PRINCIPLES OF COMPILER DESIGN

1. Introduction to compilers:-

A compiler is a program that reads a program written in one language (source language (or) high level language) and translates it into an equivalent program in another language.(target language (or) low level language)

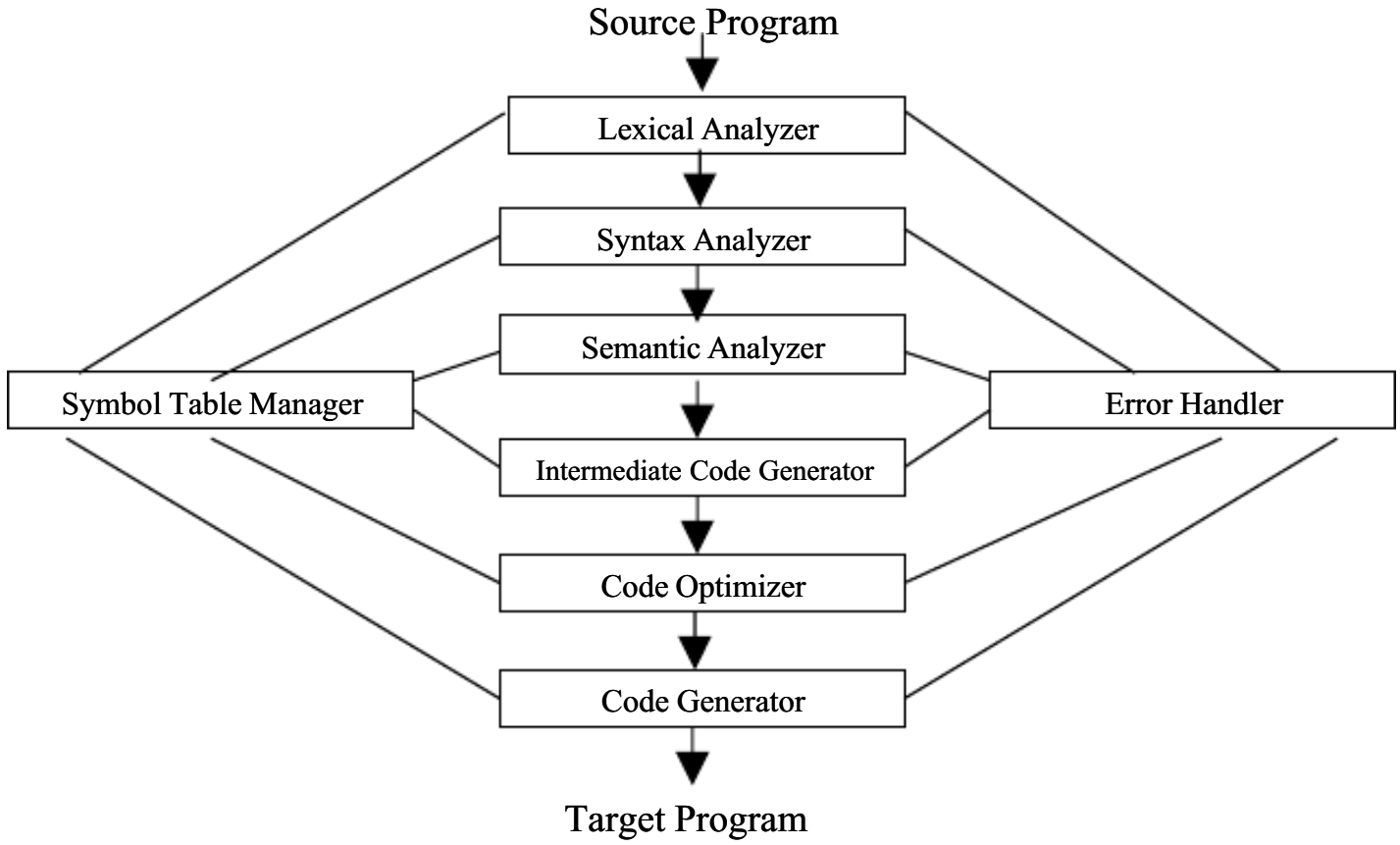
Compiler:- It converts the high level language into an equivalent low level language program.

Assembler:- It converts an assembly language(low level language) into machine code.(binary representation)

PHASES OF COMPILER

There are two parts to compilation. They are

(i) Analysis Phase
(ii) Synthesis Phase



Analysis Phase:-

The analysis phase breaks up the source program into constituent pieces. The analysis phase of a compiler performs,

1. Lexical analysis
2. Syntax Analysis
3. Semantic Analysis

1.Lexical Analysis (or) Linear Analysis (or) Scanning:-

The lexical analysis phase reads the characters in the program and groups them into tokens that are sequence of characters having a collective meaning.

Such as an Identifier, a Keyboard, a Punctuation, character or a multi character operator like ++.

" The character sequence forming a token is called **lexeme**"

For Eg. Pos = init + rate * 60

Lexeme	Token	Attribute value
rate	ID	Pointer to symbol table
+	ADD	
60	num	60
init	ID	Pointer to symbol table

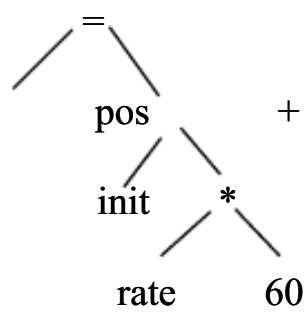
2. Syntax Analysis (or) Hierarchical Analysis:-

Syntax analysis processes the string of descriptors (tokens), synthesized by the lexical analyzer, to determine the syntactic structure of an input statement. This process is known as **parsing.**

ie, Output of the parsing step is a representation of the syntactic structure of a statement.

Example:-

pos = init + rate * 60



3. Semantic Analysis:-

The semantic analysis phase checks the source program for *semantic errors.*

Processing performed by the semantic analysis step can classified into

a. Processing of declarative statements
b. Processing of executable statements

During semantic processing of declarative statements items of information are added to the *lexical tables*.

Example:- (symbol table or lexical table)

real a, b;

id	a	real	length	……
id	b	real	length	…..

Synthesis Phase:-

1. Intermediate code generation
2. Code optimization
3. Code Generator

1. Intermediate code generation:-

After syntax and semantic analysis some compilers generate an explicit intermediate representation of the source program. This intermediate representation should have two important properties.

a. It should be easy to produce
b. It should be easy to translate into the target program.

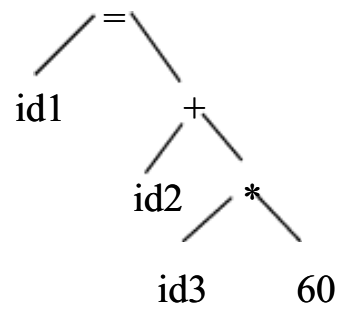
We consider the intermediate form called "Three Address Code". It consists of sequence of instructions, each of which has atmost three operands.

Example:-

pos = init + rate * 60
pos = init + rate * int to real (60)

Might appear in three address code as,

temp1 = int to real (60)
temp2 = id3 * temp1
temp3 = id2 + temp2
id1 = temp3



2. Code Optimization:-

The code optimization phase attempts to improve the intermediate code, so that faster running machine code will result.

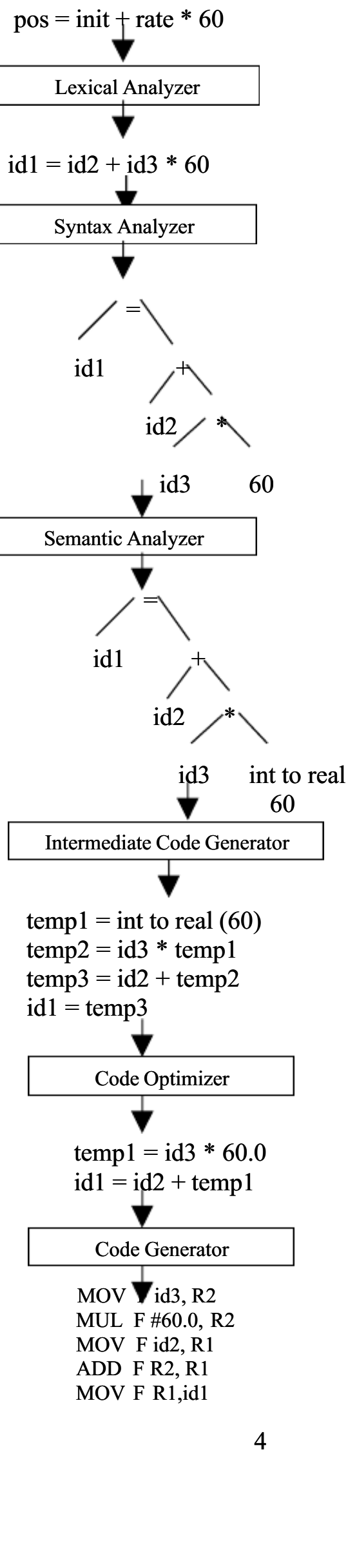
3. Code Generation:-

The final phase of the compiler is the generation of the target code or machine code or assembly code.
Memory locations are selected for each of the variables used by the program. Then intermediate instructions are translated into a sequence of machine instructions that perform the same task.

Example:-

```
MOV  F id3, R2
MUL  F #60.0, R2
MOV  F id2, R1
ADD  F R2, R1
MOV  F R1,id1
```

Translation of a statement



Symbol table management:-

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 record for each identifier quickly and to store or retrieve data from that record quickly.

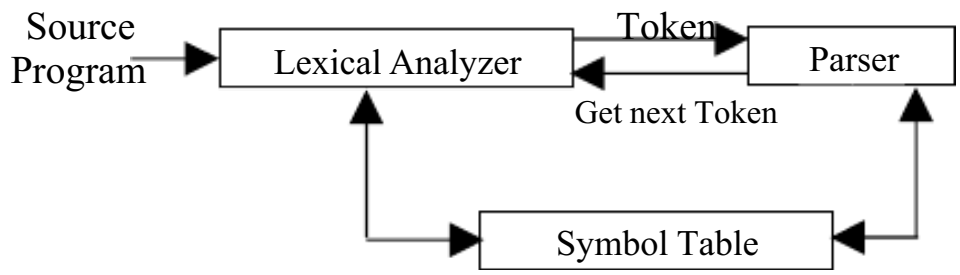
Error Handler:-

Each phase can encounter errors.

- The lexical phase can detect errors where the characters remaining in the input do not form any token of the language.
- The syntax analysis phase can detect errors where the token stream violates the structure rules of the language.
- During semantic analysis, the compiler tries to construct a right syntactic structure, but no meaning to the operation involved.
- The intermediate code generator may detect an operator whose operands have in compatible types.
- The code optimizer, doing control flow analysis may detect that certain statements can never be reached.
- The code generator may find a compiler created constant that is too large to fit in a word of the target machines.

Role of Lexical Analyzer:-

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



After receiving a "get next token" command from the parser, the lexical analyzer reads input characters until it can identify a next token.

Token:-

Token is a sequence of characters that can be treated as a single logical entity. Typical tokens are,

(a) Identifiers
(b) Keywords
(c) Operators
(d) Special symbols
(e) Constants

Pattern:-

A set of strings in the input for which the same token is produced as output, this set of strings is called pattern.

Lexeme:-

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

Finite Automata

Definition:-

A recognizer for a language is a program that takes as input a string x and answers "yes" if x is a sentence of the language and "no" otherwise.

A better way to covert a regular expression to a recognizer is to construct a generalized transition diagram from the expression. This diagram is called finite automation.

A finite automation can be,

1. Deterministic finite automata
2. Non-Deterministic finite automata

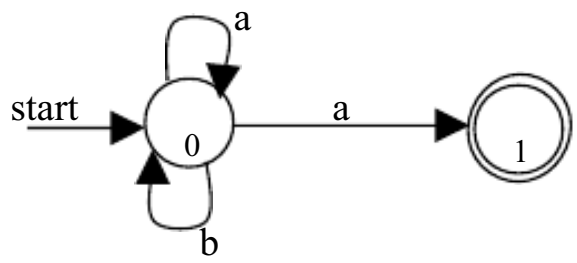
1. **Non – deterministic Finite Automata**:- [**NFA**]

A NFA is a mathematical model that consists of,

1. a set of states S
2. a set of input symbol Σ
3. a transition function δ
4. a state S_0 that is distinguished as start state
5. a set of states F distinguished as accepting state. It is indicated by double circle.

Example:-

The transition graph for an NFA that recognizes the language (a/b)* a



The transition table is,

State	Input Symbol	
	a	b
0	0,1	0
1	-	-

2. **Deterministic Finite Automata**:- [**DFA**]

A DFA is a special case of non – deterministic finite automata in which,

1. No state has an ε – transition
2. For each state S and input symbol there is atmost one edge labeled a leaving S.

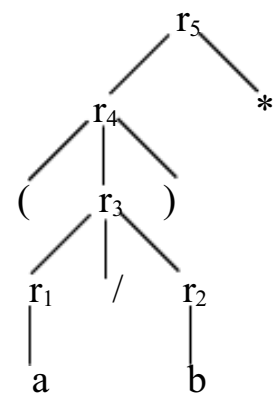
PROBLEM:-

1. Construct a non – deterministic finite automata for a regular expression (a/b)*

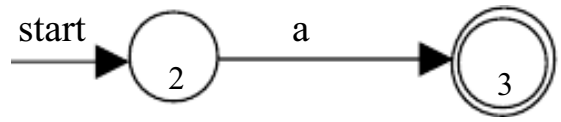
Solution;-

r = (a/b)*

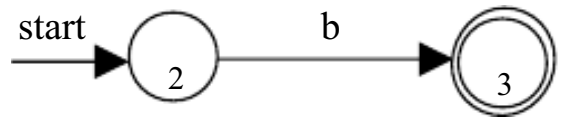
Decomposition of (a/b)* (parse tree)



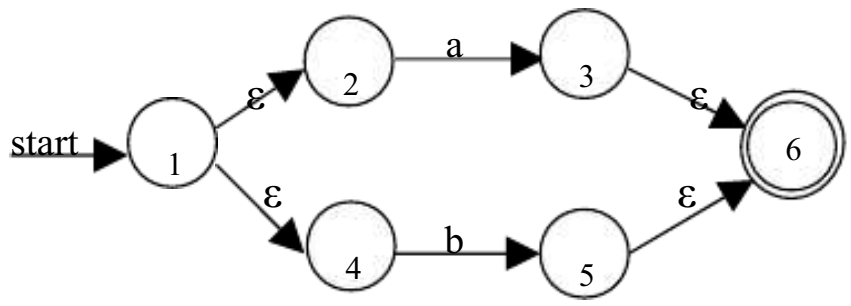
For r_1 construct NFA



For r_1 construct NFA

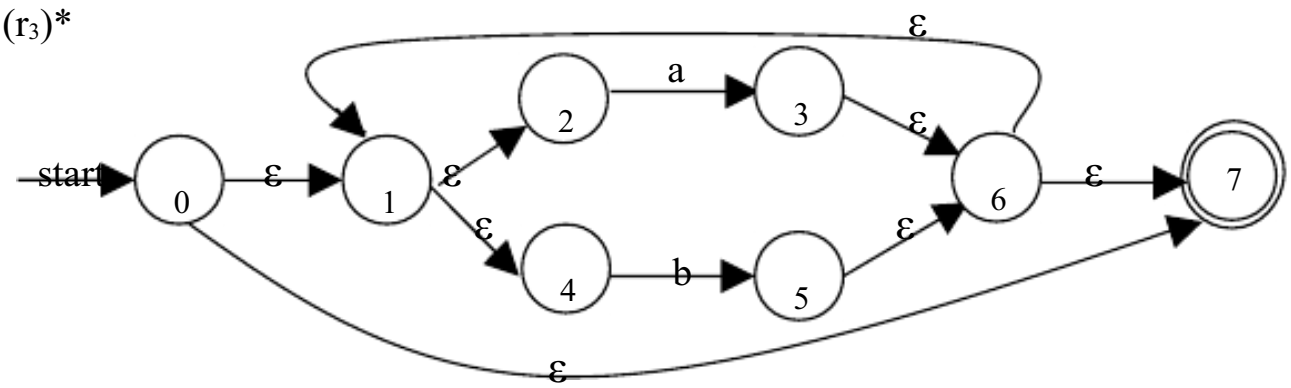


NFA for $r_3 = r_1/r_2$



NFA for r_4, that is (r_3) is the same as that for r_3.

NFA for $r_5 = (r_3)^*$



Problem 2: Tabulate the operator precedence relation for the grammar

E→E+E | E-E | E*E | E/E | E▲E | (E) | -E | id

Solution:-

Assuming 1.▲ has highest precedence and right associative

2. * and / have next higher precedence and left associative

3. + and – have lowest precedence and left associative

	+	-	*	/	▲	id	(	)	$
+	>	>	<	<	<	<	<	>	>
-	>	>	<	<	<	<	<	>	>
*	>	>	>	>	<	<	<	>	>
/	>	>	>	>	<	<	<	>	>
▲	>	>	>	>	<	<	<	>	>
id	>	>	>	>	>	-	-	>	>
(	<	<	<	<	<	<	<	=	-
)	>	>	>	>	>	-	-	>	>
$	<	<	<	<	<	<	<	-	-

Derivations:-

The central idea is that a production is treated as a rewriting rule in which the non-terminal in the left side is replaced by the string on the right side of the production.

For Ex, consider the following grammar for arithmetic expression,

E→E+E | E*E | (E) | -E |id

That is we can replace a single E by –E. we describe this action by writing

E => -E , which is read "E derives –E"

E→(E) tells us that we could also replace by (E).

So, E*E => (E) * E or E*E => E* (E)

We can take a single E and repeatedly apply production in any order to obtain sequence of replacements.

E => -E

E => -(E)

E => -(id)

We call such sequence of replacements is called *derivation*.

Suppose α A β => α γ β then

A→ γ is a production and α and β are arbitrary strings of grammar symbols.

If $\alpha_1 \Rightarrow \alpha_2 \ldots\ldots \Rightarrow \alpha_n$ we say α1 derives α_n

The symbol,

=> means " derives in one step"

=> means "derives zero or more steps"

=> means "derives in one or more steps"

Example:- E→E+E | E*E | (E) | -E |id. The string –(id+id) is a sentence of above grammar.

E => -E
=> -(E)
=> -(E+E)
=> - (id+E)
=> -(id+id)

The above derivation is called left most derivation and it can be re written as,

E => -E

=> -(E)

=> -(E+E)

=> - (id+E)

=> -(id+id)

we can write this as E => -(id+id)

Example for Right most Derivation:-

Right most derivation is otherwise called as *canonical derivations*.

E => -E

=> -(E)

=> -(E+E)

=> - (id+E)

=> -(id+id)

Parse trees & Derivations:-

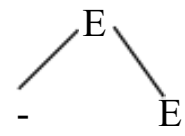
A parse tree is a graphical representation for a derivation that filters out the choice regarding replacement order.

For a given CFG a parse tree is a tree with the following properties.

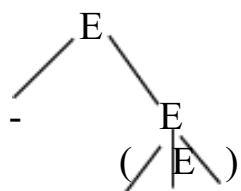
1. The root is labeled by the start symbol
2. Each leaf is labeled by a token or ε
3. Each interior node is labeled by a NT

Ex.

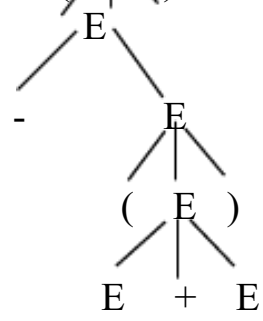
E => -E



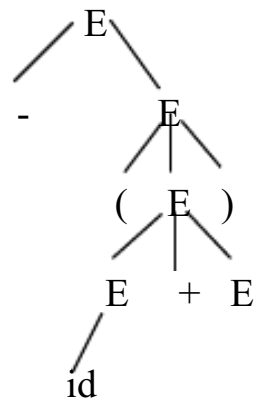
E => -(E)



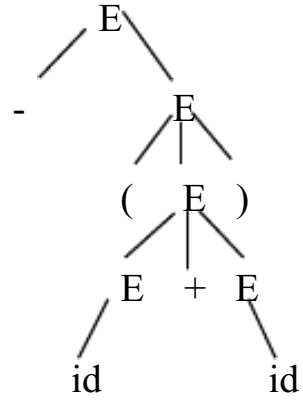
E => -(E+E)



E => -(id+E)



E => -(id+E)



Top-Down Parsing:-

Top down parser builds parse trees starting from the root and creating the nodes of the parse tree in preorder and work down to the leaves. Here also the input to the parser is scanned from left to right, one symbol at a time.

For Example,

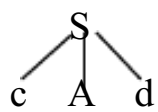
S→cAd

A→ab/a and the input symbol w=cad.

To construct a parse tree for this sentence in top down, we initially create a parse tree consisting of a single node S.

An input symbol of pointer points to c, the first symbol of w.

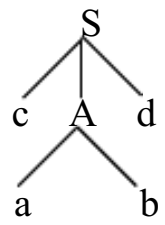
w = cad



The leftmost leaf labeled c, matches the first symbol of w. So now advance the input pointer to 'a' the second symbol of w.

w = cad

and consider the next leaf, labeled A. We can then expand A using the first alternative for A to obtain the tree.



We now have a match for the second input symbol. Advance the input pointer to d,

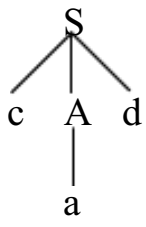
w = cad

We now consider the third input symbol, and the next leaf labeled b. Since *b does not match d,* we report failure and go back to A to see whether there is another alternative for A.

In going back to A we must reset the input pointer to position 2.

W = cad

We now try the second alternative for A to obtain the tree,



The leaf a matches the second symbol of w and the leaf d matches the third symbol. Now we produced a parse tree for w = cad using the grammar S→cAd and A→ab/a.
This is successful completion.

FOLLOW(F) = FIRST(T′) = { * , ε} + FOLLOW(T′) = { *, + ,) ,$ }

(iii) Numbering the Grammar:-

1. E→E+T
2. E→T
3. T→T*F
4. T→F
5. F→(E)
6. F→id

Augmented Grammar

E′→E

E→E+T

E→T

T→T*F

T→F

F→(E)

F→id

Closure (I′)

E′→.E
E→.E+T
E→.T
T→ .T*F
T→ .F
F→ .(E)
F→ .id
} I_0

GO TO(I_0, E)

E′→E. } I_1
E→E.+T

GO TO (I_0, T)

E→T. } I_2
T→ T.*F

GO TO(I_0, F)

T→ F. I_3

GO TO(I_0, ( )

F→(.E)
E→.E+T
E→.T
T→.T*F
T→.F
F→.(E)
F→.id

} I_4

GO TO(I_0, id)

F→id . I_5

GO TO(I_1, +)

E→E+.T
T→.T*F
T→.F
F→.(E)
F→.id

} I_6

GO TO(I_2, *)

T→T*.F
F→.(E)
F→.id

} I_7

GO TO(I_4, E)

F→(E.)
E→.E+T

} I_8

GO TO(I_4, T)

E→T.
T→T.*F

} I_2

GO TO(I_4, F)

T→F. I_3

GO TO(I_4, ( )

F→(.E)
E→.E+T
E→.T
T→.T*F
T→.F
F→.(E)
F→.id

} I_4

Types of Three Address Statements:-

1. Assignment statement of the form x := y op z
2. Assignment instructions of the form x := op z
   where op is a unary operation.
3. Copy statements of the form x := y
   where, the value of y is assigned to x.
4. The Unconditional Jump GOTO L
5. Conditional Jumps such as if x relop y goto l
6. param x and call p, n for procedure calls and return y.
7. Indexed assignments of the form x := y[i] and x[i] := y
8. Address and pointer assignments, x :=&y, x := *y and *x := y

Implementations of Three Address Statements:

It has three types,

1. Quadruples
2. Triples
3. Indirect Triples

Quadruples:-

A Quadruple is a record structure with four fields, which we call op, arg1, arg2, and result. The op field contains an internal code for the operator.

For Eg, the three address statements,

x := y op z is represented by

y in arg1

z in arg2

x in result.

The quadruples for the assignment a:=b* -c + b* -c are,

	op	arg1	arg2	result
(0)	uminus	c		t_1
(1)	*	b	t_1	t_2
(2)	uminus	c		t_3
(3)	*	b	t_3	t_4
(4)	+	t_2	t_4	t_5
(5)	:=	t_5		a

Triples:-

A triple is a record structure with three fields: op, arg1, arg2. This method is used to avoid entering temporary names into the symbol table.

Ex. Triple representation of a:= b * -c + b * -c

	op	arg1	arg2
(0)	uminus	c	
(1)	*	b	(0)
(2)	uminus	c	
(3)	*	b	(2)
(4)	+	(1)	(3)
(5)	assign	a	(4)

Indirect Triples:-

Listing pointers to triples rather than listing the triples themselves are called indirect triples.

Eg. Indirect Triple Representation of a := b * -c + b * -c

	statement
(0)	(10)
(1)	(11)
(2)	(12)
(3)	(13)
(4)	(14)
(5)	(15)

	op	arg1	arg2
(10)	uminus	c	
(11)	*	b	(10)
(12)	uminus	c	
(13)	*	b	(12)
(14)	+	(11)	(13)
(15)	assign	a	(14)

BASIC BLOCKS & FLOW GRAPHS

Basic Blocks:

A block of code means a block of intermediate code with *no jumps in* except at the beginning and *no jumps out* except at the end.

A basic block is a sequence of consecutive statements in which flow of control enters at the beginning and leaves at the end without halt or possibility of branching except at the end.

Algorithm for Partition into Basic Blocks:-

Input: - A sequence of Three Address statements.

Output:- A basic blocks with each three address statement in exactly one block.

Method:-

1. We first determine the set of leaders, the first statement of basic blocks.
   The rules we use are the following,
   (i) The first statement is a leader.
   (ii) Any statement that is the target of a conditional or unconditional GOTO is a leader.
   (iii) Any statement that immediately follows a GOTO or unconditional GOTO statement is a leader.
2. For each leader, its basic block consists of the leader and all statements up to but not including the next leader or the end of the program.

Example:-

Consider the fragment of code, it computes the dot product of two vectors A and B of length 20.

```
Begin
        PROD:=0
        I:=1
        Do
          Begin
                PROD:=PROD+A[I]*B[I]
                I:=I+1
          End
        While I<=20
End
```

A list of three address statements performing the computation of above program is, (for a machine with four bytes per word)

So the three address statements of the above Pascal code is,

1. PROD:=0
2. I:=1
3. t1:=4*I
4. t2:=A[t1]
5. t3:=4*I
6. t4:=B[t3]
7. t5:=t2*t4
8. t6:=PROD+t5
9. PROD:=t6
10. t7:=I+1
11. I:=t7
12. if I<=20 GOTO (3)

The Leaders are, 1 and 3. So there are two Basic Blocks

Block 1.

```
1.PROD:=0
 2.  I:=1
```

Block 2.

```
3   t1:=4*I
4   t2:=A[t1]
5   t3:=4*I
6   t4:=B[t3]
7   t5:=t2*t4
8   t6:=PROD+t5
9   PROD:=t6
10  t7:=I+1
11  I:=t7
12. If  I<=20  GOTO (3)
```