Monday
21/07/14

UNIT-1

Over view of Compilation

Phases of Compilation

Lexical Analysis

Regular grammer and Regular expression for common programming language feature

Pass and phases of translation

Interpretation

Boot strapping

Data structure in compilation

LEX (Lexical Analyzer generator)

phases of compilation:-

Compiler:- It converts the high level language to equivalent low level language program. A compiler operates in phases. A phase is a logically interrelated operation that takes source program in one representation and produce output in another representation. There are two parts of compilation.

1) Analysis phase:
   1) Lexical analyzer
   2) Syntax analyzer
   3) Semantic analyzer
2) Synthesis phase:
   4) Intermediate code generation
   5) Code optimizer
   6) Code generator.

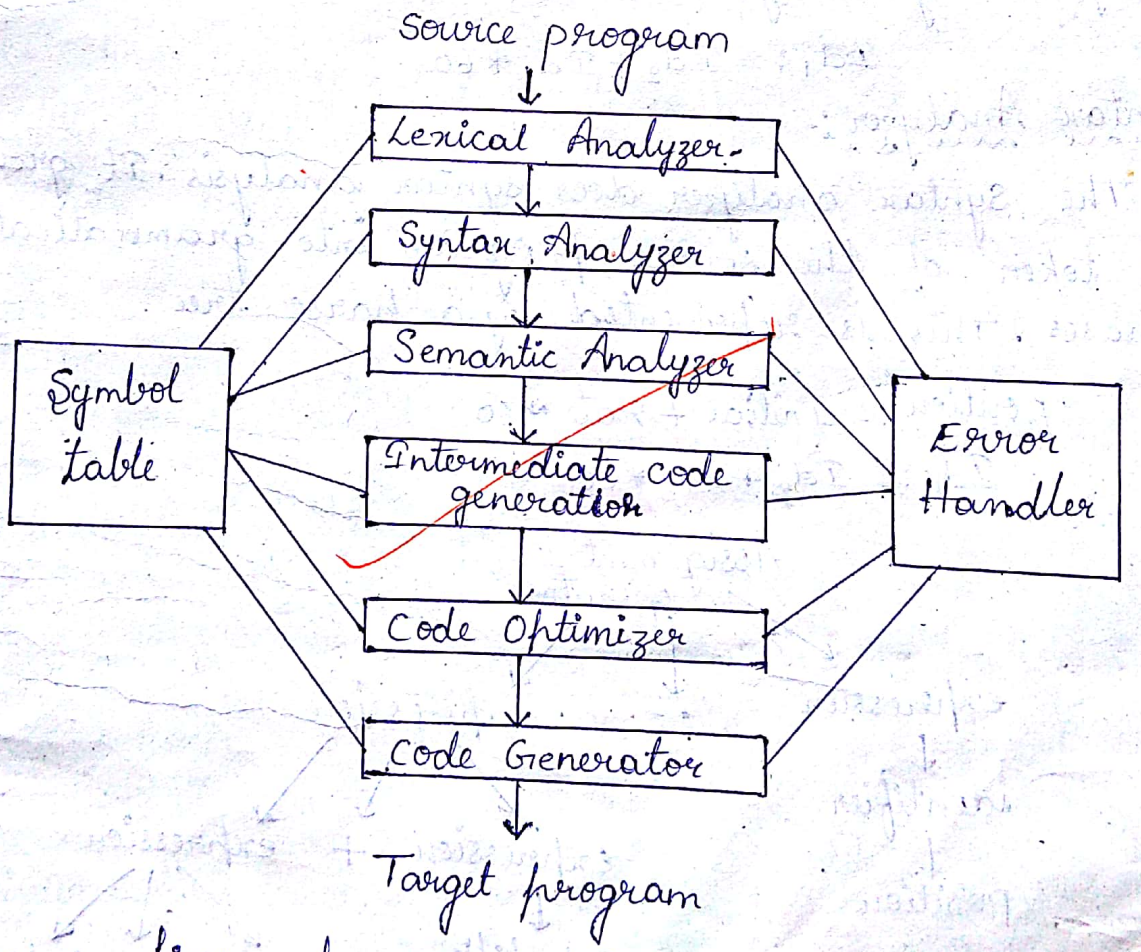


fig:- phases of compilation

Lexical Analyzer :- (Scanning)

The lexical analyzer does lexical analysis. This phase reads the characters in the source program into a stream of tokens. Tokens are the sequence of characters having a collective meaning.

The character sequence forming a token is called the lexeme for the token.

Eg:- position := initial + rate * 60

Where position, initial, rate are identifiers when lexical analyzer finds the identifier position it generates a token simply "Id". Therefore the identifiers are position, initial and rate are the lexeme. The statement after lexical analyzer is given by.

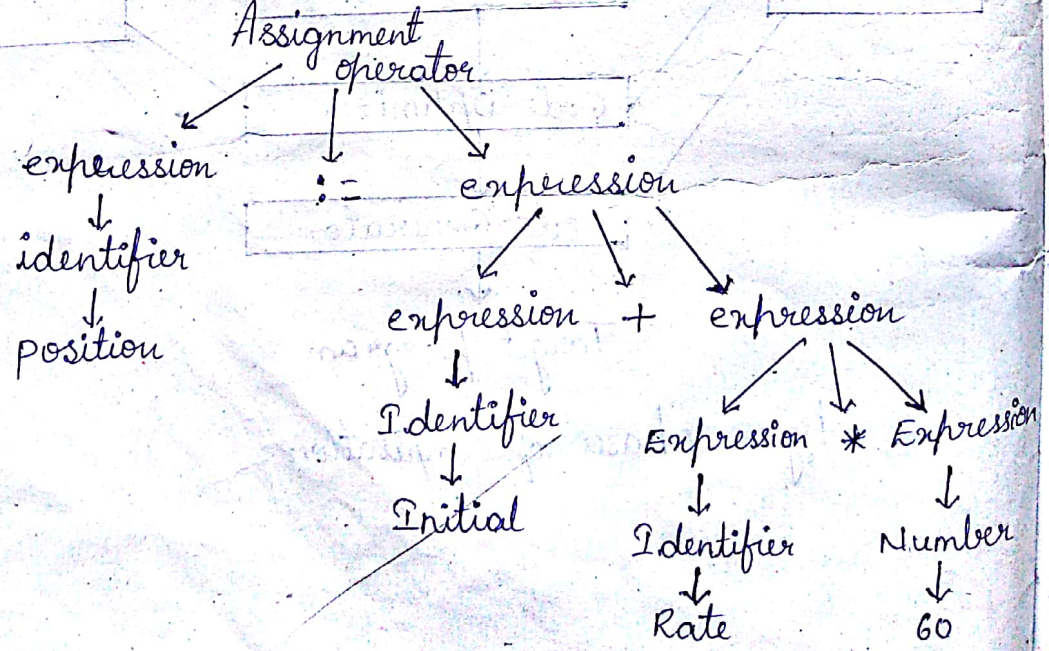
$$Id_1 := Id_2 + Id_3 * 60$$

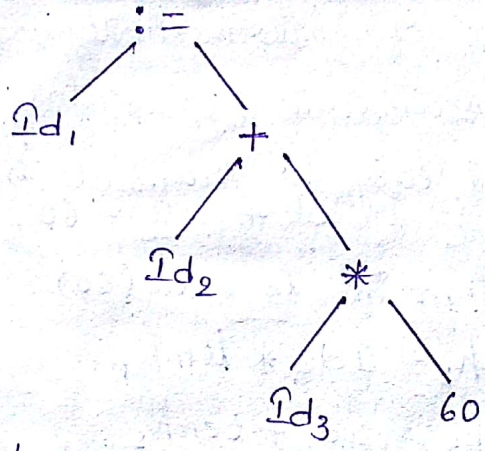
Syntax Analyzer :-

The Syntax analyzer does syntax analysis. It groups the token of the source program into grammatical phases. This is represented by a parse tree.

position := Initial + rate * 60

$Id_1 := Id_2 + Id_3 * 60$





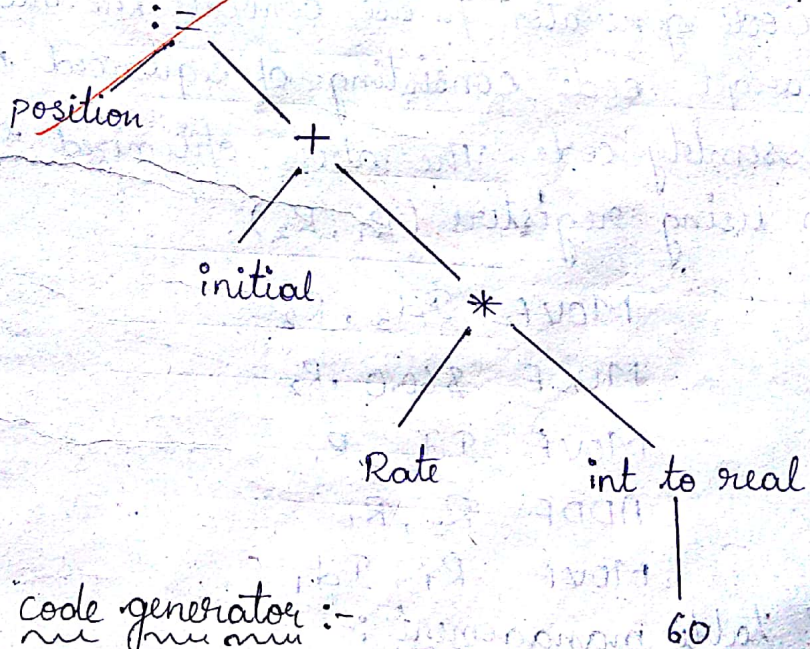
A syntax tree is compressed representation of parse tree in which operators appears as the interior nodes and operands of an operator are the children of the node for the operator.

Semantic Analyzer :-

Semantic analyzer checks the source program for semantic errors.

Semantic analysis perform type checking the characters grouped as taken will be recorded in a table is called as symbol table.

Eg:- position := initial + rate * 60



Intermediate code generator :-

After syntax and semantic analysis some compilers generates an intermediate code representation of source program. This representation should be easy to produce and easy to translate into the target program.

One popular type of intermediate language is called as "3-address-code"

Eg:- position := initial + rate * 60, we get

$temp_1$ = int to real (60)

$temp_2 = Id_3 * temp_1$

$temp_3 = Id_2 + temp_2$

$Id_1 = temp_3$

Code optimization :-

Code optimization phase improves the intermediate code i.e., it reduces the code by removing the repeated or unwanted instructions from the intermediate code.

The above given intermediate code can be optimized as the following code.

Eg:- $temp_1 := Id_3 * 60$

$Id_1 := Id_2 + temp_1$

Code generator :-

Code generator phase converts the intermediate code into target code consisting of sequenced machine code (or) assembly code. The above optimized code can be written using registers (R_1, R_2).

```
MOVF  Id3 , R2
MULF  $60.0 , R2
MOVF  Id2 , R1
ADDF  R2 , R1
MOVF  R1 , Id1
```

Symbol table management :-

* The data structure used to record this information is called symbol table. This data structure allows and to find the record for each identifier quickly and store (or) retrieve data.

Error handler :-

* Error handler is invoked when a fault in the source program is detected.
* Syntax and semantic analysis phases usually handled large no. of errors.

Consider the following fragment of c code

position := initial + rate * 60. by using phases of compilation.

Sol

position := initial + rate * 60

↓

Lexical Analyzer

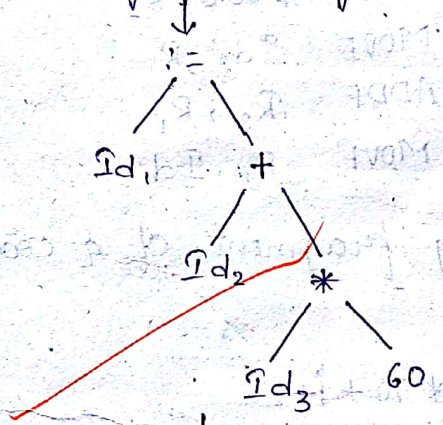
↓

$Id_1 := Id_2 + Id_3 * 60$

↓

Syntax Analyzer

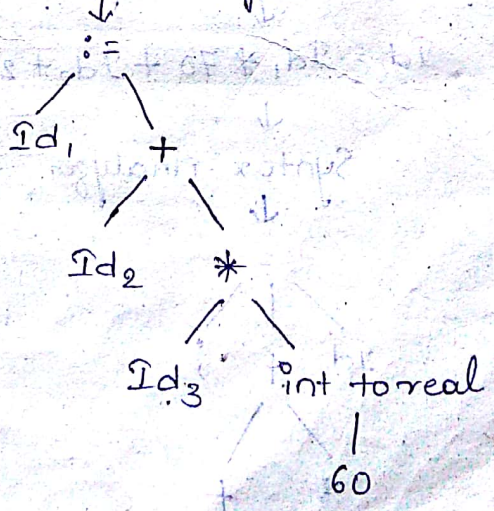
↓



↓

Symantic Analyzer

↓



↓

Intermediate code generator

↓

$temp_1$ = int to real (60)

$temp_2$ = Id_3 * $temp_1$

$temp_3$ = Id_2 + $temp_2$

Id_1 = $temp_3$

↓

Code optimization

↓

$temp_1$:= Id_3 * 60

Id_1 := Id_2 + $temp_1$

↓

Code generator

↓

```
MOVF  Id3 , R2
MULF  $60.0 , R2
MOVF  Id2 , R1
ADDF  R2 , R1
MOVF  R1 , Id1
```

Consider the following feragment of c code float i,j

i = i * 70 + j + 2 ;

i = i * 70 + j + 2

↓

Lexical Analyzer

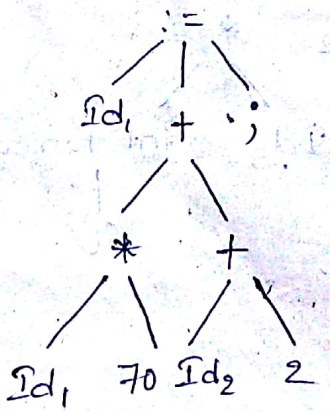
↓

Id_1 = Id_1 * 70 + Id_2 + 2 ;

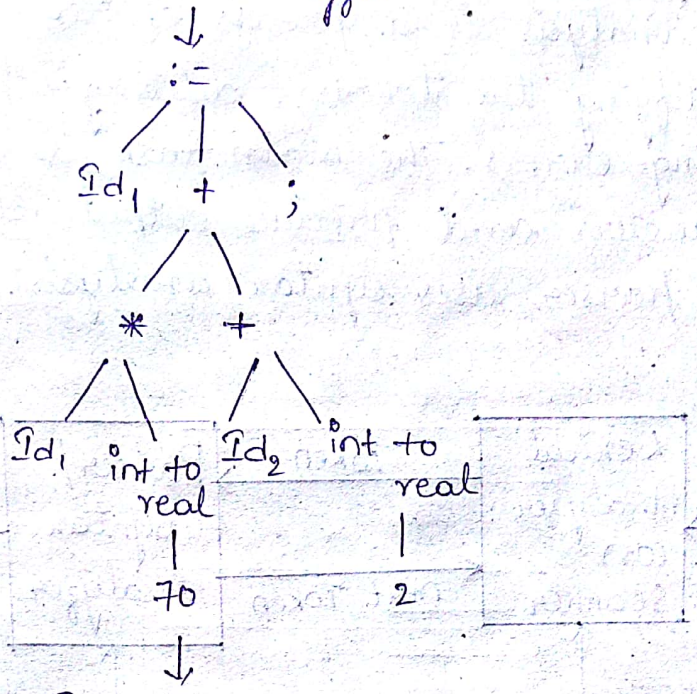
↓

Syntax Analyzer

↓



Semantic Analyzer



Intermediate code generator

$t_1 := \text{int to real}(70)$

$t_2 := Id_1 * t_1$

$t_3 := \text{int to real}(2)$

$t_4 := Id_2 + t_3$

$t_5 := t_2 + t_4$

$Id_1 := t_5$

Code optimizer

$t_1 := Id_1 * 70$

$t_2 := Id_2 * 2$

$Id_1 := t_1 + t_2$

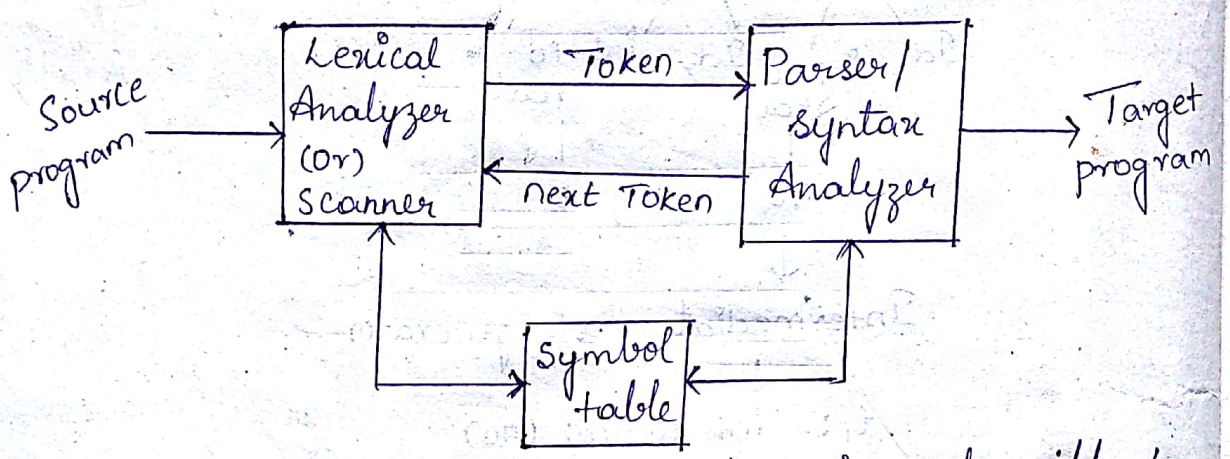
Code generator

```
MOVF   Id1, R2
MULF   $70.0, R2
MOVF   Id2, R1
AddF   $2.0, R1
AddF   R2, R1
MOVF   R1, Id1
```

=

Lexical analyzer :-

Lexical analysis is a process of reading the input string, identifying the tokens, deleting white spaces and reporting errors. The main task is to read the input characters and produce output sequence of tokens. The parser uses syntax analysis.



The scanner of lexical analyzer is responsible for

→ removal of white space and comments
→ Identifying the ⇒ constants
⇒ Identifier
⇒ Keywords.
→ Generate the tokens
→ reporting errors (Handling errors).

Removing of white space and comments :-

Whenever the scanner encounters a blank, tab, a new line characters it blindly deletes. The reason is the parser need not worry about white spaces. Even comments can be ignored by the parser, as they will be deleted by the scanner while generating the tokens.

Identifying : i) Constants :-

Scanner, whenever encounters a constant it makes an entry into symbol table and return the pointer.

ii) Identifier :- An identifier can be a name of a function variable however the grammer of any language will consider identifier as a token.

Eg:- a:=a+1 can be written as Id1 := Id1 + 1.

iii) Keyword :-

→ A keyword in any language will follow all the rules that are imposed on an identifier.

→ The scanner do not get confused between an identifier and keyword.

Generating the tokens :-

Whenever the scanner identifies a valid token, it generates a tuple of the form $<q_1, q_2>$ where q_1 is a valide lexeme and q_2 is pointer to symbol table for q_1.

Reporting errors :-

The lexical analyzer in some compilers is responsible for making a copy of source program and marking errors in it.

The scanner should be careful about duplication of a single error.

The lexical analysis is completely depend on the source language and independent of machine language.

Regular grammer and regular expression for common programming language features.

Regular expression :- Regular expression are useful for representing certain sets of strings in an algebraic fashion.

Regular expression can describes the language accepted by the finite state automata.

Union :- If two R.E R_1 and R_2 is a R.E then

$$R = R_1 + R_2$$

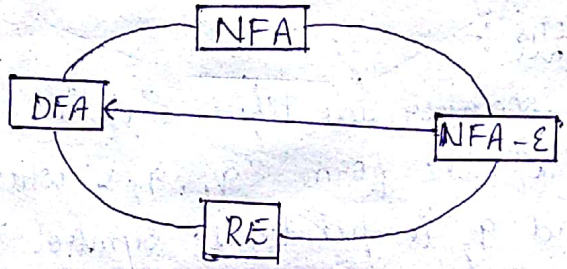
Concatenation :- If two R.E R_1 and R_2 is a R.E then

$$R = R_1 \cdot R_2$$

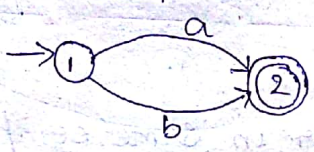
Iteration :- (Closure)

If a R.E R and it can be written as R^* is a regular expression.

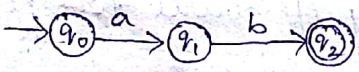
Note : 1 Relationship of FA



1) $a+b$ =



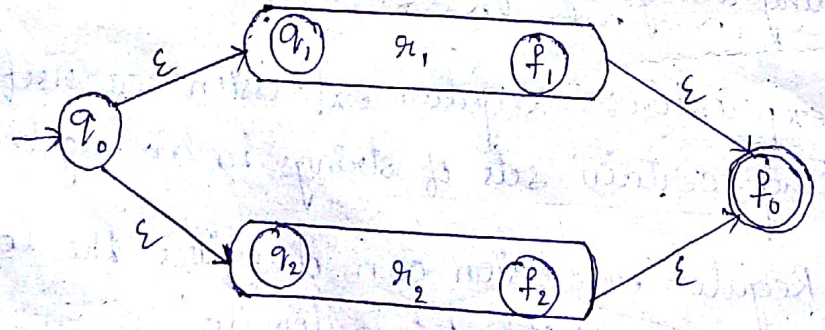
2) $a.b$ =



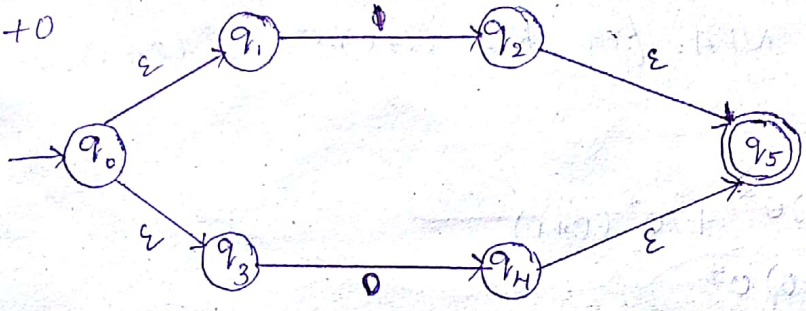
3) a^* =

Note : 2 Let R be a R.E then their exist an NFA with ε transition that accept $L(r)$

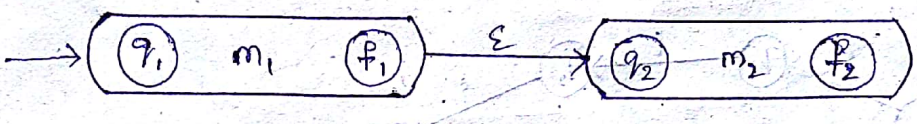
1) $r = r_1 + r_2$



2) 1+0



3) $r = r_1 r_2$



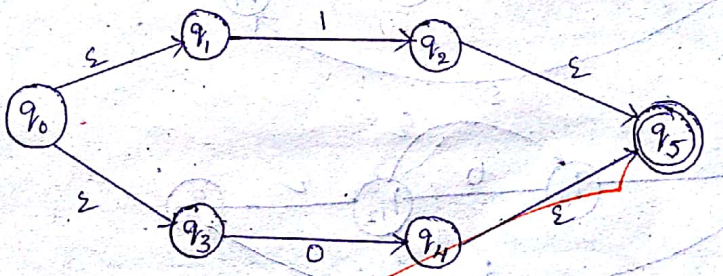
Eg:- Construct NFA for R.E $(1+0)0^*$

Sol Given that

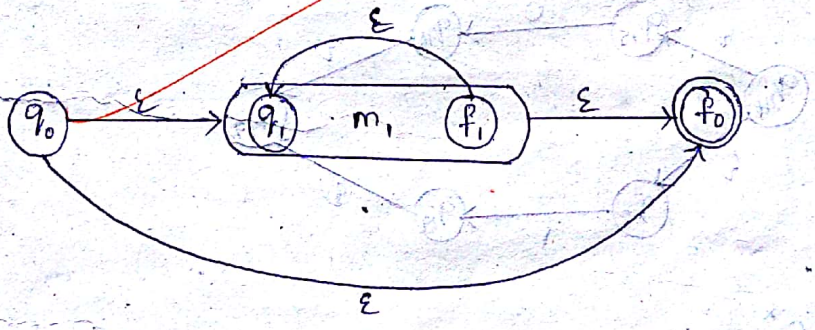
Let $r_1 = 1+0$

$r_2 = 0^*$

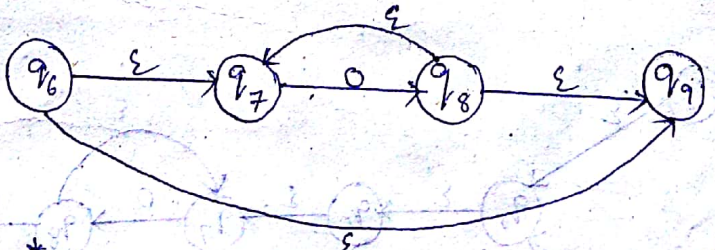
$r_1 =$



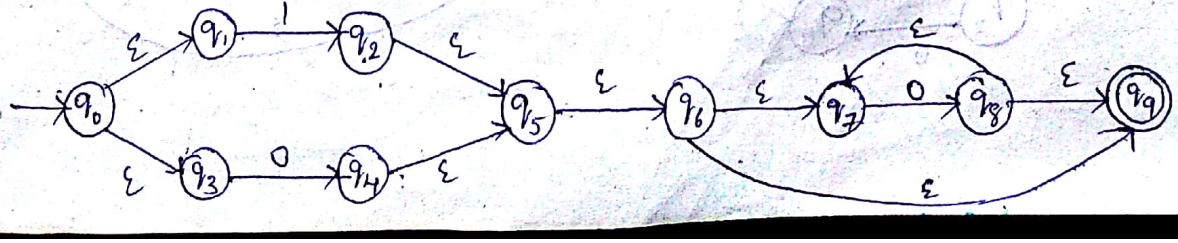
Note:-



$r_2 =$



$R = (1+0)0^*$



14M

Eg:- *** Construct NFA for R.E $(1+0)0^* + 0^*(0+1)$

Sol

Given that

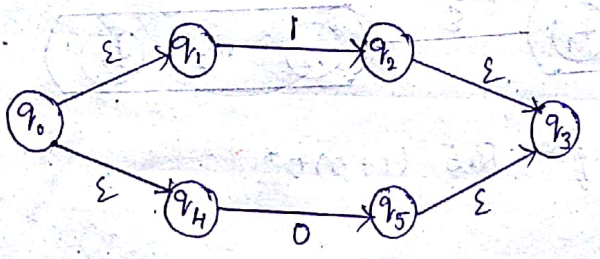
$r = (1+0)0^* + 0^*(0+1)$

$r_1 = (1+0)0^*$

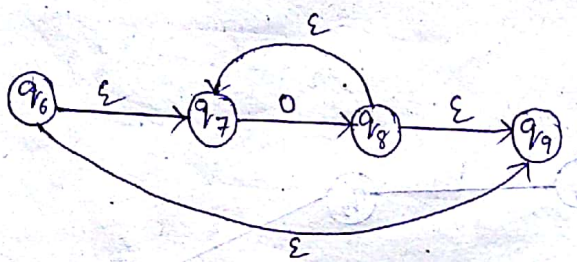
$r_2 = 0^*(0+1)$

$r_3 = 1+0$, $r_4 = 0^*$, $r_5 = 0^*$, $r_6 = 0+1$

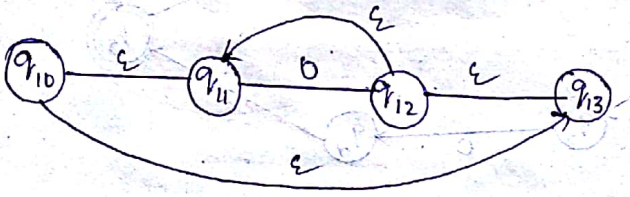
$r_3 = 1+0$



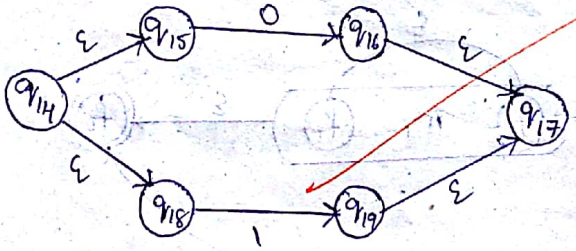
$r_4 = 0^*$



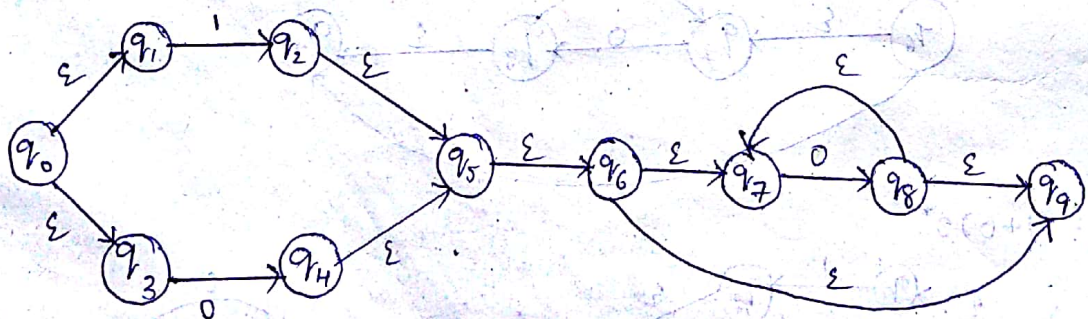
$r_5 = 0^*$



$r_6 = 0+1$

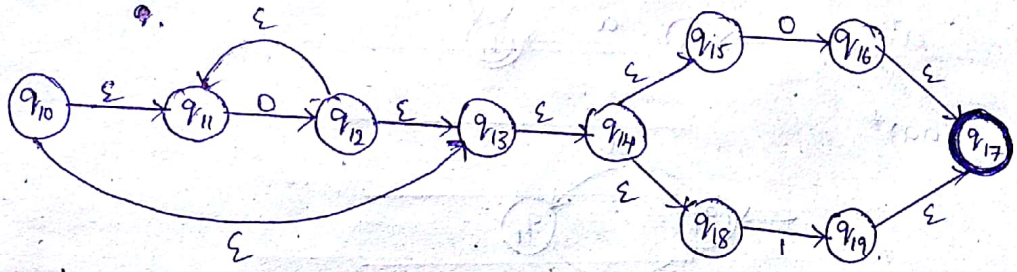


$r_1 = (1+0)0^*$

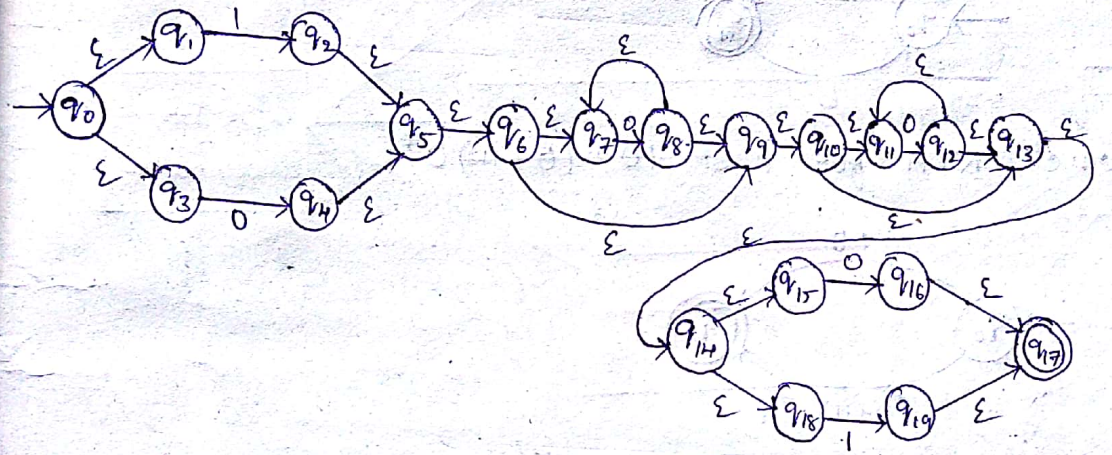


$q_{12} = 0^*(0+1)$

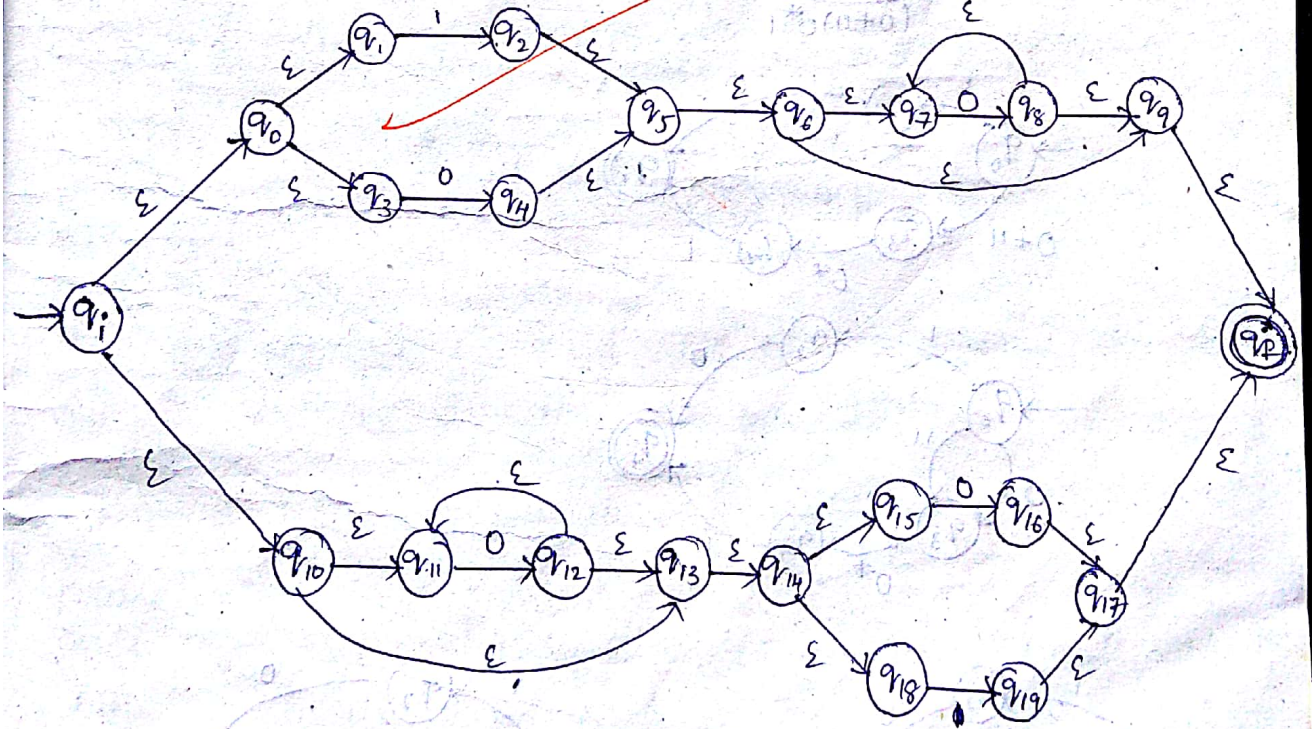
9.



$R = (1+0)0^* \cdot 0^*(0+1)$

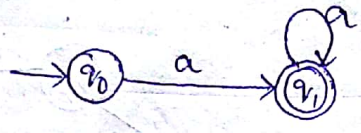


$R = (1+0)0^* + 0^*(0+1)$

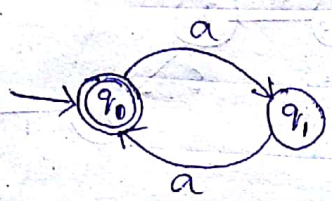


Note 3:-

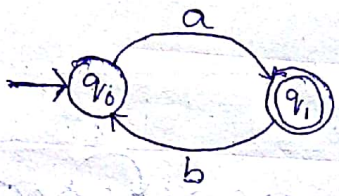
a^+ =



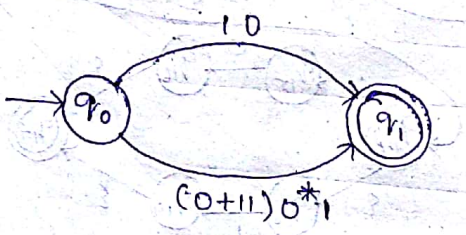
$(aa)^*$ =

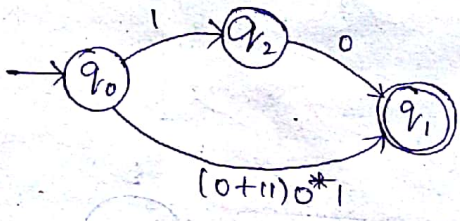


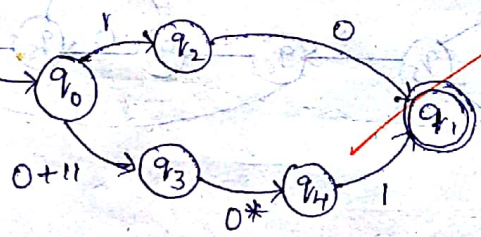
$a(ba)^*$:

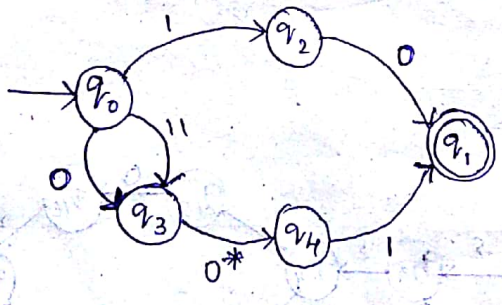


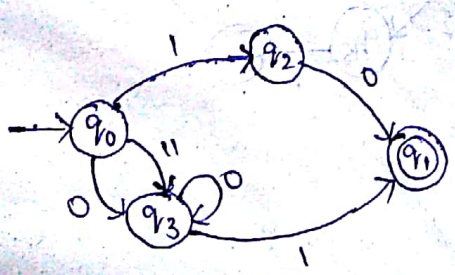
Construct NFA for R.E $10+(0+11)0^*1$

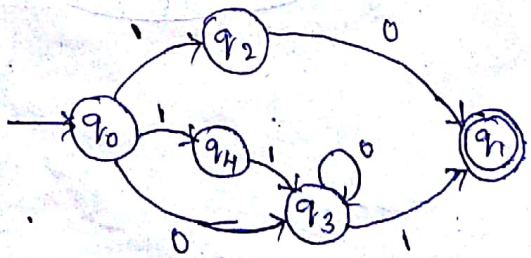












Rules for R.E :-

1. $\Theta + R = R + \Theta = R$
2. $\Theta R = R\Theta = R$
3. $\varepsilon R = R\varepsilon = R$
4. $\varepsilon^* = \varepsilon$ and $\emptyset^* = \emptyset$
5. $R + R = R$
6. $R^* R^* = R^*$
7. $RR^* = R^* R = R^+$
8. $(R^*)^* = R^*$
9. $\varepsilon + RR^* = \varepsilon + R^* R = R^*$
10. $(PQ)^* P = P(QP)^*$

Pass and Phases of translation :-

PASS :- A pass is one complete scan of program which includes reading an input source program and converting the source program into machine understandable form means that ~~target~~ program.

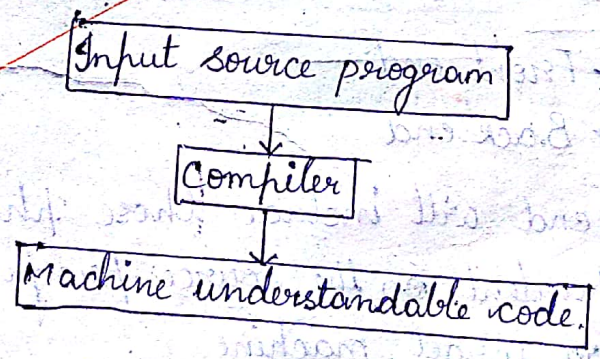
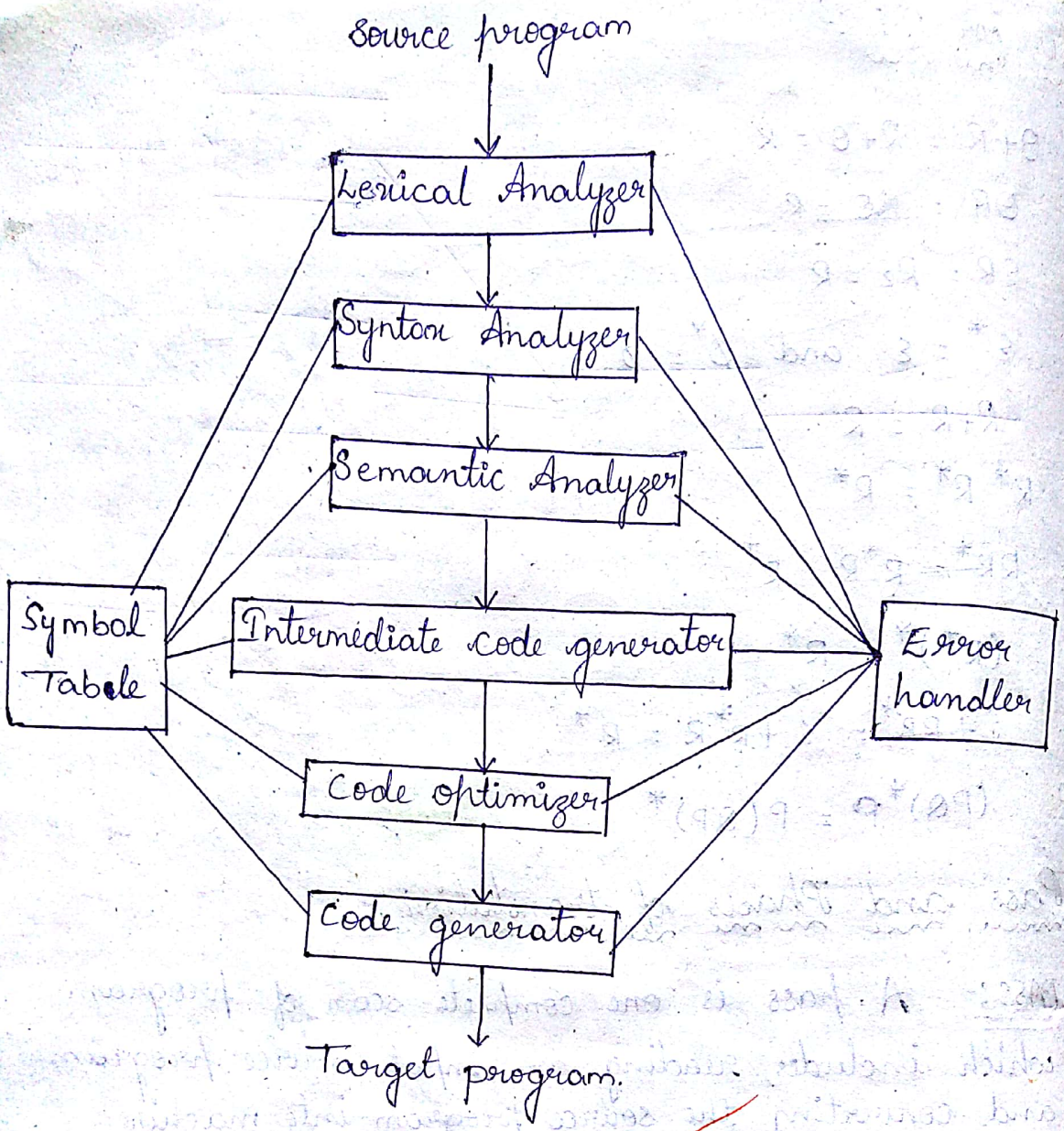


fig: logical structure of pass

Phase :- A computer is divided into no. of parts, segments (or) selections and each of the section is known as phase. A general compiler has 6 phases.

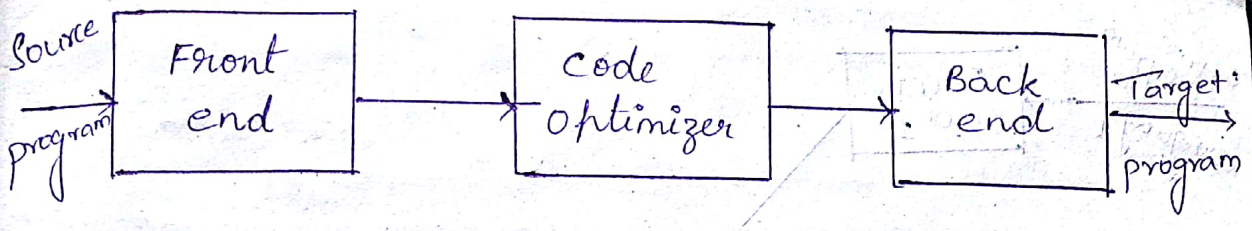


Now the compiler can broadly divided into 2 parts

1) Front end
2) Back end

⇒ The front end will include those phases of compiler which are dependent on the source program and independent on target machine.

⇒ The back end will include those phases of compiler which are independent of the source program and dependent on the target program.



By dividing the compiler into 2 parts we can do the following two things.

1) For single front end we can attach a different back end to compile a language on a machine

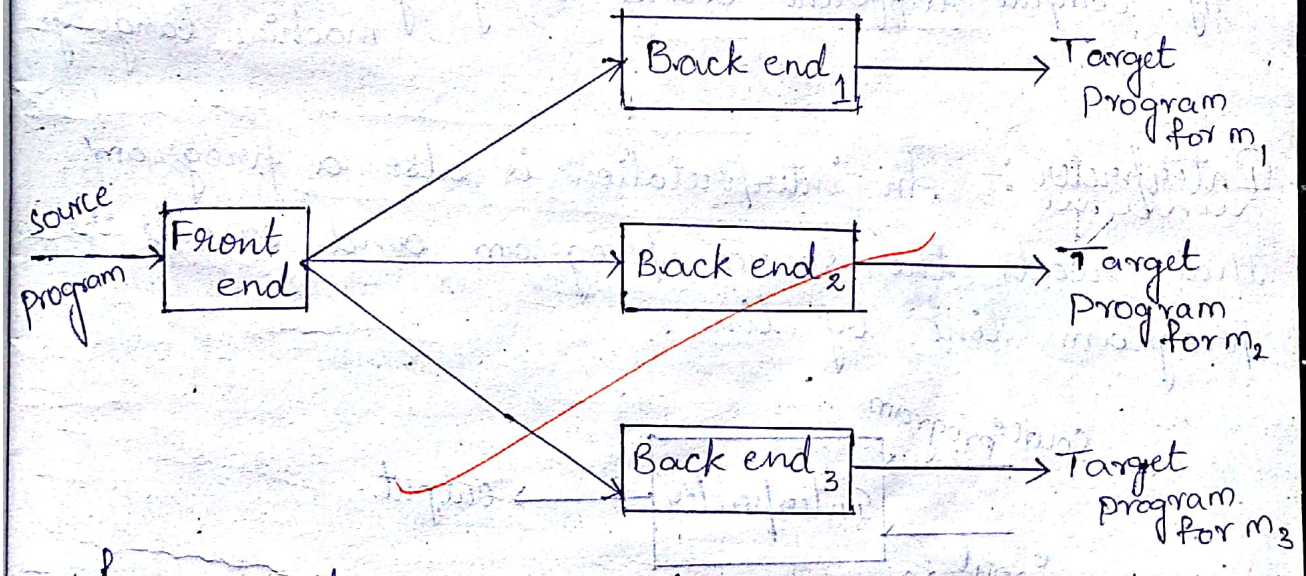


fig:- Compile some source language on different back end (or) Machine language

2) To compile different language on same machine take a back end and attach different front end as necessasary.

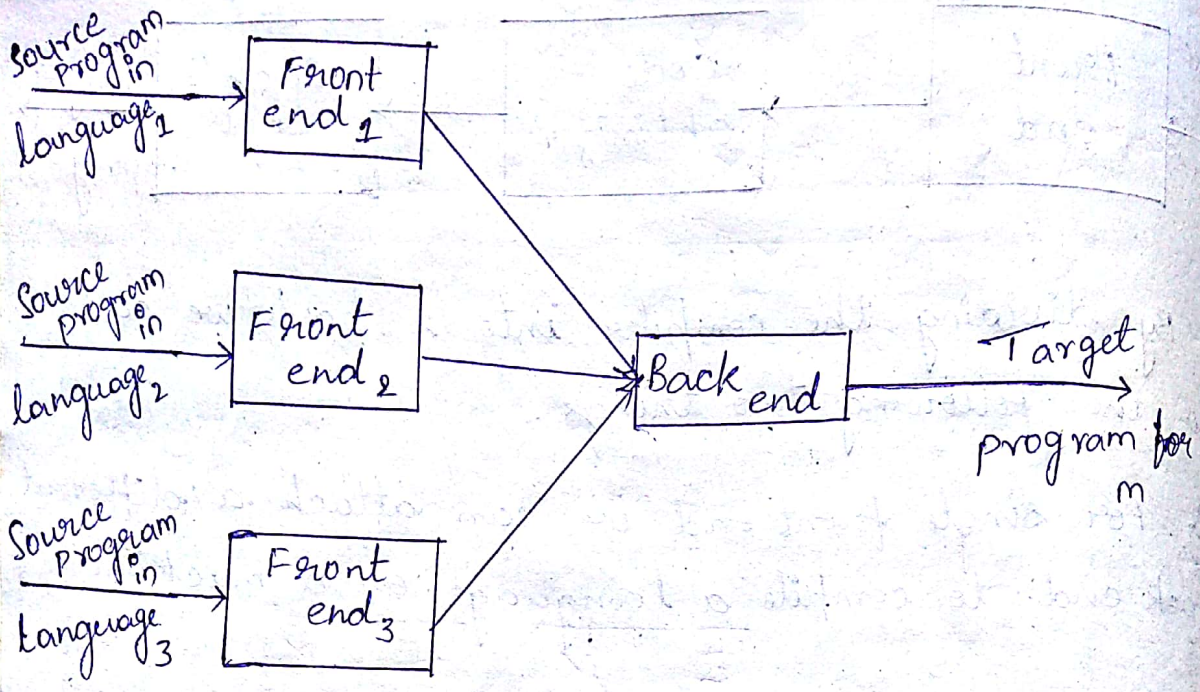
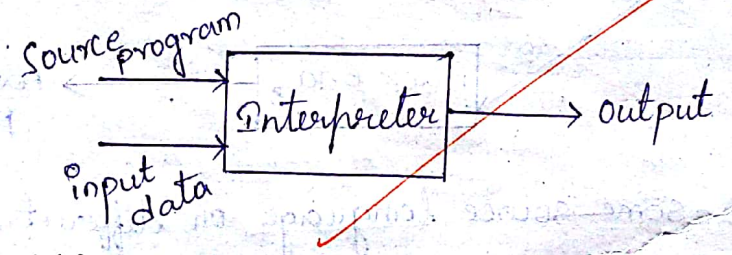


fig: compile different source language on same machine languag

Interpreter :- An interpretation is also a program that reads the source program and execute the program line by line.



Bootstrapping :- It is a process in which simple language is used to translate into more complicated program. This complicate programs can further handle even more complicated program and so on...

Writing a compiler for any high level language is a complicated process. It takes lot of time to write a compiler. Hence simple language is used to generate target code in some stages.

to clearly understand the bootstrapping technique.

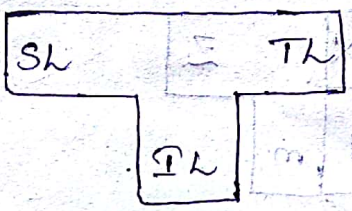
Consider the following scenario, in bootstrapping of a compiler needs three languages they are SL, IL, TL.

Here

SL - Source language

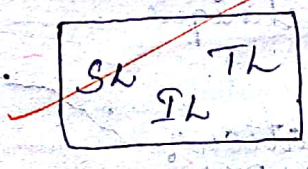
IL - Implimented language

TL - Target language.



This diagram is usually (or) generally called as T-diagram due to shape formed.

→ The relation b/w three languages SL, IL, TL can be written as

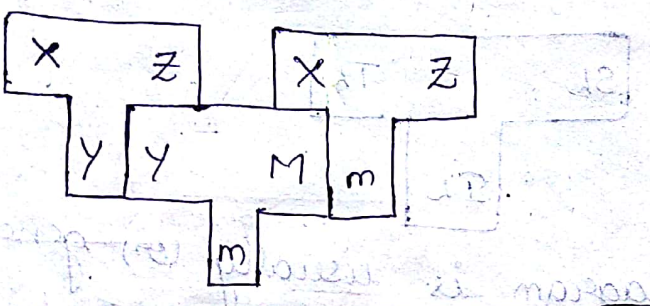


Cross Compiler:- The compiler which runs on machine (m_1) and produce object code for another machine (m_2) is called cross compiler.

Eg:- Suppose if we want to write a cross compiler for new language 'X'. The implimentation language of this compiler is say 'Y' and the target code being generated is in language 'Z'. That is we create X_Y^Z. Now if existing compiler 'Y' runs

on machine and generate code for M. and it denoted by $Y_m M$.

Now if we run $X_Y Z$ using $Y_m M$ then we get a compiler $X_m Z$. That means a compiler for source language X that generates a ~~code~~ Target code in language Z and which runs on a machine M



There fore $X_Y Z + Y_m M = X_m Z$

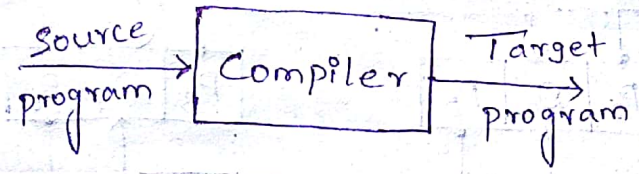
Means that source code 'X' is converted into target code 'Z'. Thus we can successfully implemented bootstrapping technique.

Data structure in compilation:-

A data structure is used to store the data in a machine to compile the compiler. The data structure i.e., used when a compiler is constructed are given bellow. Let us take the c-language, because we all know the data structure in c-language. As a general observation the data structures that are used in a compiler are

1) Arrays
2) Linked Lists
3) Stacks

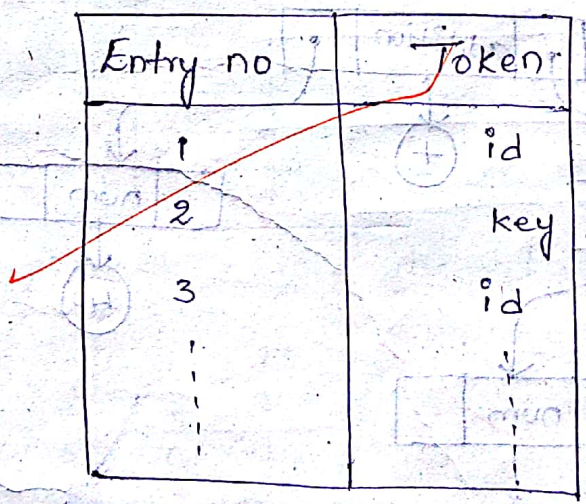
Structure of Compiler :-



The Compilation process looks very simple when we look at the above figure. However, the compiler is also a software. which needs to be written in a programming language.

Array :- An array is a collection of similar data type items in a seqvential form. Now we have looked at the defination for an array, but how does the array relate to data structure used in compiler ?

The six phases of the compiler interract with symbol table, symbol table is used to store the tokens, value, and refferences.



Linked List :-

A linked list is another data structure that is used in a Compiler. It can be considered as an extended from an array. Linked list is a linear data structure of an array but their exist relations (or) links b/w the elements stored.

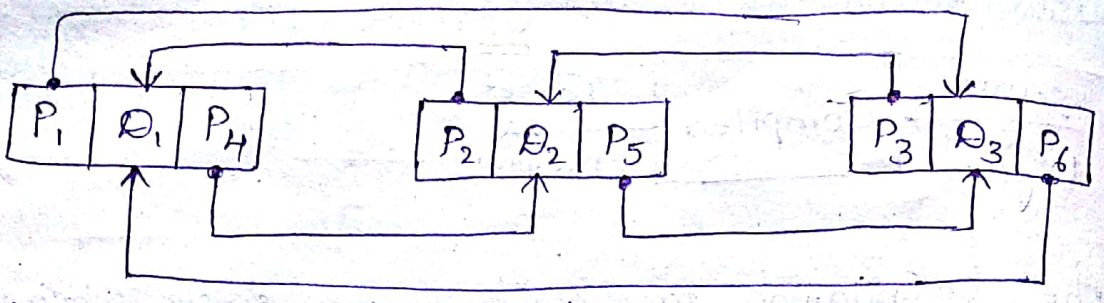
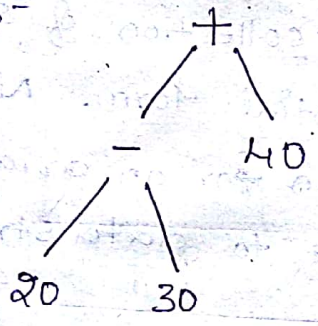


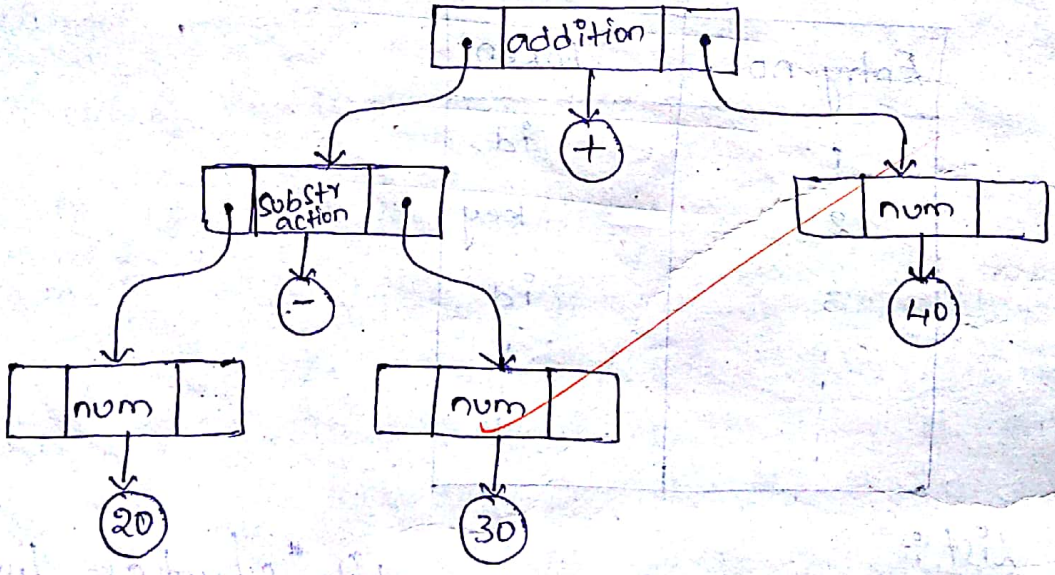
fig: A Linked list for 3 elements

Eg:- 20-30+40

Syntax tree :-



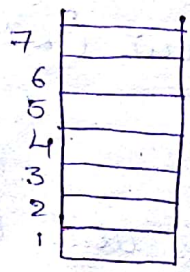
The linked list representation of above syntax tree (or) parse tree can represent bellow



Stack :-

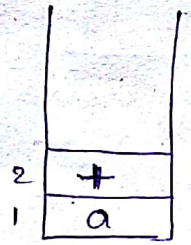
Stack is an array (or) collection of unrelated data which means we can share any type of data in a stack. The type of data in one cell of stack may be different that of another cell.

Eg:- a + b - c * d

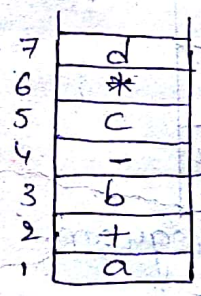


insert 'a' into the stack.

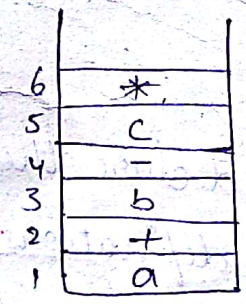
insert '+' into the stack.



Similarly insert b, -, c, *, d, into the stack



Delete the 'd' from the stack



Similarly delete all element from the stack

Now the operation push and pop are mutually exclusive means that either push can be done (or) pop can be done. but not both.

The stacks can be used in the syntax analysis phase of a compiler.

→ Therefore stack is last in first out (LIFO)

LEX :- (Lexical Analyzer):-

Lex program is a tool that is used to generate lexical analyzer (or) scanner. The tool is reffered as a lex compiler and its input specification is lex language.

Any lex program contains three parts (or) section

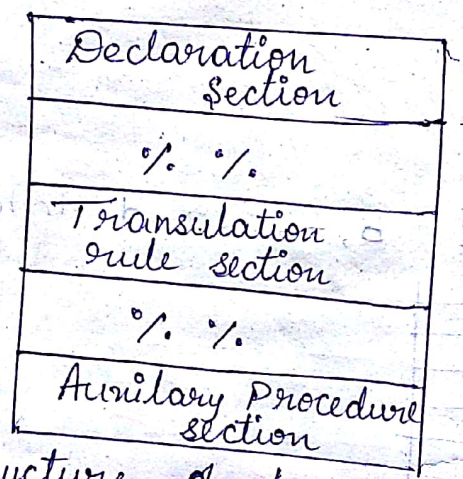


fig: Structure of Lex program.

Each section can be separated by double percentage.

Declaration section:-

This section is used to declare the variables, manfast constants. and regular definations. Here variable can store the value from memory allocation and assign to the value to a perticular variable.

A manfest constant is an identifier i.e., used to represent a constant.

Eg:- In C manfest constant declare as

```
# define MAX = 10
```

→ A regular defination is a seqvence of regular expressions where each regular expression is given a name for notational constant

$$r_1 \longrightarrow re_1$$
$$r_2 \longrightarrow re_2$$
$$r_3 \longrightarrow re_3$$
$$\vdots \qquad \vdots$$
$$r_n \longrightarrow re_m$$

Translation rule section :-

The translation rule is a set of statements which defines the any action to be performed for each regular expression.

Re_1 {method$_1$}
Re_2 {method$_2$}
$\vdots$ $\vdots$
Re_n {methodm}

Auxilary procedure section :-

This procedures are used by the methods in translation rule section. The auxilary procedure can be compiled seperatly, however load with the scanner.

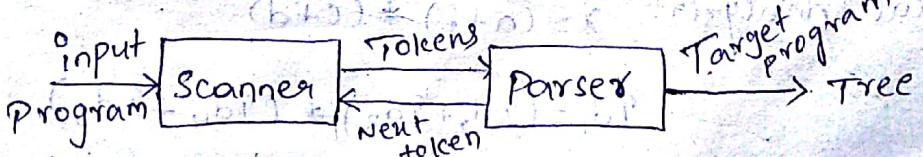


fig: co-ordination b/w scanner & parser

Behaviour of lexical analyzer with parser:-

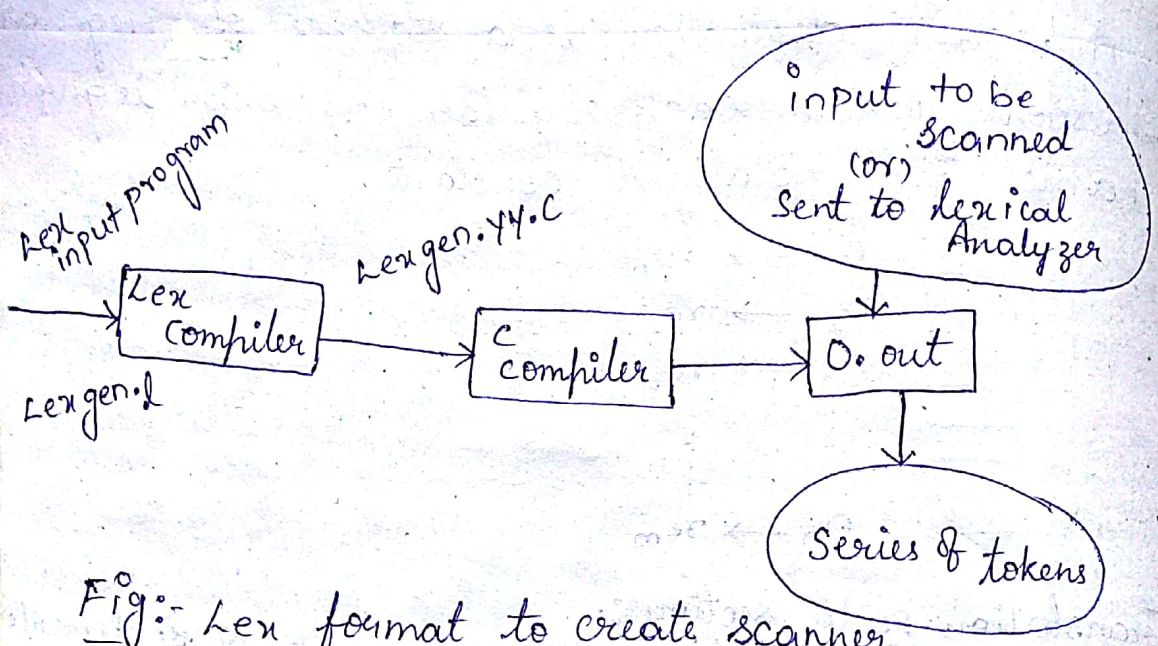


Fig:- Lex format to create scanner

The lex compiler takes as input a lex program i.e. Lexgen.l and generate Lexgen.yy.c, which C-progeram. Then the C-program means that Lexgen .yy.c is compiled by the C-compiler to generate object code file i.e., O. out. O. out becomes the scanner (or) lexical analyzer.

2018 JAN 8M

1) Show that the o/p generated by each phase for the following expression $x = (a+b) * (c+d)$ by phases of compilation.

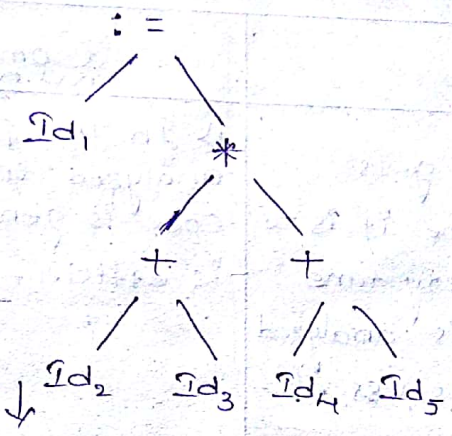
Sol Given expression $x = (a+b) * (c+d)$

Lexical Analyzer:- $x = (a+b) * (c+d)$

$\underline{id}_1 = (\underline{id}_2 + \underline{id}_3) * (\underline{id}_4 + \underline{id}_5)$

↓

Syntax Analyzer:-



Intermediate code generator:-

$temp_1 = Id_2 + Id_3$

$temp_2 = Id_4 + Id_5$

$temp_3 = temp_1 * temp_2$

$Id_1 = temp_3$

Code optimization:-

$temp_1 = Id_2 + Id_3$

$temp_2 = Id_4 + Id_5$

$temp_3 = temp_1 * temp_2$

$Id_1 = temp_3$

Code generator:-

```
MOVF   Id2, R0
ADDF   Id3, R0
MOVF   Id4, R1
ADDF   Id5, R1
MULF   R1, R0
MOVF   R0, Id1
```

Difference between Interpreter and compiler ?

Interpreter

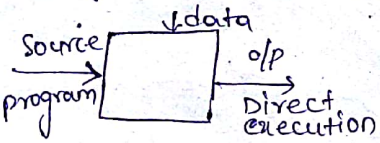
1) The Source program gets interpreted every time it is to be executed and every time the source program is analyzed Hence interpretation is less efficient than compiler.

2) The interpreter do not produce object code.

3) The interpreter can be made Portal because they do not produce object code.

4) Interpreters are simpler & give us improved debugging environment.

5) An interpreter is a kind on transulator which produces the results directly when the source language & data is given to it as i/p



6) Eg of interpreter

A ups debugger in basically a graphical source level debugger but it contains built in c. interpreter which can handle multiple source files

Compiler

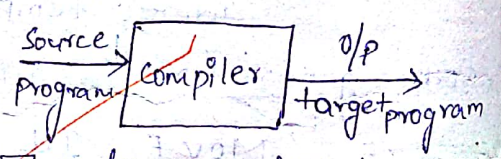
1) In the process of compiler analyzed only once and then code is generated. Hence compiler is efficient than interpreter

2) The compiler produce the object code.

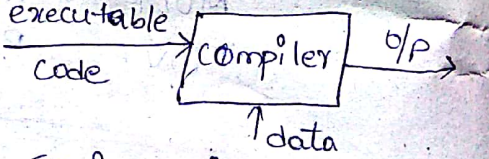
3) The compiler has to be present on the host machine when particular program needs to be compiled.

4) The compiler is a complex Program & it requries large amount of memory.

5) An compiler is a kind of transulator which takes only source program as i/p and converts it into object code.



Then loader perform loading and link editing & prepares an executable code compiler takes this executable code and data as i/p and produced o/p.



6) Eg of compiler

Borland c compiler (or) turbo c compiler compiles the programs written in c (or) c++.

Difference between pass and phase.

PASS	PHASE
1) various phases are logically grouped together to form a pass. The process of complition can be carried out in single pass (or) in multiple parser.	1) The processing of compilation is carried out in various steps. These steps are referred as phases. The phases of compilation are lexical analyzer, syntax analyzer, Intermediate code generator, code optinization, code generator.
2) The task of compilation is carried out in single (or) multiple passes.	2) The task of compilation is carried out in analysis and synthesis phase.
3) The execution of program in passes the compilation model can be viewed as front end and back end model.	3) The phases namely lexical analyzer, syntax analyzer, and Semantic analyzer are machine independent phases and the phases such as code optinization & code generator are machine dependent phases.

2) Show that the o/p generated by each phase for the following expression i := i * 70 + j + 2 by phases of compilation.

Sol Given that i := i * 70 + j + 2

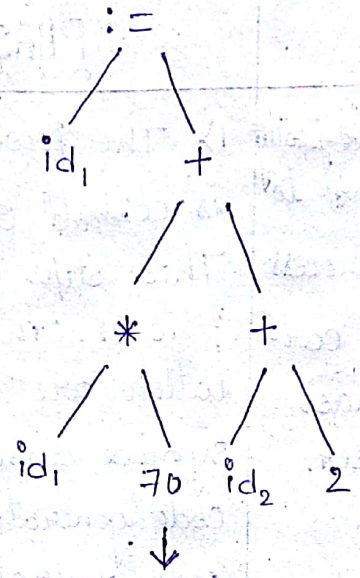
Lexical Analyzer :-

i := i * 70 + j + 2

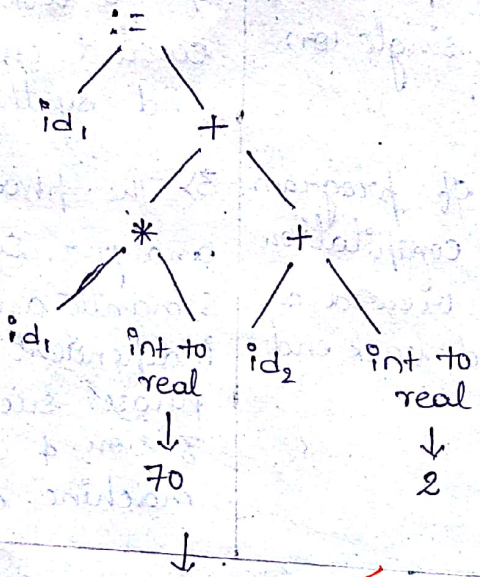
$id_1 := id_1 * 70 + id_2 + 2$

↓

Syntax Analyzer :-



Semantic analyzer :-



Intermediate code generator :-

$t_1 := \text{int to real}(70)$

$t_2 := id_1 * t_1$

$t_3 := \text{int to real}(2)$

$t_4 := id_2 + t_3$

$t_5 := t_2 + t_4$

$id_1 := t_5$

Code optimization :-

$t_1 := id_1 * 70.0$

$t_2 := id_2 * 2.0$

$t_3 := t_1 + t_2$

$id_1 := t_3$

Code generator :-

```
MOVF   id1, R1
MULF   #20.0, R1
MOVF   id2, R2
ADDF   #2.0, R2
ADDF   R2, R1
MOVF   R1, id1
```