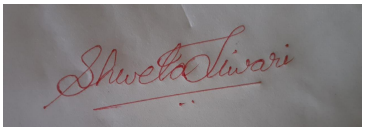
CD: UNIT-1
INTRODUCTION TO COMPILER DESIGN
FALL SEMESTER, YEAR (V, 3rd)

FALL SESSION (2022-23)
(CD)
MS. SHWETA TIWARI
Published: September 2, 2022
shwetatiwari08@recabn.ac.in
shwetatiwari08aug@gmail.com



UNIT-1

CD
COMPILER DESIGN

By SHWETA TIWARI

COMPILER DESIGN

UNIT I – INTRODUCTION TO COMPILER DESIGN

PREPARED FOR
Engineering Students
All Engineering College

PREPARED BY
SHWETA TIWARI
Guest Faculty

Language translator

A program written in a high-level language is called source code. To convert the source code into machine code, translators are needed.

A translator takes a program written in source language as input and converts it into a program in target language as output.

It also detects and reports the error during

translation**. Roles of translator are:**

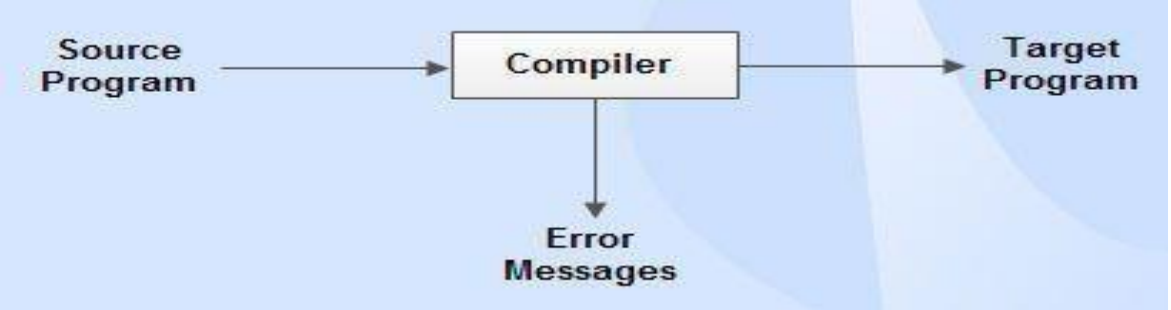
- Translating the high-level language program input into an equivalent machine language program.
- Providing diagnostic messages wherever the programmer violates specification of the high- level language program.

Different type of translators

The different types of translators are as follows:

Compiler

Compiler is a translator which is used to convert programs in high-level language to low-level language. It translates the entire program and also reports the errors in the source program encountered during the translation**.**



Interpreter

Interpreter is a translator which is used to convert programs in high-level language to low-level language. Interpreter translates line by line and reports the error once it encounters it during the translation process.

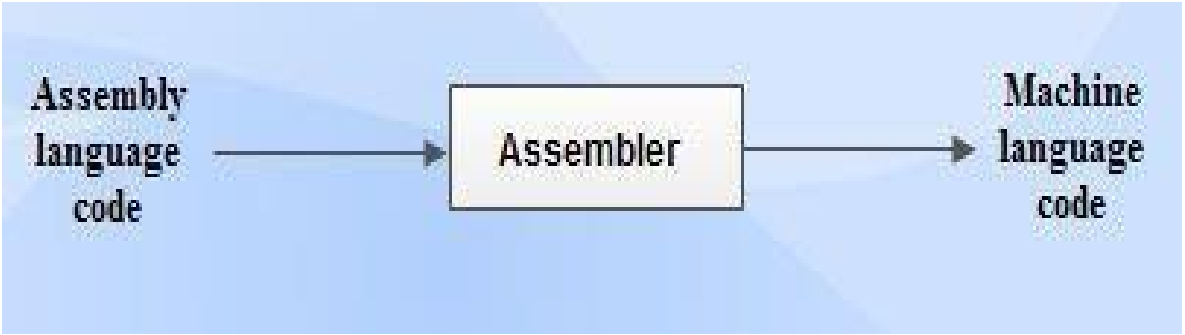
It directly executes the operations specified in the source program when the input is given by the user.

It gives better error diagnostics than a compiler.

SI. No	Compiler	Interpreter
1	Performs the translation of a program as a whole.	Performs statement by statement translation.
2	Execution is faster.	Execution is slower.
3	Requires more memory as linking is needed for the generated intermediate object code.	Memory usage is efficient as no intermediate object code is generated.
4	Debugging is hard as the error messages are generated after scanning the entire program only.	It stops translation when the first error is met. Hence, debugging is easy.
5	Programming languages like C, C++ use compilers.	Programming languages like Python, BASIC, and Ruby use interpreters.

Assembler

Assembler is a translator which is used to translate the assembly language code into machine language code



Language Processing System

We write programs in high-level language, which is easier for us to understand and remember. These programs are then fed into a series of tools and OS components to get the desired code that can be used by the machine. This is known as the Language Processing System.

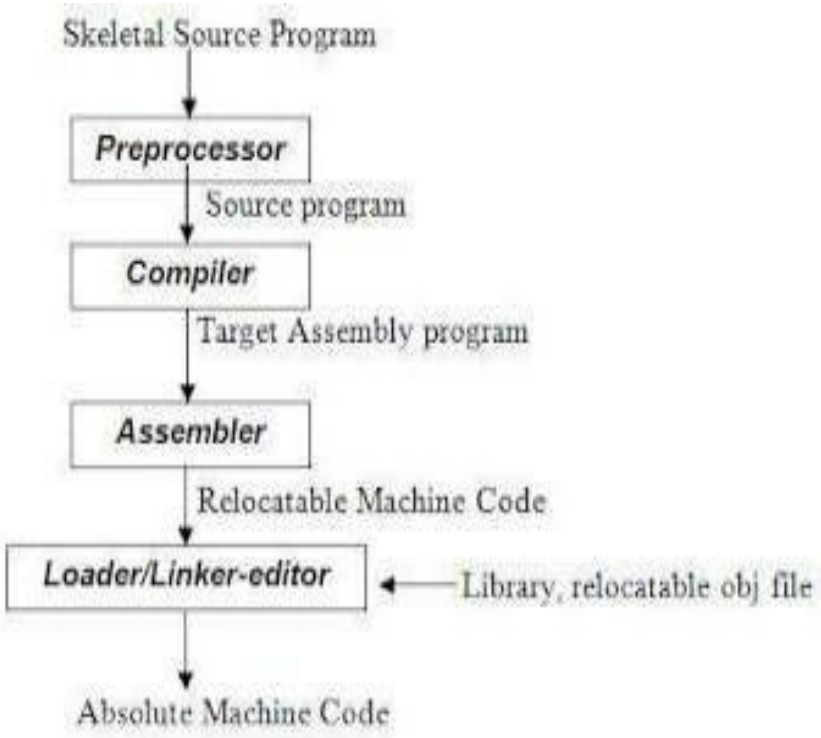


Fig 1.1 Language –processing System

Pre-processor

A source program may be divided into modules stored in separate files. The task of collecting the source program is entrusted to a separate program called pre-processor. It may also expand macros into source language statements.

Compiler

Compiler is a program that takes a source program as input and produces an assembly language program as output.

Assembler

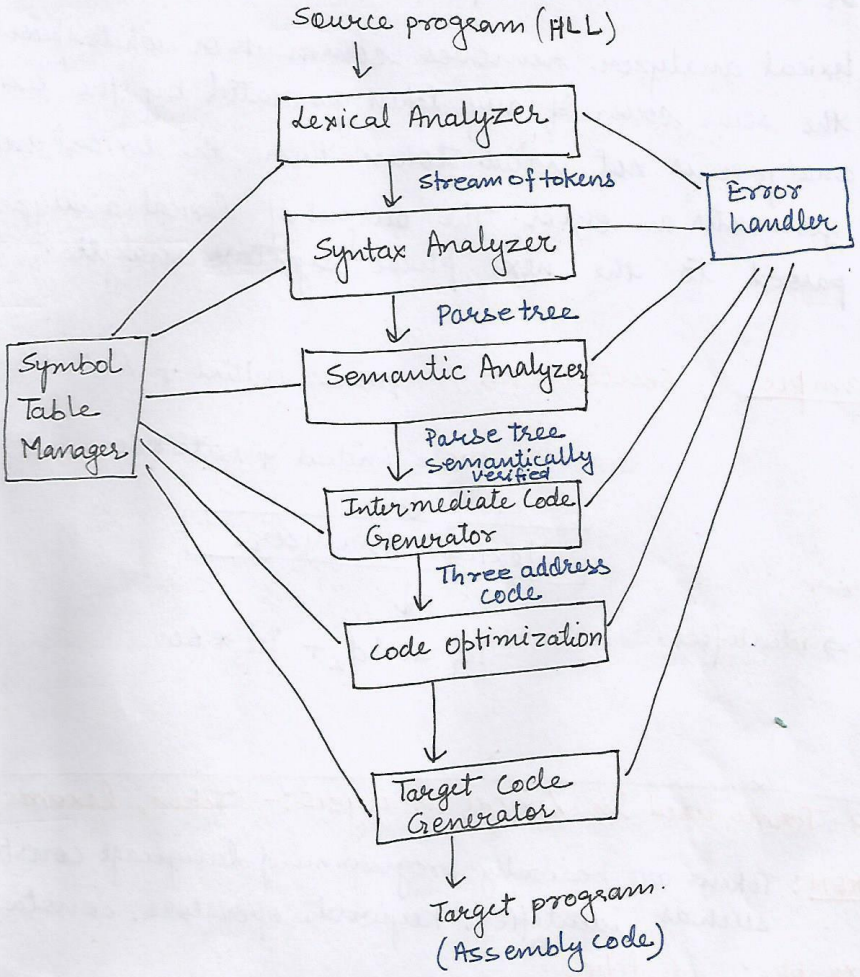
Assembler is a program that converts assembly language programs into machine language programs. It produces relocatable machine code as its output.

Loader and link-editor

- The relocatable machine code has to be linked together with other relocatable object files and library files into the code that actually runs on the machine.
- The linker resolves external memory addresses, where the code in one file may refer to a location in another file.
- The loader puts together the entire executable object files into memory for execution.

PHASES IN COMPILER:

→ Conceptually a compiler operates in phases, each of which transforms the source program from one representation to another.



Phases of compiler

① **LEXICAL ANALYSIS:** The lexical analysis phase reads the characters in the source program and groups them into a stream of tokens.

Token such as identifier, keyword, a punctual character or a multicharacter operator like <=.

lexical analyzer removes comments or white spaces from the souce code. If any token generated by the lexical analyzer is not valid token then the lexical analyzer generates an error. The output of lexical analyzer is passed to the next phase syntax analysis.

Example Sentence:⇒ pos = initial + rate * 60.

pos = initial + rate * 60

↓

Lexical Analyzer

↓

id ⇒ identifier. $id_1 = id_2 + id_3 * 60$

3- Terms used in lexical analysis:- Token, lexeme, pattern

TOKEN: Tokens are basically programming language constructs such as identifier, keyword, operators, constants etc.

Example: identifier.

LEXEMES: A lexeme is a sequence of characters in the source program that is matched by the pattern for a token.

Example:- pos, rate, initial.

PATTERN: A pattern is a rule describing the set of lexemes that can represent a particular token in source program.

Example:- rules to form identifier.

(2) SYNTAX ANALYZER:

Syntax analyzer takes a stream of tokens as an input and produces a parse tree as an output. Error generated by the syntax analyzer is known as syntax error or syntactic error.

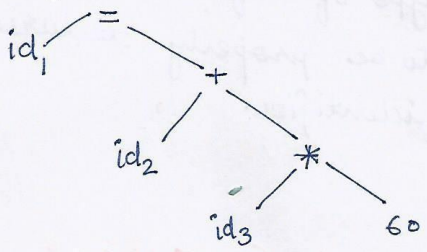
Example:

$id_1 = id_2 + id_3 * 60$

↓

Syntax analyzer

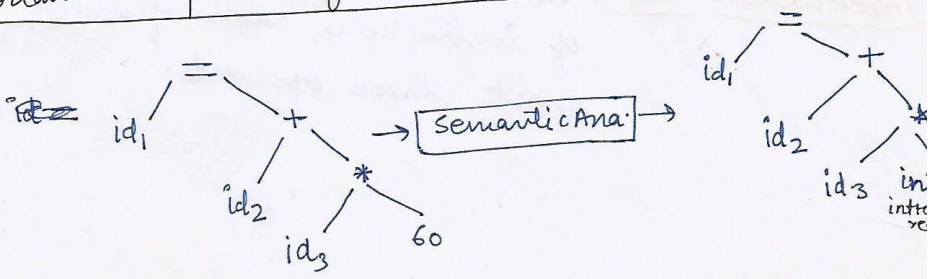
↓



SEMANTIC ANALYZER: Semantic analysis phase checks the source program for semantic error and gathers type information for the subsequent code-generation phase.

Semantic analyzer uses hierarchical structure determined by the syntax analysis phase and produces semantically verified parse tree (hierarchical structure).

An important component of semantic analysis is type checki



Intermediate code Generation:

Some compilers generate an explicit intermediate representation of the source program.
This intermediate representation should have two important properties:

(i) it should be easy to produce.
(ii) it should be easy to translate into target code.

Intermediate code Generator

$t_1 = \text{inttoreal}(60)$
$t_2 = id_3 * t_1$
$t_3 = id_2 + t_2$
$id_1 = t_3$

CODE OPTIMIZER: code optimization phase attempts to improve the intermediate code.

> Optimization means "Transforming a piece of code to make more efficient (either in terms of time or space)" without changing its output or side effects.

Example:-

$t_1 = \text{inttoreal}(60)$
$t_2 = id_3 * t_1$
$t_3 = id_2 + t_2$
$id_1 = t_3$

→ [code optimizer] →

$t_1 = id_3 * 6.0$
$id_1 = id_2 + t_1$

CODE GENERATION: The final phase of compiler is the generation of target code.

Memory locations are selected for each of the variables used by the program. Then, intermediate instructions are each translated into a sequence of machine instructions that perform the same task.

$t_1 = id_3 * 60.0$

$id_1 = id_2 + t_1$

↓

code generator

↓

```
MOVF  id3 , R2
MULF  #60.0 , R2
MOVF  id2 , R1
ADDF  R2 , R1
MOVF  R1 , id1
```

SYMBOL TABLE: A symbol table is a data structure containing a record for each identifier, with fields for the attributes of the identifier. The data structure allows us to find the ~~retrieve~~ record for each identifier quickly and to store or retrieve data from that record quickly.

attributes of Identifier:- such as storage allocated for an identifier, its type, its scope, in case of procedure names, such things as the number and types of its arguments, the method of passing each argument (e.g. by reference) and the type returned, if any.

ERROR DETECTION AND REPORTING:

Each phase can encounter errors. After detecting an error, a phase must deal with that error, so that compilation can proceed to find further errors.

> The syntax and semantic analysis phases usually handle a large fraction of the errors detectable by compiler

The lexical analysis phase detect errors where the characters remaining in the input do not form any token of the langua

The syntax analysis phase detect error where the ~~syntactic~~ the token streams violates the structure rule.

The semantic analysis phase detect errors where syntactic structure does not produce any meaning to the operation invoked.

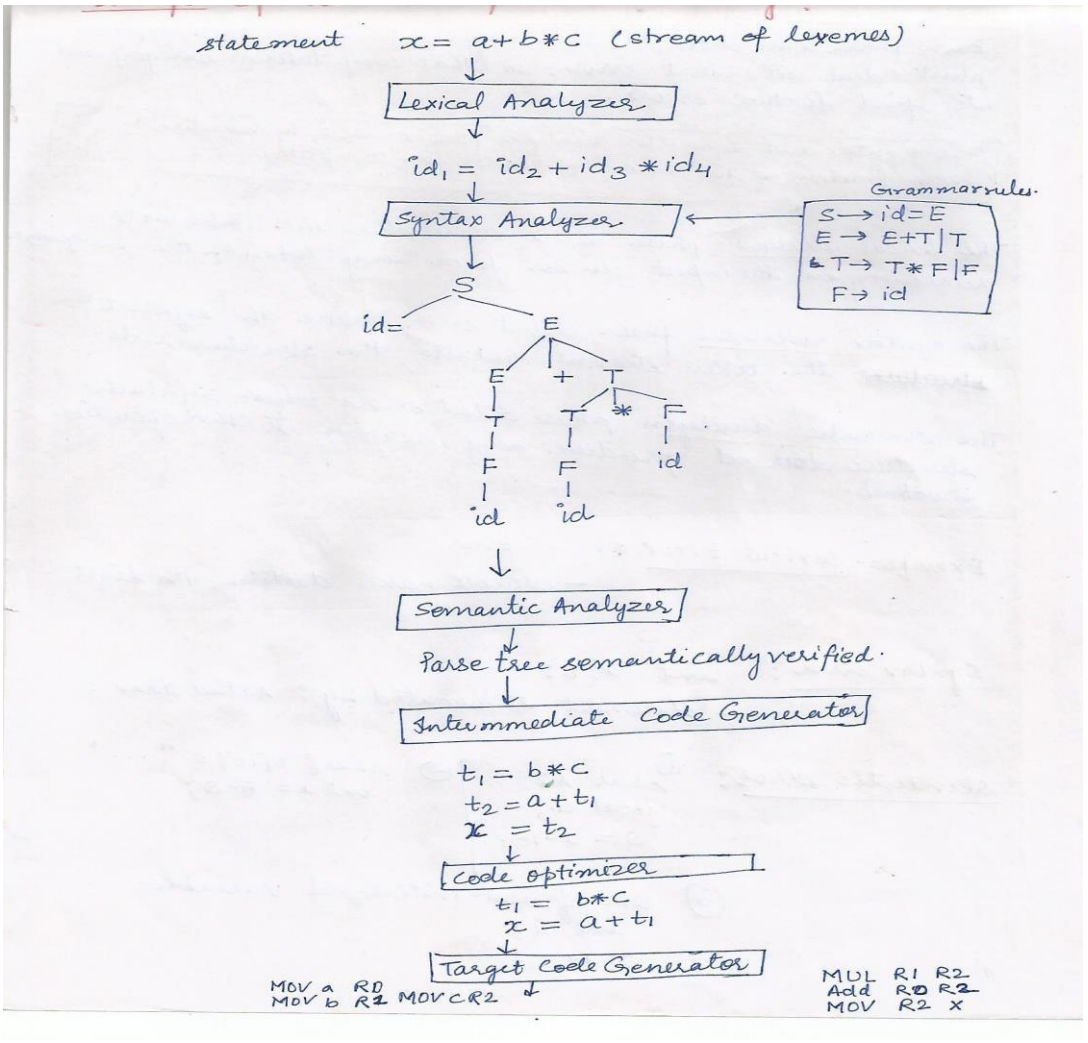
Example- Lexical Error : 2var
- variable name starting with digit

Syntax error : int a, b:
- statement terminated by : rather than ;

Semantic error: ① char s;
float a;
s = a-10;

② char s[10] = "..."
int a = 5-s;

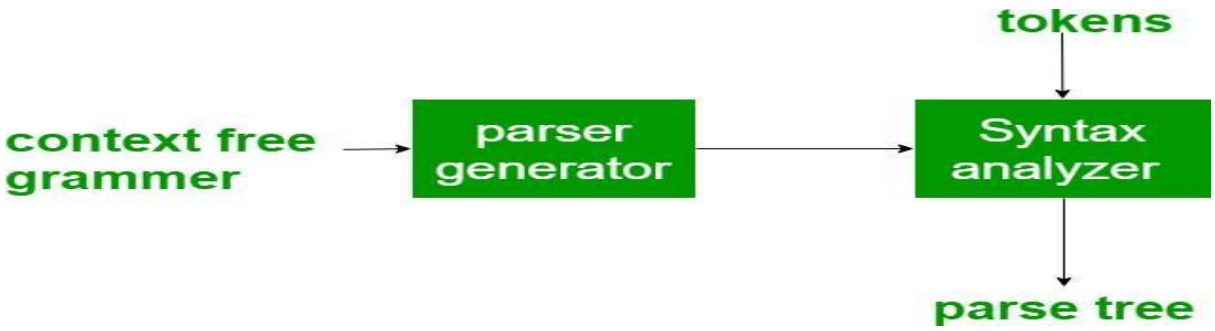
③ use of non-initialized variable.
int i;
i++;



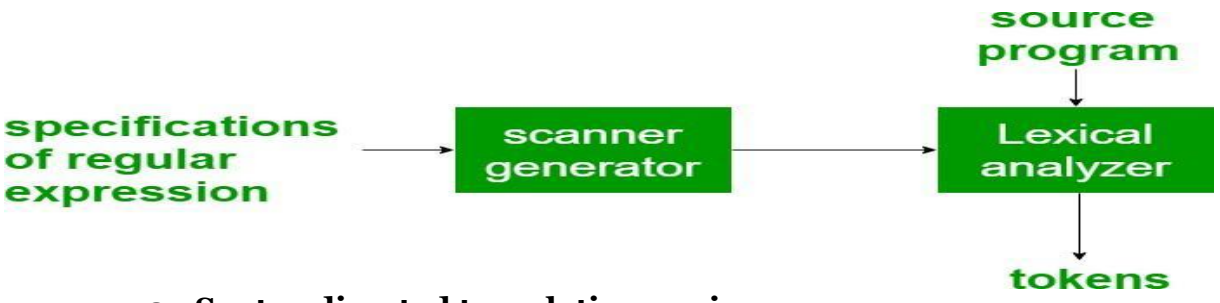
Compiler construction tools

The compiler writer can use some specialized tools that help in implementing various phases of a compiler. These tools assist in the creation of an entire compiler or its parts. Some commonly used compiler construction tools include:

1. **Parser Generator –**
   It produces syntax analyzers (parsers) from the input that is based on a grammatical description of a programming language or on a context-free grammar. It is useful as the syntax analysis phase is highly complex and consumes more manual and compilation time.
   Example: PIC, EQM

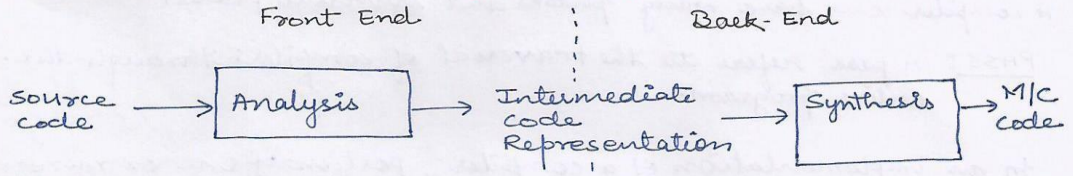


2. **Scanner Generator –**
   It generates lexical analyzers from the input that consists of regular expression description based on tokens of a language. It generates a finite automaton to recognize the regular expression.
   Example: Lex



3. **Syntax directed translation engines –**
   It generates intermediate code with three address formats from the input that consists of a parse tree. These engines have routines to traverse the parse tree and then produce the intermediate code. In this, each node of the parse tree is associated with one or more translations.
4. **Automatic code generators –**
   It generates the machine language for a target machine. Each operation of the intermediate language is translated using a collection of rules and then is taken as an input by the code generator. Template matching process is used. An intermediate language statement is replaced by its equivalent machine language statement using templates.
5. **Data-flow analysis engines –**
   It is used in code optimization. Data flow analysis is a key part of the code optimization that gathers the information, that is the values that flow from one part of a program to another.
6. **Compiler construction toolkits –**
   It provides an integrated set of routines that aids in building compiler components or in the construction of various phases of compiler.

COMPILER ARCHITECTURE (Analysis & Synthesis model)



Analysis: This part is used to divide the source program into pieces and then create an intermediate representation of that source program.

Synthesis: This part is used to construct the desired target program from intermediate representation created in analysis part.

FRONT END:→ consists of:

- Lexical Analysis phase
- Syntax Analysis phase
- Semantic Analysis phase
- Intermediate Code generation

BACKEND :→

→ code optimization
→ Target code generation

PHASES AND PASSES:

A compiler can have many phases and passes.

PASS: A pass refers to the traversal of compiler through the entire program.

In an implementation of a compiler, portions of one or more phases are combined into a module called a pass.

A pass reads the source program or the output of the previous pass, makes the transformations specified by its phases and writes output into an intermediate file, which may be then read by a subsequent pass.

The number of passes, and grouping of phases into passes usually depends on two factors
- → structure of source language.
- → The environment in which compiler operates (Platform).

Single Pass Compiler: (or Narrow Compiler)

- → A pass compiler is a compiler that passes through the source code of each compilation unit only once.
- → The whole compiler is kept in main memory together so it takes more space.
- → It is is faster than multipass compiler.
- → A one-pass compiler does not 'look-back' at the code it previously processed.

Example:— Pascal's compiler.

Multipass compiler: (Wider compiler).

- A multipass compiler processes the source code of a program several times.
- Each pass takes the result of previous pass as the input and creates an intermediate output. So it is slower than a single pass compiler.
- A multipass compiler uses less space, since the space occupied by one pass can be reused by the following pass.
- The wider scope available to these compiler allows better code generation. The intermediate code is improved pass by pass.

Example:- C++ compiler

Some language can not be compiled in a single pass as a result of design (their structure).
All programming language can be represented with a multipass compiler.

SINGLE PASS COMPILER
VERSUS
MULTIPASS COMPILER

SINGLE PASS COMPILER	MULTIPASS COMPILER
A type of compiler that passes through the parts of each compilation unit only once, immediately translating each code section into its final machine code	A type of compiler that processes the source code or abstract syntax tree of a program several times
Faster than multipass compiler	Slower as each pass reads and writes an intermediate file
Called a narrow compiler	Called a wide compiler
Has a limited scope	Has a great scope
There is no code optimization	There is code optimization
There is no intermediate code generation	There is intermediate code generation
Takes a minimum time to compile	Takes some time to compile
Memory consumption is lower	Memory consumption is higher
Used to implement programming languages such as Pascal	Used to implement programming languages such as Java

BOOTSTRAPPING :

Bootstrapping is an important concept for building new compilers.

A compiler can be characterized by three languages:

(i) the source language (S) that it compiles
(ii) the target language (T) that it generates code for.
(iii) the implementation language (I) that it is written in.

> Notation S^{T}_{I} represents a compiler for source S, target T, implemented in I.

CROSS COMPILER :

Cross compiler is a compiler that runs on one machine and produces machine code for other machine.

Bootstrap Compiler : if compiler is implemented using its own language, then the compiler is known as bootstrap compiler.

Implementing a bootstrap compiler :

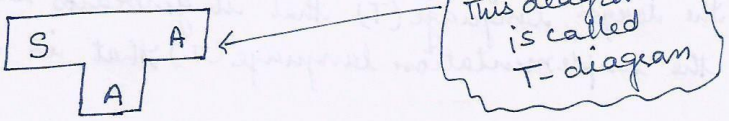
> * Bootstrapping is the technique for producing a "self-compiling compiler" that is, compiler written in the source language that it intends to compile.

Native compiler are written in the same language as the target language e.g.

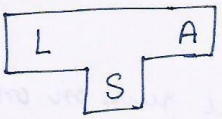
SMM is a compiler for language S that runs on machine M and generates output code that runs on machine M.

Steps involving to create a new Language L for machine A or Native compiler:

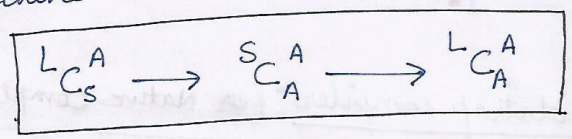
create ${}^{S}C_{A}^{A}$, a compiler for subset S of the desired language. L, using language A which runs on machine A.

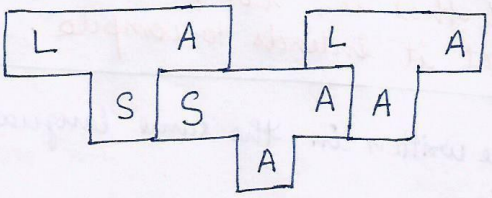


create ${}^{L}C_{S}^{A}$, a compiler for language L written in subset of L.



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





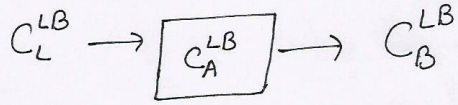
$$L_S^A + S_A^A = L_A^A$$

CROSS COMPILER:

cross compiler are written in different languages as the target language e.g. SNM is a compiler for the language S that is in a language that runs on machine N and generates output code that runs on machine M.

Steps involving to produce a compiler for a different machine B

1. Convert L_S^A into L_L^B. Recall that language S is a subset of language L.
2. Compile L_L^B to produce L_A^B, a cross compiler for L which runs on machine A and produces code for machine B.
3. Compile L_L^B with the cross compiler to produce L_B^B, a compiler for language L which runs on machine B.



Bootstrapping a compiler to a second machine.

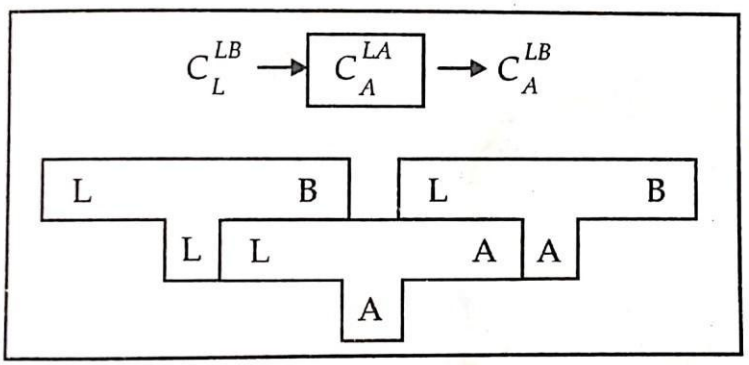
C_A^{LA} → is a native compiler

C_L^{LB} → is a bootstrap compiler

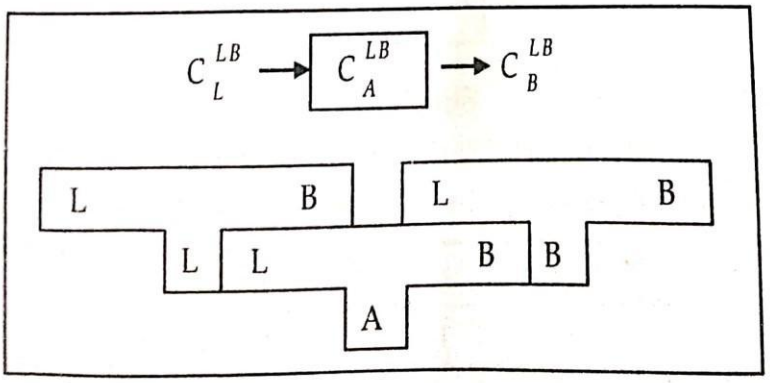
C_A^{LB} → is a cross compiler

Example 1.9. **Consider the following, see how the cross compilers are created using bootstrapping,**

Solution.



(*a*)



(*b*)

Example 1.10. Create a cross compiler for EQN using C compiler, written in PDP-11, producing code in PDP-11 and a EQN language producing code for text formatter, TROFF written in *C*.

Solution. We have

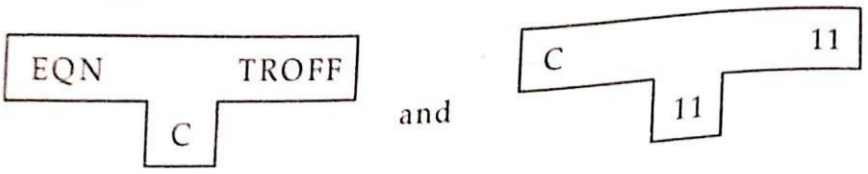


Fig. 1.35.

where, 11 signifies PDP-11, we can create cross compiler for EQN using T-diagram as follows :

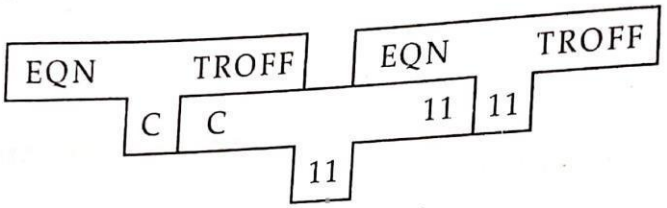


Fig. 1.36.

e. we have created

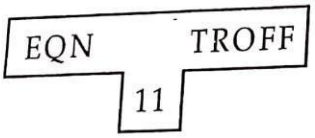


Fig. 1.37.

This can be written in formula as,

$$C_{C}^{EQN\ TROFF} \longrightarrow \boxed{C_{11}^{C\ 11}} \longrightarrow C_{11}^{EQN\ TROFF}$$

Fig. 1.38.

Finite machines and regular expressions and their applications to lexical analysis

FINITE STATE MACHINE: A finite automata is an abstract machine that works as a recognizer for the string that comprises a regular language.

The idea is that we can feed an input string into a FA and it will answer "yes" or "no" depending on whether or not the input string belongs to the language that the automaton recognizes.

A finite automaton, or finite-state machine is a 5-tuple $(Q, \Sigma, q_0, A, \delta)$ where

$Q \Rightarrow$ a finite non-empty set of states

$\Sigma \Rightarrow$ a finite non-empty set of input symbols

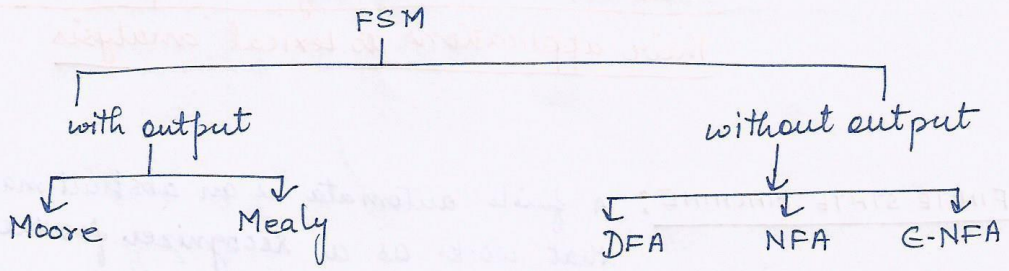
$q_0 \Rightarrow$ $q_0 \in Q$ (the initial state)

A or F $\Rightarrow$ $A \subseteq Q$ (the set of accepting states or final states)

$\delta \Rightarrow$ a function from $Q \times \Sigma \rightarrow Q$ (the transition function)

For any element q of Q and any symbol $a \in \Sigma$, we interpret $\delta(q, a)$ as the state to which the FA moves, if it is in state q and receives the input a.

TYPES OF FSM :→



Deterministic Finite Automata :

In DFA, for each state there is unique transition on each symbo
Each DFA is represented by 5-tuples. $(Q, \Sigma, q_0, F, \delta)$ where

$Q \rightarrow$ a finite non empty set of states.
$\Sigma \rightarrow$ a finite non-empty set of input symbols.
$q_0 \rightarrow$ starting state, $q_0 \in Q$.
$F \rightarrow$ a set of final state
$\delta \rightarrow$ a transition function.

$$Q \times \Sigma \rightarrow Q.$$

Non-Deterministic Finite Automata :

NFA allows zero, one, or more transitions from a state on the same input symbol.
Formally we denote NFA by a 5-tuple. $(Q, \Sigma, \delta, q_0, F)$ where

$Q \rightarrow$ non empty finite set of states
$\Sigma \rightarrow$ non-empty finite set of input symbols.
$q_0 \rightarrow$ starting state
$F \rightarrow$ set of final states
$\delta \rightarrow$ a transition function which maps $Q \times \Sigma$ to 2^Q.

$$Q \times \Sigma \longrightarrow 2^Q \text{ (power set of } Q)$$

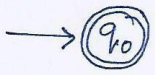
power set of Q means the set of all subsets of Q.

ϵ-NFA or NFA with Epsilon:

Formally, we define a NFA with epsilon to be a quintuple $(Q, \Sigma, q_0, \delta, F)$ with all components as before but δ, the transition function, maps $Q \times (\Sigma \cup \{\epsilon\}) \rightarrow 2^Q$.

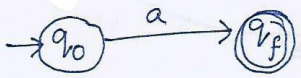
ϵ NFA from regular expression r:-

(i) $r = \epsilon$

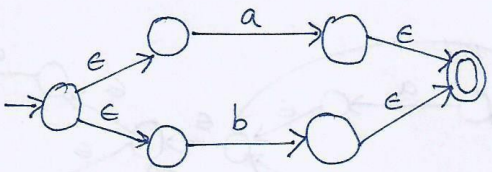


(ii) $r = \phi$

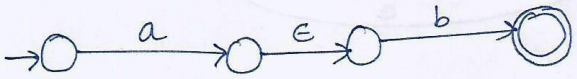
(iii) $r = a$



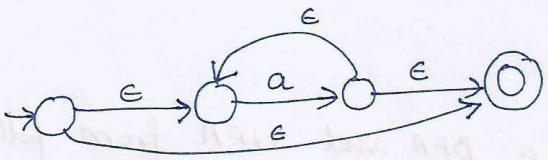
(iv) $r = (a+b)$



(v) $r = ab$



(vi) $r = a^*$



Subset Construction algorithm

Thompson's Subset Construction Algorithm

1. A = ∈-closure(0), i.e. start state.
2. For each input symbol a ∈ ∑ do.

Let T be set of states to which there is a transition on a from states in A.

// look for the states in A on which there is a transition on input symbol, say a, b

// then T_a = { set of states to which transition is made on a}.

// T_b = {set of states to which transition is made on b}

B = ∈-closure (T_a)

C = ∈-closure (T_b)

Mark A as its transition state has been found and store in set D′

3. Repeat the step 2 with newly found states i.e. B and C and this process will be repeated till all the newly found states are marked and there is atleast one state which contains final state of NFA.

Example 3.11. Find the DFA for the following regular expression

$$(a \mid b)^* abb.$$

Solution. Let us first construct an NFA, then we can apply the subset construction algorithm.

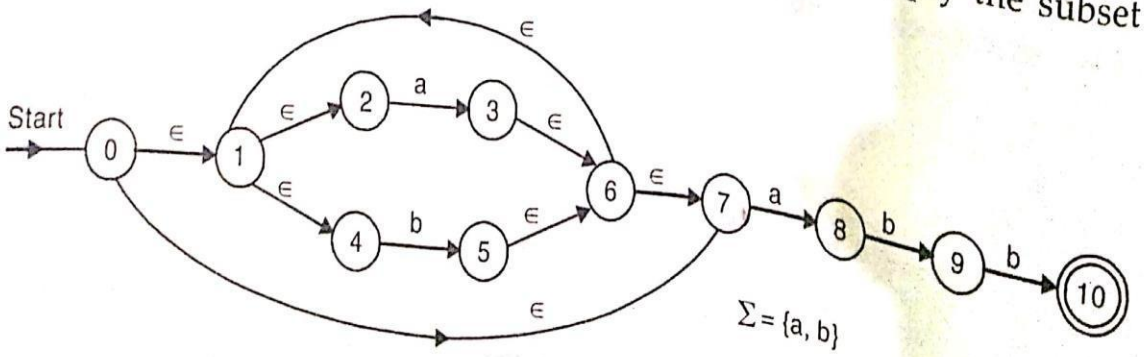


Fig. 3.15.

Now applying the algorithm

$A = \epsilon$–closure(0)
$= \{0, 1, 2, 4, 7\}$

Here, there is transition on input symbol a, *b* on 2, 4, 7 to 3, 5, 8

(2) $\xrightarrow{a}$ (3)

(4) $\xrightarrow{b}$ (5)

(7) $\xrightarrow{a}$ (8)

Thus, $T_a = \{3, 8\}$
$T_b = \{5\}$

Hence, $B = \epsilon$–closure (T_a)
$C = \epsilon$–closure (T_b)

Thus, we have found transition for *A i.e.* to *B* on *a*, and to *C* on *b*.

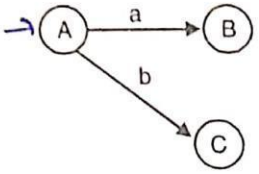


Fig. 3.16.

Now, repeat the process for *B*.

$B = \epsilon$–closure $(T_a) = \epsilon$–closure $\{3, 8\}$
$= \epsilon$–closure $(3) \cup \epsilon$–closure $\{8\}$
$= \{1, 2, 3, 4, 6, 7\} \cup \{8\}$
$= \{1, 2, 3, 4, 6, 7, 8\}$
$C = \epsilon$-closure $(T_b) = \epsilon$-closure $\{5\}$
$= \{1, 2, 4, 5, 6, 7\}$

Let *D* be the set of marked states. Thus add *A* to *D i.e.* $D = \{A\}$

For *B*, $B = \{1, \underline{2}, \underline{3}, \underline{4}, 6, \underline{7}, 8\}$ → (3, 8)
$T_a = \{3, 8\}$
$T_b = \{5, 9\}$
ϵ-closure $(T_a) = \epsilon$-closure $\{3, 8\} = B$
ϵ-closure $(T_b) = \epsilon$-closure $\{5, 9\}$
$= \{1, 2, 4, 5, 6, 7, 9\} = D$

Thus, we have found transitions for *B*

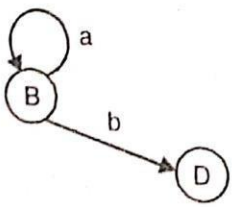


Fig. 3.17.

Hence, $D = \{A, B, \ldots\}$ *i.e.* *B* is marked.

For *C*, $C = \{1, \underline{2}, \underline{4}, 5, 6, \underline{7}\}$
$T_a = \{3,8\}$ from states 2,7 on *a*.

Thus, $\in$-closure $(T_a) = \in$-closure $\{3, 8\}$

$= B$

$T_b = \{5\}$ from state 4 on b.

$\in$-closure $(T_b) = \in$-closure$\{5\}$

$= C$

Thus, transitions from C on input a, b are

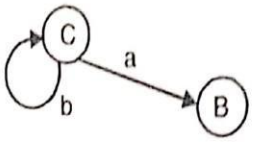


Fig. 3.18.

Thus C is marked.

Hence,

For *D*,

$D' = \{A, B, C, ...\}$

$D = \{1, 2, 4, 5, 6, 7, 9\}$

$T_a = \{3, 8\}$

$T_b = \{5, 10\}$

$\in$-closure$(T_a) = B$

$\in$-closure$(T_b) = \{1, 2, 4, 5, 6, 7, 10\}$

$= E$

Thus, we have found transitions for D.

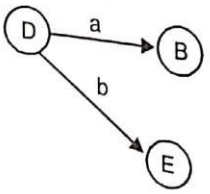


Fig. 3.19.

Hence, D is included in marked state, *i.e.* $D' = \{A, B, C, D, ...\}$

For *E*,

$E = \{1, 2, 4, 5, 6, 7, 10\}$

$T_a = \{3, 8\}$

$T_b = \{5\}$

$\in$-closure$(T_a) = B$

$\in$-closure$(T_b) = C$

Thus,

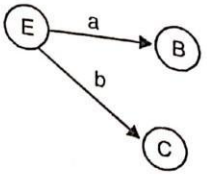


Fig. 3.20.

Hence, E is marked

$D' = \{A, B, C, D, E\}$

Thus, there is no new state and all the states are marked and also E is final state, as it contains the final state 10 of NFA.

Thus, final DFA is obtained y combining all the states of DFA.

Transition Table is given as

δ	a	b
$\rightarrow A$	B	C
B	B	D
C	B	C
D	B	E
E	B	C

$\Sigma = \{a, b\}$

$Q = \{A, B, C, D, E\}$

$q_0 = A$

$q_f = E$

$\delta \leftarrow$ Shown above

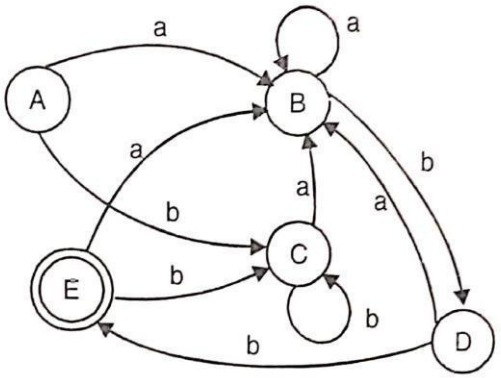


Fig. 3.21. DFA obtained from regular expression and NFA.

REGULAR EXPRESSIONS: The language accepted by finite automata are easily described by simple expressions called regular expressions.

Regular expressions has 3 operations

(i) + (union)
(ii) . concatenation
(iii) * Kleene closure

Let Σ be an alphabet. The regular expressions over Σ and the sets they denote are defined recursively as follows:

1. ϕ is a regular expression and denote the empty set ϕ.
2. ϵ is a regular expression and denote the set $\{\epsilon\}$
3. for each a in Σ, a is a regular expression and denote the set $\{a\}$.
4. if r and s are regular expressions denoting the language R and S, respectively then
   - $(r+s)$ regular expression denotes the set $R \cup S$
   - (rs) regular expression denotes the set RS
   - r^* regular expression denotes the set R^*

Question: write down the regular expression for the language that accepts all strings of length 2 over $\{a, b\}$

Ans $L = \{aa, ab, ba, bb\}$

step 1 In this case first step is to write language
$L = \{aa, ab, ba, bb\}$

step 2 Then apply union on all strings i.e

$$aa + ab + ba + bb$$

since the language is finite, hence regular expression will be the union of all strings.

$$a(a+b) + b(a+b)$$

$$\boxed{(a+b)(a+b)}$$ ← regular expression for all strings of length 2 over $\Sigma = \{a, b\}$.

Similarly RE for strings of length 3 over $\Sigma = \{a, b\}$

$$(a+b)(a+b)(a+b)$$

Question: How many number of strings of length n can be formed over an alphabet $\Sigma = \{a, b\}$

Ans: length of $\Sigma = |\Sigma| = 2$

To fill n places we have two options

— — — — — — — - - - - - upto n

$\{a,b\}$ $\{a,b\}$ $\{a,b\}$ $\{a,b\}_n$

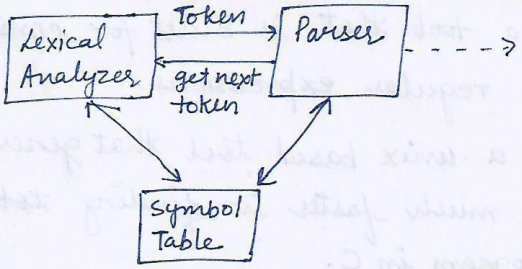
So number of strings of length $n = 2^n$

or we can write it as

$$\boxed{\text{No. of string of length } n \text{ over an } \Sigma = |\Sigma|^n}$$

Implementation of LEXICAL ANALYZERS:

The lexical analyzer is the first phase of compiler. Its main task is to read the input characters and produce as output a sequence of tokens that the parser uses for syntax analysis.



Interaction of lexical analyzer with parser.

RESPONSIBILITIES OF LEXICAL ANALYZER: Lexical analyzer is the only phase which reads input character by character. It performs following tasks too.

1. Reading lexemes and converting them into tokens
2. Removing comments
3. Removing white spaces
4. Showing errors.

Ques: Find number of tokens in the following program.

```
int max(x,y)
int x,y;
/* find max of x and y */
{
return (x > y ? x : y);
}
```

Ans: no. of tokens ⇒ 25

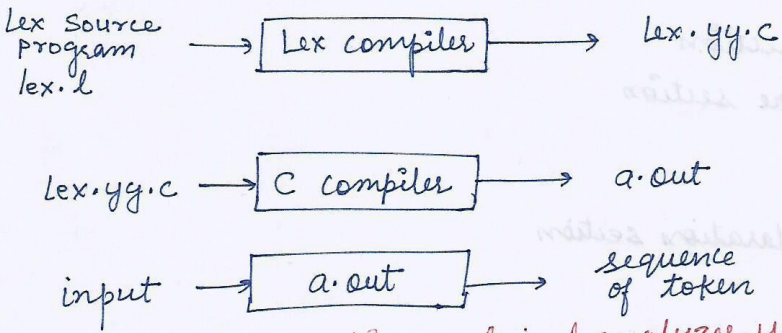
For lexical analyzer there is a tool called "Lex"
For syntax analyzer there is a tool called "YACC"

LEXICAL ANALYZER GENERATOR:

- Lex is a tool that is built for constructing lexical analyzer using regular expressions.
- Lex is a unix based tool that generates lexical analyzer.
- Lex is much faster in finding tokens than handwritten LEX program in C.

LEX:

The Lex compiler is a tool that allows one to specify a lexica analyzer from regular expressions.
Inputs are specified in 'Lex language'.



creating a lexical analyzer using Lex.

① Lex specification file is created with the name lex.l or extension.l or x.l in lex language.

② Then lex.l is run through lex compiler to produce a C pro which is actually a lexical analyzer program.

③ specification file lex.l consists of regular expressions for th token

④ The program lex.yy.c consists of a tabular represent -ion of a transition diagram constructed from the regular expressions of lex.l.

→ In specification file of lex actions are associated with each regular expression.

→ These actions are simply pieces of C codes. These C codes are directly carried over to the lex.yy.C.

→ Finally lex.yy.c is run through the C compiler to produce an object program a.out

→ When some input stream is given to a.out then sequence of tokens get generated.

Question How do we write the specification rules in lex program.

LEX specification:

A lex program consists of three parts

1. Declaration section
2. Rule section
3. Procedure section

```
%{
    Declaration section
%}
%%
    Rule section
%%
    Auxiliary procedure section
```

→ In declaration section, declaration of variable, constant can be done and some regular definitions (statements used as components of regular expressions appearing in the translation rules) can also be written.

→ Rule section consists of regular expressions with associated actions like

```
R1      { action 1 }
R2      { action 2 }
 :
 :
Rn      { action n }
```

→ In procedure section, all the required procedures are defined. sometimes these procedures are required by the actions in rule section.

After activating by the parser, lexical analyzer starts reading its remaining input character by character at a time.
When string is matched with one of the regular expressions R_i then corresponding action i will get executed.

Commands to seen the Lex program say name.l

```
$ lex name.l
$ gcc lex.yy.c
$ ./a.out
```

CONTEXT FREE GRAMMAR:

A context free grammar is denoted by G 4 tuple $G = (V, \Sigma, P, S)$ where

V is a finite set of nonterminals or variables

Σ is a finite set of terminals

S is a start symbol and an element of set V.

P is a finite set of grammar rules (or production rules) of the form

$$\boxed{A \longrightarrow \alpha, \text{ where } A \in V, \alpha \in (V \cup \Sigma)^*}$$

Example:

$$G = (\{E\}, \{+, *, (,), id\}, P, E)$$

$$P = \{ E \to E+E,\ E \to E*E,\ E \to (E),\ E \to id \}$$

LANGUAGE GENERATED BY CFG:

A language L is a context-free language (CFL) if there is a CFG G so that $\boxed{L = L(G)}$

Let $G = (V, \Sigma, P, S)$ be a CFG. The language generated by G is

$$L(G) = \{ x \in \Sigma^* \mid S \overset{*}{\Rightarrow}_G x \}$$

Derivation:

① Left Most Derivation

② Right most derivation

Grammar $E \to E+E \mid E*E \mid id$

string $\Rightarrow$ id+id*id. Derivation of the given string using left most and right most derivations

LMD

$E \rightarrow E+E$
$\rightarrow id+E$
$\rightarrow id+E*E$
$\rightarrow id+id*E$
$\rightarrow id+id*id$

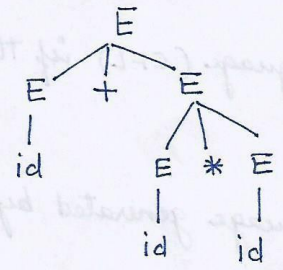
RMD

$E \rightarrow E+E$
$\rightarrow E+E*E$
$\rightarrow E+E*id$
$\rightarrow E+id*id$
$\rightarrow id+id*id$

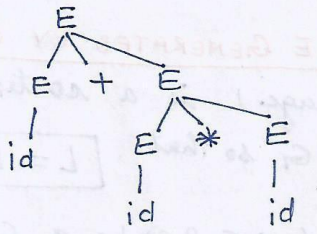
PARSE TREE: A Derivation tree or parse tree is an ordered rooted tree that graphically represents a string derived from a CFG.

Root vertex: must be labelled by the start symbol
Vertex: labelled by non-terminal
Leaves: labelled by terminal or ϵ

Example



Parse tree for LMD



Parse tree for RMD.

Ambiguity:

A grammar G is said to be ambiguous if it generates more than one left most derivation or right most derivation.

Example: $E \rightarrow E+E \mid E*E \mid id$

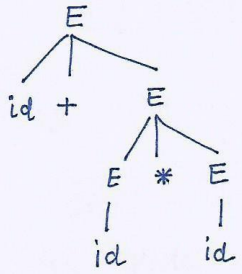
This given grammar is ambiguous because we have two left most derivation for string id+id*id

(1) $E \rightarrow E+E$
$\rightarrow id+E$
$\rightarrow id+E*E$
$\rightarrow id+id*E$
$\rightarrow id+id*id$

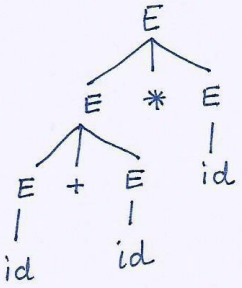
(2) $E \rightarrow E*E$
$\rightarrow E+E*E$
$\rightarrow id+E*E$
$\rightarrow id+id*E$
$\rightarrow id+id*id$

For parsers, it is desirable that the grammar be made unambiguous, for if it is not, it can not uniquely determine which parse tree to select for sentence.

Parse tree ①



Parse tree ②



Example string: 2+3*4

Acc. to parse tree ①

$E \rightarrow id + (E * E)$
$id + (id * E)$
$id + (id * id)$
$2 + (3 * 4)$
$\Rightarrow 14$

According to parse tree ②

$E \rightarrow E * E$
$(E+E) * E$
$(id+E) * E$
$(id+id) * E$
$(id+id) * id$
$(2+3) * 4$
$= 20$

In both cases answer is different. Hence no parser allows ambiguous grammar.

Example of ambiguous grammar.

① $S \rightarrow aS \mid Sa \mid a$

② $S \rightarrow aSbS \mid bSaS \mid \epsilon$

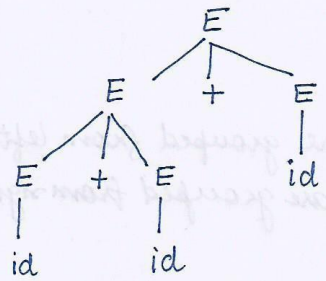
③ $R \rightarrow R+R \mid RR \mid R^* \mid a \mid b \mid c$

some of the ambiguous grammars can be converted to unambiguous grammars.

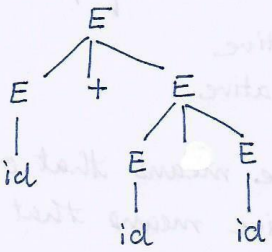
Example

$$E \rightarrow E+E \mid E*E \mid id$$

string: id + id + id.

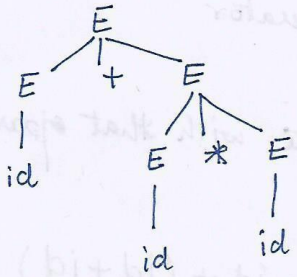


Parse tree ①

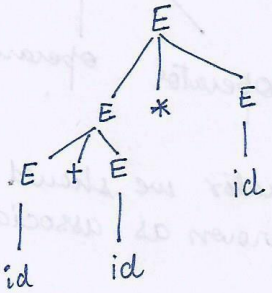


Parse tree ②

string id + id * id



Parse tree ①



Parse tree ②

BNF Notation : (BACKUS NAUR FORM):

BNF is a meta language (a language describing the other languag

The reason for using meta language is "it allows to:

- → Determine whether a series of characters is valid.
- → Generate well-formed statements
- → Break down a statement into constituent parts so it can be converted into machine code

BNF is a formal notation to describe the syntax of a given languag
BNF is just a variant of context free grammar (CFG), with the symbol "::=" used in place of '→' to mean "is defined as" or "is defined by"

EXAMPLE:

<digit> ::= 0|1|2|3|4|5|6|7|8|9

In this particular rule <digit> is non-terminal
0,1,2, --- 9 are terminals.

<integer>::= <digit> | <digit><integer>
<word> ::= <letter> | <letter><word>
<letter> ::= a|b|c|d|e|f|--- |z
<arithmetic op>::= +|-

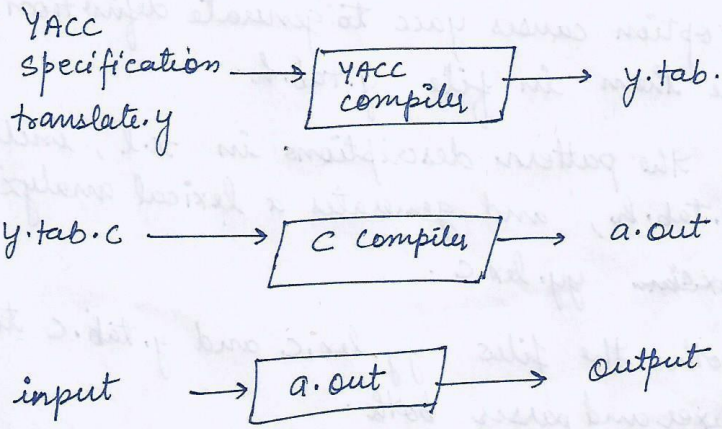
SO BNF is a formal, mathematical way to specify context-free grammar.
BNF is a meta language
BNF is precise and unambiguous.
BNF grammar uses terminals and non-terminals

PARSER GENERATOR

YACC : (Yet Another Compiler Compiler)

- YACC is the standard parser generator for unix based system.
- YACC generates code for parser in C programming language.
- YACC is useful to build a bottom up (shift reduce) (LALR) parser from a context free grammar.

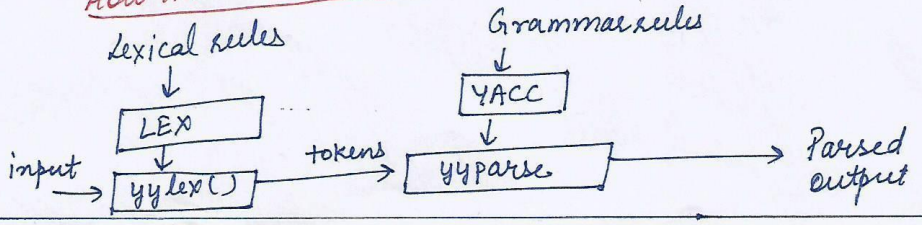
creating an input/output translator with YACC.

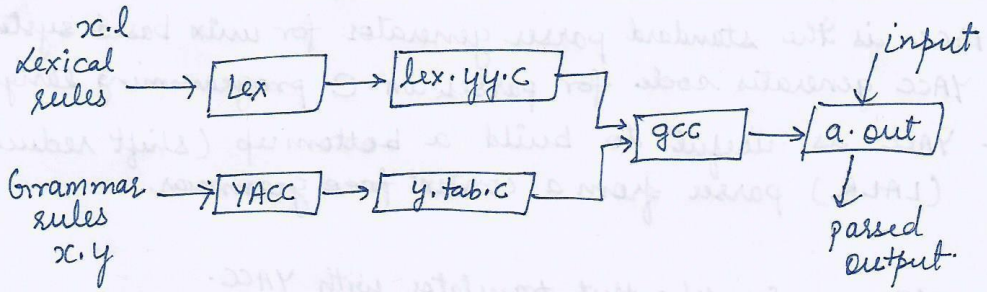


Structure of YACC source program:

```
%{
    declarations        (optional)
%}
%%
%%
    Rules               (compulsory)
%%
```

How YACC works with LEX :→





The -d option causes yacc to generate definition for tokens and place them in file y.tab.h.

Lex reads the pattern descriptions in x.l, includes files y.tab.h, and generates a lexical analyzer as in ~~yy.lex.yy~~ yy.lex.c.

compile both the files yy.lex.c and y.tab.c to create lexer and parser both.

CAPABILITIES OF CFG:

context free grammar are capable of describing most, but not all, of the syntax of programming languages.
This section indicates what programming constructs can and, can not be described by context free grammars.

→ Regular expressions are capable of describing the syntax of tokens. Any syntactic construct that can be described by a regular expression can also be described by a context free grammar.

regular expression $(a+b)(a+b+0+1)^*$

context-free grammar $S \rightarrow aA \mid bA$

$A \rightarrow aA \mid bA \mid 0A \mid 1A \mid \in$

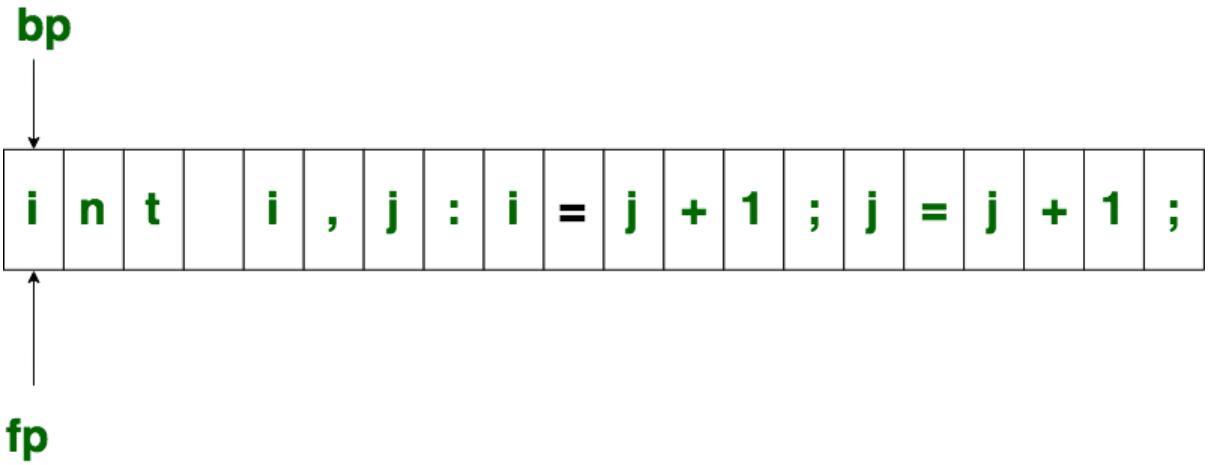
There are also some language for which we can not find a context free grammar.
These language represents real programming constructs as well.

$L_1 = \{ wcw \mid w \text{ is in } (a+b)^* \}$ is not a CFG

$L_2 = \{ a^n b^m c^n d^m \mid n \geq 1 \text{ and } m \geq 1 \}$ is not a CFG

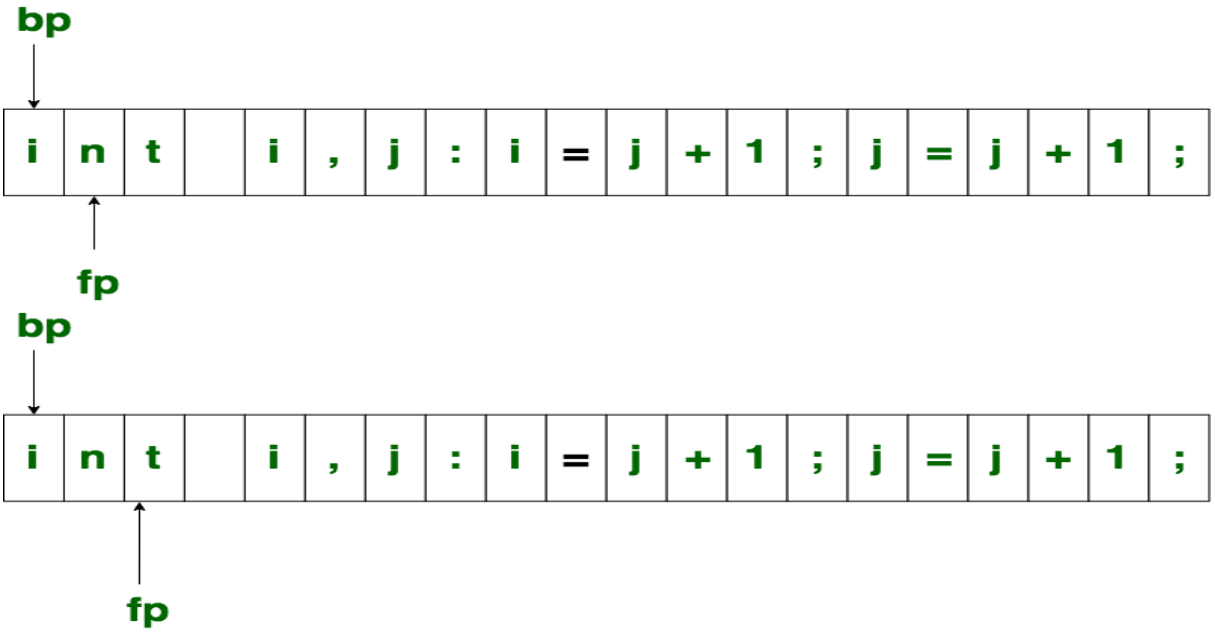
Input Buffering in Compiler Design

The lexical analyzer scans the input from left to right one character at a time. It uses two pointers begin ptr(**bp**) and forward to keep track of the pointer of the input scanned.



Initial Configuration

Initially both the pointers point to the first character of the input string as shown below



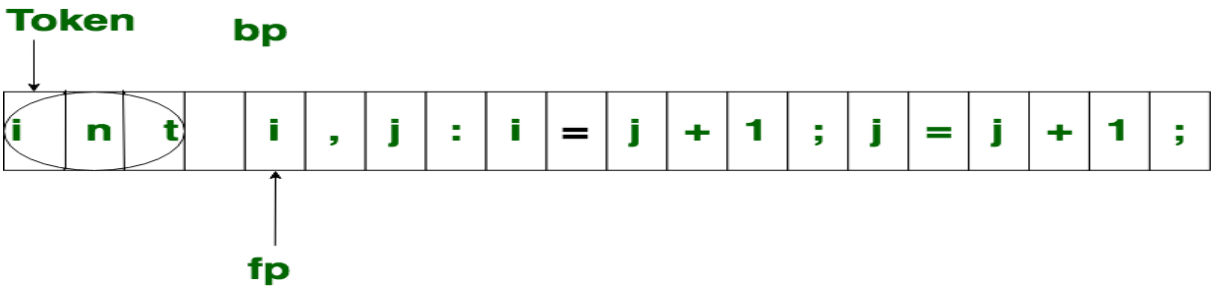
Input Buffering

The forward ptr moves ahead to search for the end of lexeme. As soon as the blank space is encountered, it indicates the end of lexeme. In the above example, as soon as ptr (fp) encounters a blank space the lexeme “int” is identified.

The fp will be moved ahead at white space, when fp encounters white space, it ignores and moves ahead. then both the begin ptr(bp) and forward ptr(fp) are set at the next token.

The input character is thus read from secondary storage, but reading in this way from secondary storage is costly. hence buffering technique is used. A block of data is first read into a buffer,

and then second by lexical analyzer. There are two methods used in this context: One Buffer Scheme, and Two Buffer Scheme. These are explained as follows below.

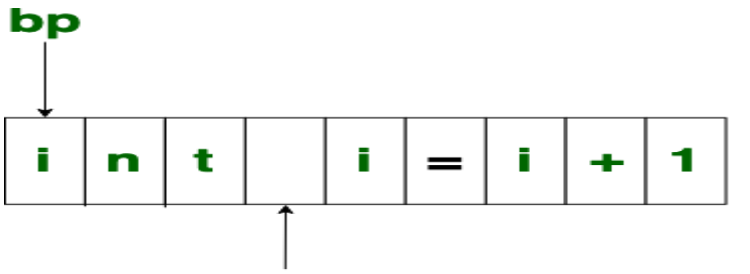


Input buffering

Buffering Methods:

1. One Buffer Scheme:

In this scheme, only one buffer is used to store the input string but the problem with this scheme is that if lexeme is very long then it crosses the buffer boundary, to scan rest of the lexeme the buffer has to be refilled, that makes overwriting the first of lexeme.



One buffer scheme storing input string

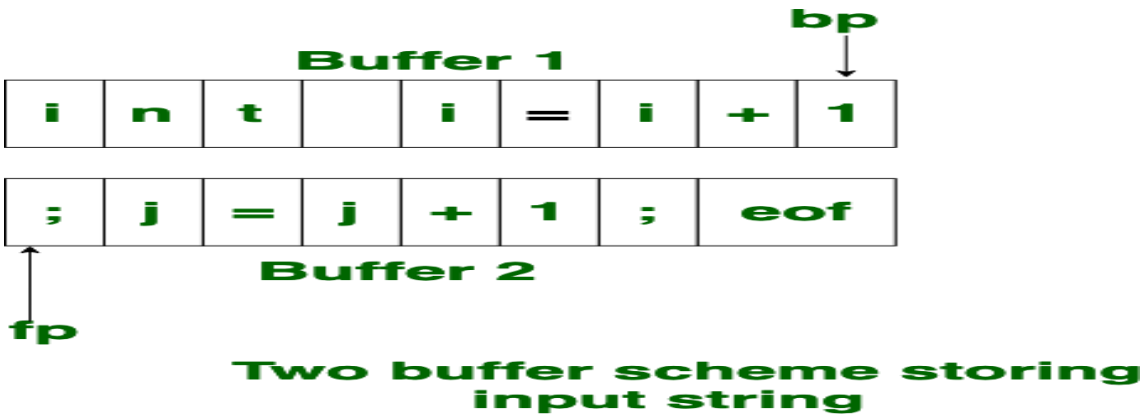
2.Two Buffer Scheme:

To overcome the problem of one buffer scheme, in this method two buffers are used to store the input string. The first buffer and second buffer are scanned alternately. When the end of the current buffer is reached the other buffer is filled. The only problem with this method is that if the length of the lexeme is longer than the length of the buffer then scanning input cannot be scanned completely.

Initially both the bp and fp are pointing to the first character of the first buffer. Then the fp moves towards the right in search of the end of lexeme. as soon as a blank character is recognized, the string between bp and fp is identified as corresponding token. to identify, the boundary of the first buffer end of buffer character should be placed at the end of the first buffer.

Similarly the end of the second buffer is also recognized by the end of the buffer mark present at the end of the second buffer. when fp encounters first eof, then one can recognize end of first buffer and

hence filling up the second buffer is started. in the same way when a second eof is obtained then it indicates a second buffer. alternatively both the buffers can be filled up until the end of the input program and stream of tokens is identified. This eof character introduced at the end is called ***Sentinel*** which is used to identify the end of the buffer.



Two buffer scheme storing input string