rightarrow \alpha$  of a  
grammar  $G$ :-

→ For each terminal  $a$  in  $FIRST(\alpha)$ ,  
add  $A \rightarrow \alpha$  to  $M[A, a]$

→ If  $A$  is in  $FIRST(\alpha)$ , then:-  
for each terminal  $a$  in  $FOLLOW(A)$   
add  $A \rightarrow \alpha$  to  $M[A, a]$

→ If  $A$  is in  $FIRST(\alpha)$  and  $\$$  in  
 $FOLLOW(A)$ , then:-  
add  $A \rightarrow \alpha$  to  $M[A, \$]$ .

All other undefined entries of the  
parsing table are error entries.

### LL(1) Grammars:-

LL(1) → One input symbol  
used as a look-ahead  
symbol to determine  
parser action.  
Input scanned from  
left to right  
Left-most  
derivation

→ A grammar whose parsing table has no  
multiple entries is said to be LL(1) grammar.

→ A left-recursive grammar can not be a  
LL(1) grammar.

→ A grammar is not left factored, it cannot  
be a LL(1) grammar.

→ An ambiguous grammar can not be a  
LL(1) grammar.



Example:-  $E \rightarrow TE'$ ,  $E' \rightarrow +TE' \mid \Lambda$ ,  
 $T \rightarrow FT'$ ,  $T' \rightarrow *FT' \mid \Lambda$ ,  
 $F \rightarrow (E) \mid id$

Construct LL(1) parsing table:-

Solution:-

$FIRST(F) = \{ (, id \}$ ,  
 $FIRST(T') = \{ *, \Lambda \}$ ,  $FIRST(T) = \{ (, id \}$   
 $FIRST(E') = \{ +, \Lambda \}$ ,  $FIRST(E) = \{ (, id \}$   
 $FIRST(TE') = \{ (, id \}$ ,  $FIRST(+TE') = \{ + \}$ ,  
 $FIRST(\Lambda) = \{ \Lambda \}$ ,  $FIRST(FT') = \{ (, id \}$ ,  
 $FIRST(*FT') = \{ * \}$ ,  $FIRST(CE)) = \{ ( \}$ ,  
 $FIRST(id) = \{ id \}$

$FOLLOW(E) = \{ \$, ) \}$ ,  $FOLLOW(E') = \{ \$, ) \}$ ,  
 $FOLLOW(T) = \{ +, ), \$ \}$ ,  $FOLLOW(T') = \{ +, ), \$ \}$ ,  
 $FOLLOW(F) = \{ *, +, ), \$ \}$

	id	+	*	(	)	\$
E	$E \rightarrow TE'$			$E \rightarrow TE'$		
E'		$E' \rightarrow +TE'$			$E' \rightarrow \Lambda$	$E' \rightarrow \Lambda$
T	$T \rightarrow FT'$			$T \rightarrow FT'$		
T'		$T' \rightarrow \Lambda$	$T' \rightarrow *FT'$		$T' \rightarrow \Lambda$	$T' \rightarrow \Lambda$
F	$F \rightarrow id$			$F \rightarrow (E)$		

Bottom-Up Parsing:-

It is also known as shift-reduce parsing.

Handle:-

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

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

Conflicts during Shift Reduce Parsing:-

(1) Shift/Reduce Conflict:-

Whether make a shift operation or a reduction.

(2) Reduce/Reduce Conflict:-

Parser cannot decide which of several reductions to make.

→ If a shift-reduce parser can not be used for a grammar, that grammar is called as Non-LR(k) grammar.



## LR(R) Grammars:-



→ An ambiguous grammar can never be a LR grammar.

## Types of Shift-Reduce Parsing:-

Two main

categories:-

(1) Operator - Precedence Parser

(2) LR-Parsers:-

Covers wide range of

grammars:-

(i) SLR - Simple LR

(ii) CLR - Most general LR (Canonical LR)

(iii) LALR - Intermediate LR (Look ahead)

SLR, CLR, LALR work same, only their parsing tables are different.

## Operator Precedence Parsing:-

### Operator Grammar:-

→ Small, but an important class of grammars.

In an operator grammar, no production rule can have:-

(i)  $\wedge$  at right side,

(ii) two adjacent non-terminals at the right side.

### Using precedence relations to find handles:-

(1) Scan the string from left end until the first  $>$  is encountered.

(2) Then scan backwards (to the left) over any  $=$  until a  $<$  is encountered.

(3) The handle contains everything to left of the first  $>$  and to the right of the  $<$  is encountered.

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



Example:-  $E \rightarrow E + E$ ,  $E \rightarrow E * E$ ,  $E \rightarrow id$

Stack	Input	Action
\$	id + id * id \$	shift
\$ id	+ id * id \$	Reduce
\$ E	+ id * id \$	Shift
\$ E +	id * id \$	Shift
\$ E + id	* id \$	Reduce
\$ E + E	* id \$	Shift
\$ E + E *	id \$	Shift
\$ E + E * id	\$	Reduce
\$ E + E * E	\$	Reduce
\$ E + E	\$	Reduce
\$ E	\$	Accept.

Advantages:-

Powerful enough for expressions in programming languages.

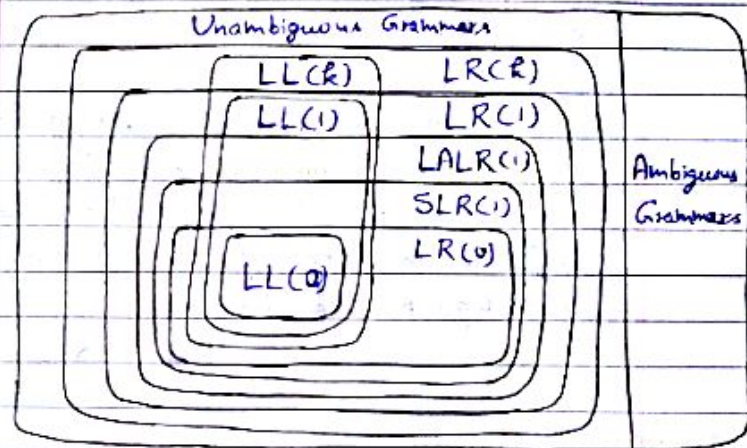
Disadvantages:-

- (1) It can not handle the unary minus (the lexical analyzer should handle that).
- (2) Difficult to decide which language is recognized by the grammar.

LR-Parsing:-

→ LR parsing is a non-backtracking shift reduce parsing.

→ An LR-parser can detect a syntactic error as soon as it is possible to do so in a left-to-right scan of the input.



$$LL(2) \subset LR(2)$$

$$LR(0) \subset SLR(1) \subset LALR(1) \subset LR(1) \subset LR(2)$$

Canonical LR is also referred to as LR(1).



→ LALR parsers are often used in practice because LALR parsing tables are smaller than Canonical LR parsing tables.

→ The number of states in SLR and LALR parsing tables for a grammar  $G$  are equal.

→ We can create LR-parsing tables for ambiguous grammars, but they will have conflicts. We can resolve these conflicts in favor of one of them to disambiguate the grammar.

### Definition of Handle:-

A handle of a right sentential form  $\gamma$  is a production rule  $A \rightarrow \beta$  and a position of  $\gamma$  where the string  $\beta$  may be found and replaced by  $A$ .

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

Intermediate Codes are machine independent codes, but they are close to machine instructions.

Syntax trees, Postfix Notation, three-address codes can be used as intermediate language.

### Three Address Code:-

$$A = -B * (C + D)$$

Three address code is as follows:-

$$T_1 = -B$$

$$T_2 = C + D$$

$$T_3 = T_1 * T_2$$

$$A = T_3$$

### Quadruple:-

	Operator	Operand1	Operand2	Result
(1)	-	B		$T_1$
(2)	+	C	D	$T_2$
(3)	*	$T_1$	$T_2$	$T_3$
(4)	=	$T_3$		A

### Triple:-

	Operator	Operand1	Operand2
(1)	-	B	
(2)	+	C	D
(3)	*	(1)	(2)
(4)	=	A	(3)



Indirect Triple:-

Statement

(0) (56)

(1) (57)

(2) (58)

(3) (59)

Operator      Operand1      Operand2

(56)      -      B

(57)      +      C      D

(58)      \*      (56)      (57)

(59)      =      A      (58)

www.ankurgupta.net

Code OptimizationCode (1) Common Sub-expression Elimination:- $a = (b * c)$  $T_1 = b * c$  $\Rightarrow$  $a = T_1$  $d = (b * c) + x - y$  $d = T_1 + x - y$ (2) Compile Time Evaluation:-(i)  $A = 2 * (22.0 / 7.0) * 8$ Perform  $2 * (22.0 / 7.0)$  at compile time.(ii)  $x = 12.4$ Evaluate  $x / 2.3$  as $\Rightarrow$  $12.4 / 2.3$  at compile time. $y = x / 2.3$ ~~Dead Code Elimination~~(3) Variable Propagation:- $c = a * b$  $c = a * b$  $x = a$  $x = a$  $\Rightarrow$  $d = x * b + 4$  $a = a * b + 4$ 

Here, after variable propagation,  $a * b$  and  $x * b$  will be identified as common subexpressions.



(4) Dead Code Elimination:-

Variable propagation often leads to making assignment statement into dead code.

```

c = a * b
x = a
|
|
d = a * b + a
    ⇒
c = a * b
|
|
d = a * b + a
  
```

(5) Code Motion:-

→ Reduce the evaluation frequency of expressions.  
→ Bring loop-invariant statements out of the loop.

```

a = 200;
while (a > 0)
{
    b = x + y;
    if (a * b == 0)
        printf("x, y, a");
}
    ⇒
a = 200;
b = x + y;
while (a > 0)
{
    if (a * b == 0)
        printf("x, y, a");
}
  
```

(6) Induction Variables and Strength Reduction:-

→ An induction variable is used in loop for the following kind of assignment:-  
 $i = i + \text{constant}$

→ Strength reduction means replacing the high strength operator by low strength.

```

i = 1;
while (i < 10)
{
    y = i * 4;
}
    ⇒
i = 1;
t = 4;
while (t < 40)
{
    y = t;
    t = t + 4;
}
  
```



www.ankurgupta.net

## Typed versus Untyped Languages :-

A language is typed if the specification of every operation defines types of data to which the operation is applicable.

In contrast, an untyped language, such as most assembly languages, allows any operation to be performed on any data, which are generally considered to be sequences of bits of various lengths.

## Weak and Strong Typing :-

Weak typing allows a value of one type to be treated as another.

Strong typing prevents the above.

Strongly typed languages are often termed as type-safe.

## Static versus Dynamic Typing :-

A programming language is said to be statically typed when type checking is performed during compile time.

A programming language is said to be dynamically typed when majority of its type checking is performed at run-time.

In dynamically typed languages, values have types, but variables do not, that is a variable can refer to a value of any type.



Activation Records:-

Information needed by a single execution of a procedure is managed using a contiguous block of storage called activation record.

An activation record is allocated when a procedure is entered, and it is de-allocated when that procedure exits.

Storage Management:-(1) Static Storage Management:-

Static allocation is done during compilation that remains fixed throughout execution.

It requires no run-time storage management software.

Global data in C is allocated using static storage.

It is very efficient, but it is ~~incomplete~~ incompatible with recursive subprogram calls.

(2) Dynamic Storage Management:-

It is also termed as heap storage management.

The need of heap storage arises when a language permits storage to be allocated and freed at arbitrary points during program execution.

(3) Stack based Storage Management:-

It is used when storage requirements are not known at compile time, but the requests obey a Last In First Out order.

Examples:-

- (i) Local variables in a procedure in C.
- (ii) Procedure call information (return address etc.).

→ Stack-based allocation is used in recursive sub-programs.