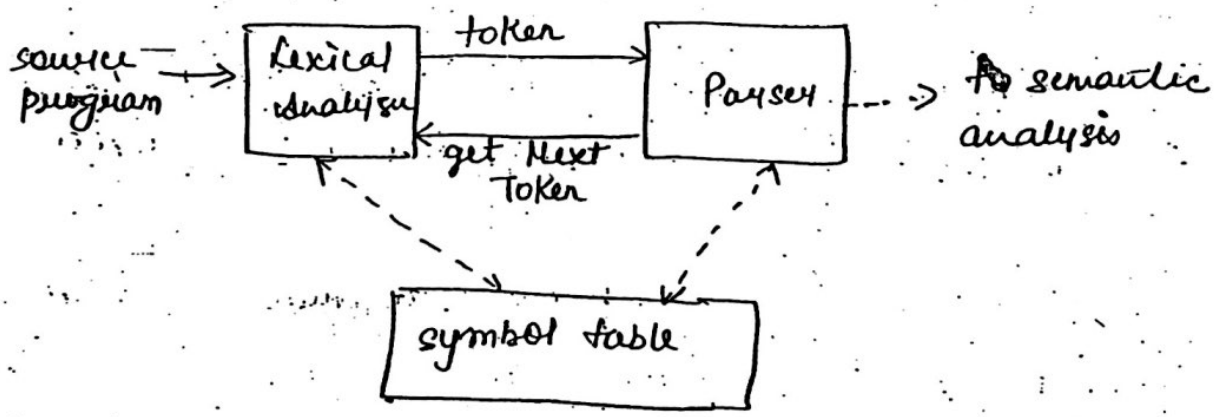
Lexical Analyzer (Scanner) -

It is the first phase of a compiler. Its main task is to read the input characters and produce a sequence of tokens that the parser (next phase) uses for syntax analysis.

Interaction b/w Lexical Analyzer and Parser

Whenever the parser need a token it calls the lexical analyzer by "get next ~~command~~ token" command, the lexical analyzer then reads characters until it can identify the next token.

Instead of analyzing the entire input string, the lexical analyzer sees enough of the I/P string to return a single token.



Interaction b/w Scanner and Parser

Tasks of a lexical analyzer

1) Recognizing reserved words (Keywords)
2) Finding identifiers - (variables)
3) Finding string and character constants
4) Ignoring comments and white spaces.
5) Counting the number of lines.
6) Reporting error messages.
7) Correlating error messages generated by the compiler with the source program.
   eg. associating line number with error messages.

Some Useful Terms,

1) Pattern - Pattern is a rule which describes a ~~token~~ set of lexemes that can represent a particular token in the source program
   for eg. identifier can be described as a letter followed by letters or digits.

2) Lexeme - Lexemes are the smallest logical units of a program. It is a sequence of characters in the source program for which a token is produced.
   eg - 10.0, +, etc.

(3) Tokens - classes of similar lexemes identified by the same token

eg: identifier, keyword etc.

following constructs are treated as tokens -

- keywords
- operators
- identifiers
- constants
- punctuation symbols etc.

---

each token is pair consisting of two values.

< token name, attribute value >

Attribute for token - more than one lexeme can match a same pattern. So the lexical analyzer must provide additional ~~redeo~~ information about that lexeme. This attribute is a pointer to the lexeme in the symbol table entry.

For operators, punctuation, keywords there is no need for an attribute value. The first component itself is sufficient to identify the lexeme.

16/9/97

Sample lexeme	token	Description of pattern
pi, count	id	letters followed by letters or digits
3.14, 286	num	any numeric constant,
"made easy"	literal	characters b/w " and "
if	if	if

ex of pattern, lexeme & token

ex: sum_two (float num1, float num2)

```
{
    float sum;
    sum = num1 + num2;
    return (sum);
}
```

Tokens	Lexeme
Keywords	float, return
identifier	sum_two, num1, num2, sum
Delimeter	), ;, (,), {, }.
Assign op	=
Addop	+

<assop>
<const>

Scanning the Input

Lexical analyser reads the source program character by character. This reading of characters from secondary storage is very costly.

So, block of data is first read into buffer and then scanned by the lexical analyser.

Two types of buffering is there

1) One buffer scheme - uses only one buffer scheme, but it has some problems, if the lexeme is longer than the length of a buffer.

2) Two buffer scheme - uses two buffers of same size that are scanned alternately.

Lexical analyser uses two pointers

lb - lexeme beginning pointer

(Lookahead pointer) fp - forward pointer - to keep track of the pos

[illegible] identified the token and the attribute of the lexeme are returned. Both pointers are then set to the beginning of the next token.

eg: sum = num1 + num2

return (sum)

sum = num1 + num2 N

2, return (sum), eof N

If fewer than n characters remain in the I/P while refilling a buffer then eof is read into the buffer after the I/P characters.

So each time fp have to make three tasks

1) checking the I buffer is full
2) checking the second buffer is full
3) check the end of I/P.

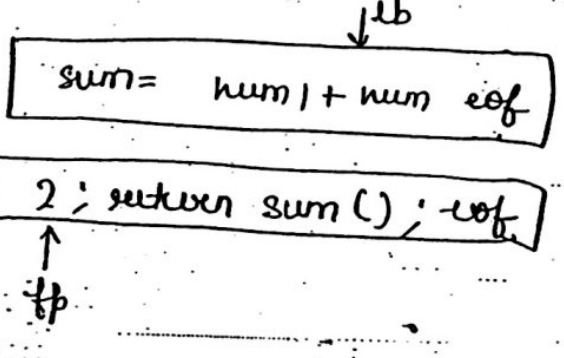
code to advance fp

```
if (fp == eob1)
    { reload buf 2
      fp = fp+1;
    }
else if (fp==eob 2)
    { reload buf 1
      fp= fp+1;
    }
else if (fp == eof)
    { terminate scanning
    }
else
    fp= fp+1;
```

//

Sentinel - These two tests can be reduced to one if we add a special character sentinel at the end of each buffer. Sentinel is not a part of the source program.

Let choose eof as sentinel



```
fp = fp+1;
if (fp == eof)
{ if (fp == eob1)
  { reload buf2;
    fp = fp+1;
  }
  elseif (fp == eob2)
    { reload buff1;
      fp = fp+1;
    }
  else
    terminate scanning;
}
else
  fp = fp+1;
```

SPECIFICATION OF TOKENS (Patterns)

Regular expressions are an important notation for specifying patterns. A regular exp. is defined by a set of strings which it matches.

Language L(r) - Set of strings generated by the regular expression r.

Alphabet is a finite set of symbols

{0,1} is a binary alphabet

String - finite sequence of symbols drawn from that alphabet.

Empty string is denoted by ϵ

ϵ - string of length 0.

ex. six is a string over an alphabet containing letters.

Terms for parts of strings

Prefix of s - obtained by removing some trailing symbols (0 or more)

eg. tab, for table, ϵ

Suffix of s - formed by deleting some leading symbols eg. able, for table, table, bla, ϵ

Substring of s - deleting a suffix and a prefix from string s.

eg. abl, ϵ, table etc for table

Proper prefix, suffix and substring of s - do not include s and ϵ

obtained

portions of s.

eg. baan is a subsequence of banana

Operations on languages

Union $L_1 \cup L_2 = \{ s \mid s \in L_1 \text{ or } s \in L_2 \}$

Intersection $L_1 \cap L_2 = \{ s \mid s \in L_1 \text{ and } s \in L_2 \}$

Concatenation $L_1 L_2 = \{ sp \mid s \in L_1 \text{ and } p \in L_2 \}$

Exponentiation $L^i = L L^{i-1}$ for $i > 1$

$L^0 = \epsilon$

Kleene closure $L^* = \bigcup_{i=0}^{\infty} L^i$

(zero or more concatenations of L)

Positive closure $L^+ = \bigcup_{i=1}^{\infty} L^i$

(one or more concatenations of L

Let $L = \{ A, \ldots, Z, a, \ldots, z \}$

$D = \{ 0, 1, \ldots, 9 \}$

then $(L \cup D)^*$ - is the set of all strings of letters and digits

and $L(L \cup D)^*$ - is set of all strings of letters and digits beginning with a letter

(1) - denotes all strings of length two with one letter followed by one digit

L^4 - set of strings having 4 length.

LUD - set of letters and digits with 62 strings of length one.

D^+ - set of all strings of one or more digits

Regular Expressions

A regular expression is defined by the set of strings it matches and is built up of basic regular expressions using a set of rules.

A regular expression is defined by the following rules.

- ϵ is a regular expression
- if symbol 'a' is in alphabet then a is a regular expression.
- If r and s are regular expressions over the alphabet then
  - a) r|s is a regular expression
  - b) rs is a regular expression.
- If r is a regular expression
  - a) r^* is a regular expression.
  - b) (r) is a regular expression

Language of identifiers is defined by regular expression

$l(l|d)^*$

operator precedence $*$

concatenation

|

eg $(a)|((b)^*(c)) \rightarrow a|b^*c$

Algebraic laws for Regular Expressions

some laws	description
1) $r\|s = s\|r$	is commutative
2) $r\|(s\|t) = (r\|s)\|t$	is associative.
3) $r(st) = (rs)t$	concatenation is associative
4) $r(s\|t) = rs\|rt$	concatenation distributes over |
5) $\epsilon r = r\epsilon = r$	ϵ is the identity for concatenation
6) $r^* = (r\|\epsilon)^*$	ϵ is guaranteed in closure
7) $r^{**} = r^*$	$*$ is idempotent.

eg :- (a|b)(a|b) denotes {aa, ab, ba, bb}

the language of the strings of length two over the alphabet Σ.

2) a^* denotes {ε, a, aa, aaa, ...}

all strings of zero or more a's

3) $(a|b)^*$ denotes {ε, a, b, aa, ab, ba, bb, aaa...}

all strings of a and b.

4) $a|a^*b$ denotes the language ——

{a, b, ab, aab, aaab ...}

string a and all strings consisting of zero or more a's and ending in b.

5) $(a|b)^+ c (a|b)^*$ - set of all strings over this alphabet that contains exactly one c.

some strings matched by above regular exp are

c, acaba, caaab, bbaabb, aabbc

Regular Definitions - It is good to give name to a regular expression so that we can refer it when we want to use it.

A regular definition is a sequence of definitions of the form -

$d_1 \rightarrow r_1$

$d_2 \rightarrow r_2$

$d_3 \rightarrow r_3$

$\vdots$

$d_n \rightarrow r_n$

where each r_i is a regular exp defined over the alphabet $\Sigma \cup \{d_1, d_2 \ldots d_{n-1}\}$, d_i is name given to that regular exp.

eg. digit = 0|1|2|...|9

regular definition for defining the digit

natural $\Rightarrow$ digit . digit*

natural $\Rightarrow$ digit+

eg. letter_ $\rightarrow$ A|B|...z|a|b...z|_

digit $\rightarrow$ 0|1|2|...9

id $\rightarrow$ letter_ (letter_ | digit)*

Design of Lexical Analyzer :

We all know that flow chart plays a main role in ~~the~~ ~~designing~~ writing of program.

Transition diagram is a kind of flow chart for the lexical analyzer.

In the construction of lexical analyzer first the specification of pattern is done. then the transition diagram for each token is drawn.

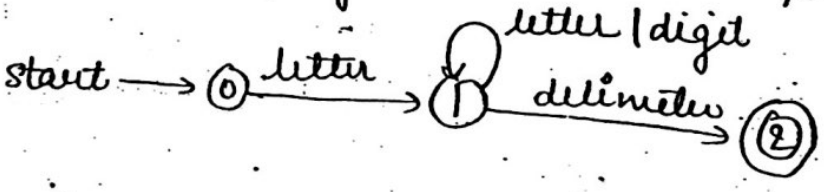
Transition Diagram – A transition diagram is a directed graph with state drawn as circles (nodes) and edges represent transitions on Input symbols. Each transition diagram has a start state and a final state.

A transition diagram is used to keep track of information about characters that are seen as the forward pointer fp scans the input.

Ex- for identifier

the pattern in form of Regular Expression is → id → letter(letter | digit)*

the transition diagram for above pattern is



Transition diagram

code for state 0

```
c = getchar();
if letter(c) then goto state 1
else fail();
```

code for state 1

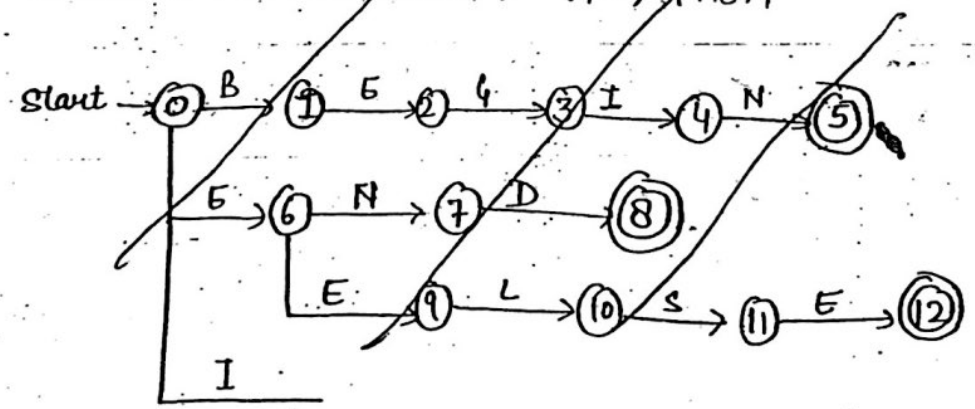
```
c = getchar();
if letter(c) or digit(c) then goto state 1
else if delimiter(c) then goto state 2
else fail();
```

code for state 2

```
Retract();
return (id, install);
```

Now let we have to draw the transition diagram for keywords: BEGIN, END, ELSE, IF, THEN

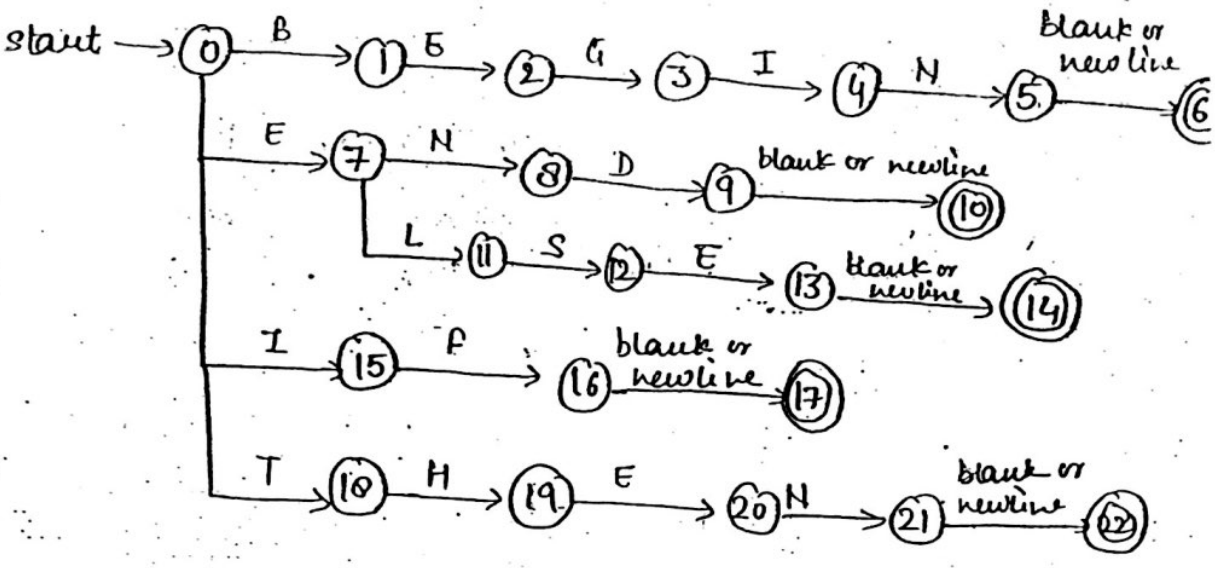
Transition Diagram ⇒



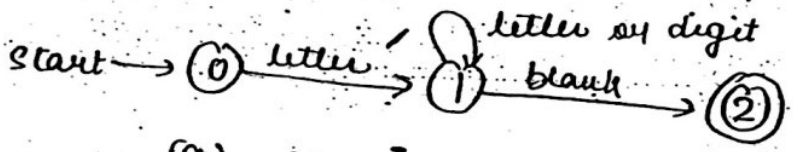
P.T.O

Now let we have to draw the transition diagram for keywords BEGIN, END, ELSE, IF, THEN

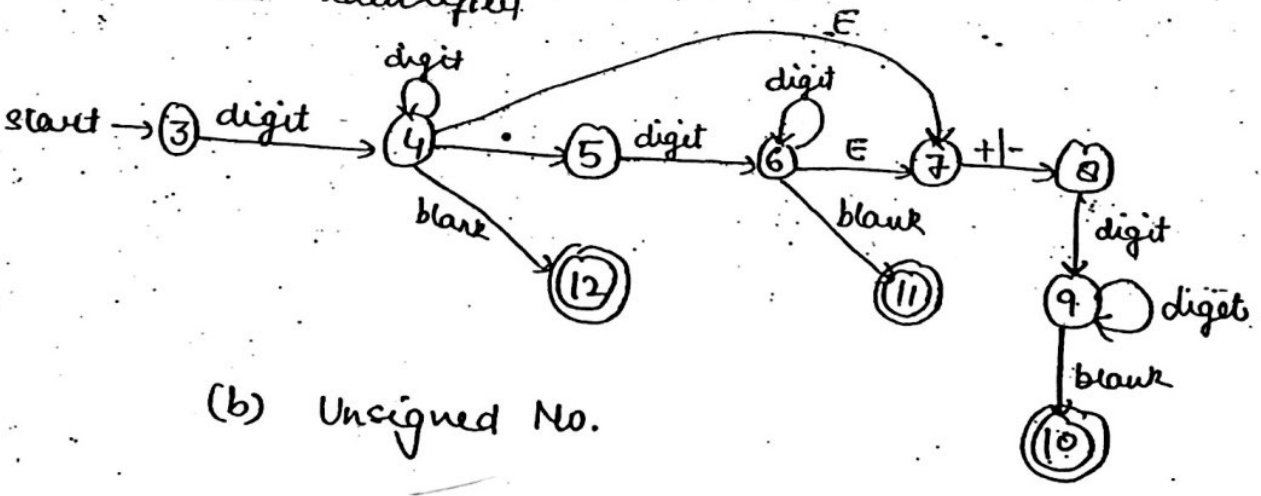
Transition diagram →



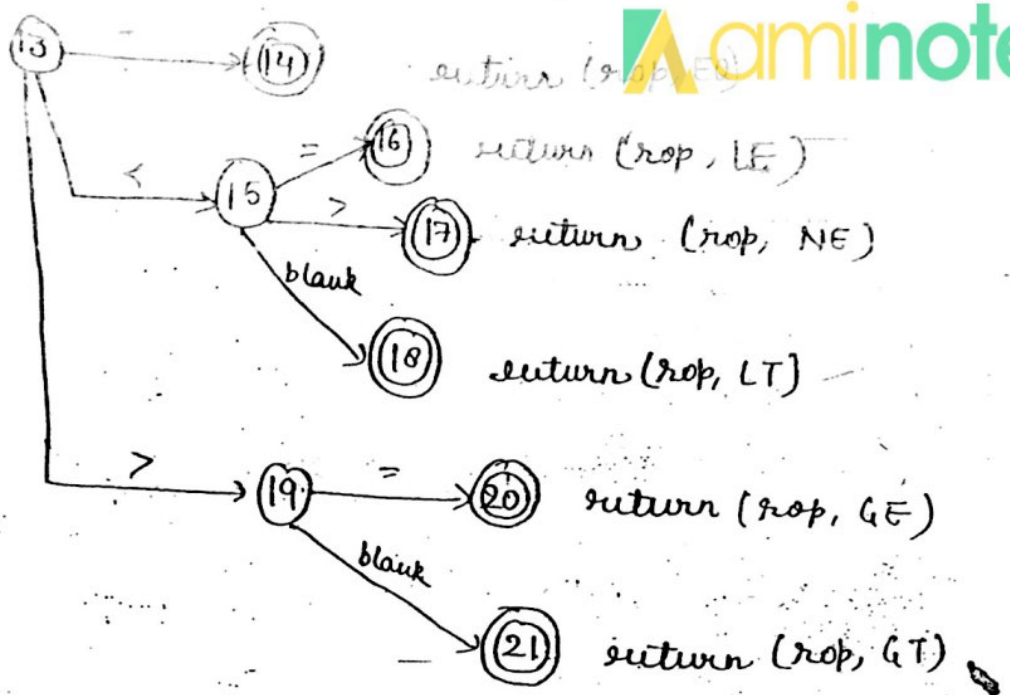
Now we have a transition diagram for identifier, unsigned number and relational operator as follows-



(a) identifier



(b) Unsigned No.



(c) transition diagram for relational operators

Converting above transition diagram into code

```
function fail()
{ fp = lb;
  case start of
  { 0: start = 3;
    3: start = 13;
    13: recover();
    default: error;
  }
}
```

```
function nexttoken()
{  state=0 ; start=0;
   while (1)
   { case state of
     0: {  lb= fp;
           c= nextchar();
           if letter(c) state=1;
           else fail();
           break;
        }
     1: {  c= nextchar();
           if (letter(c) or digit(c)) state = 1
           else state =2;
           break;
        }
     2: {  fp= fp-1;
           return (id, input[lb, fp])
           break;
        }
     /* ... write cases for 3 to 12 state similarly.. */
     13: {  c= nextchar();
           if (c=='=') state=14;
           elseif (c=='<') state=15;
           else if (c=='>') state =19;
        }
```

```
        break;
    }
15: { c = nextchar();
    if (c == '=') state = 16;
    else if (c == '>') state = 17;
    else state = 18;
    break;
    }
16: { return (rop, LE)
    break;
    }
17: { return (rop, NE)
    break;
    }
18: { fp = fp-1;
    return (rop, LT);
    break;
    }
/* ... write cases for state 19 to 21 similarly ... */
```

Finite automaton - Lexical analysis is based on a simple recognizer called a finite automaton (which is the part of the lexical analyzer that identifies the token)

Finite automaton are recognizers (or proposal), they simply say "yes" or "no" about each possible I/P string.

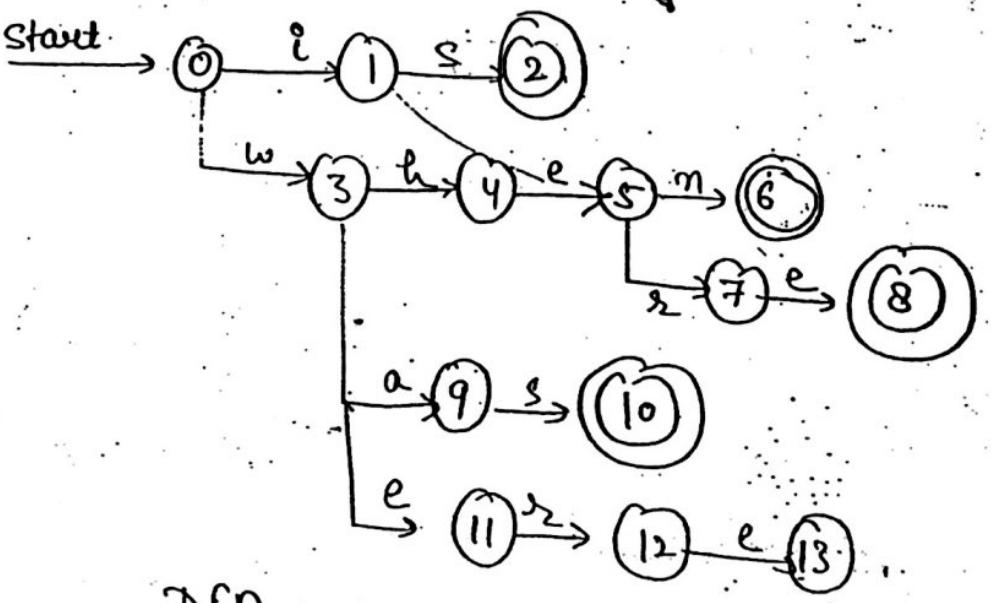
The automaton takes a sequence of inputs and performs a sequence of actions for the I/P string.

Finite automaton come in two flavours

a) NFA - non deterministic finite automaton.
Any I/P alphabet can have several edges out of the same state.

b) DFA deterministic finite automaton -
For each state and for each input alphabet there can be only edge leaving that state.

ex.



DfA

I/P alphabet is { 0, 1, 2, 3, 4, ... 13 }

Set of state = { 0, 1, 2, 3, 4, ... 13 }

Accepting state = { 2, 6, 8, 10, 13 }

~~Start~~

DFA is defined by 5 tuple

S_0 → start state

F → final state

Σ - I/P alphabet

So - set of states

Transition function → $S \times \Sigma \rightarrow S$

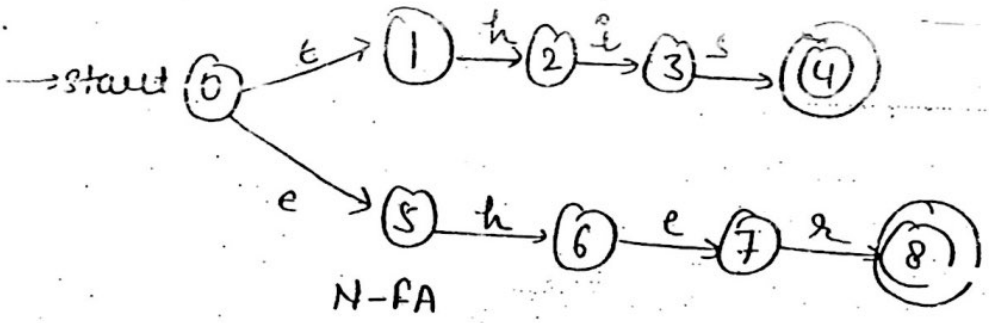
In DFA the next state can be determined by knowing the current state and the current I/P symbol.

In a DFA there are no two edges leaving a given state have the same level.

Transition table

current state	I/P alphabe
	O/P state

NFA eg: automaton that recognizes the strings this | that



N-FA

In this NFA two outgoing edges have the same label with ε.

NFA can have more states than the ~~NFA~~ equivalent DFA and it is difficult to decide which path to follow in the NFA if the transition table shows the transition to more than one state.

and also it is difficult to construct a DFA directly from a regular expression.

So lex uses the two step process

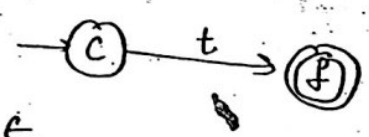
1) construct a NFA from the regular exp
2) convert the NFA to a DFA.

This construction process is guided by ... known as Thompson's construction

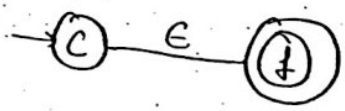
Four basic NFA structures are as follows -
c - current state , f - final state

1) Let t is symbol in Σ alphabet

for regular expression t , NFA is

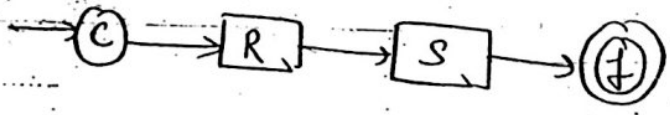


for ∈

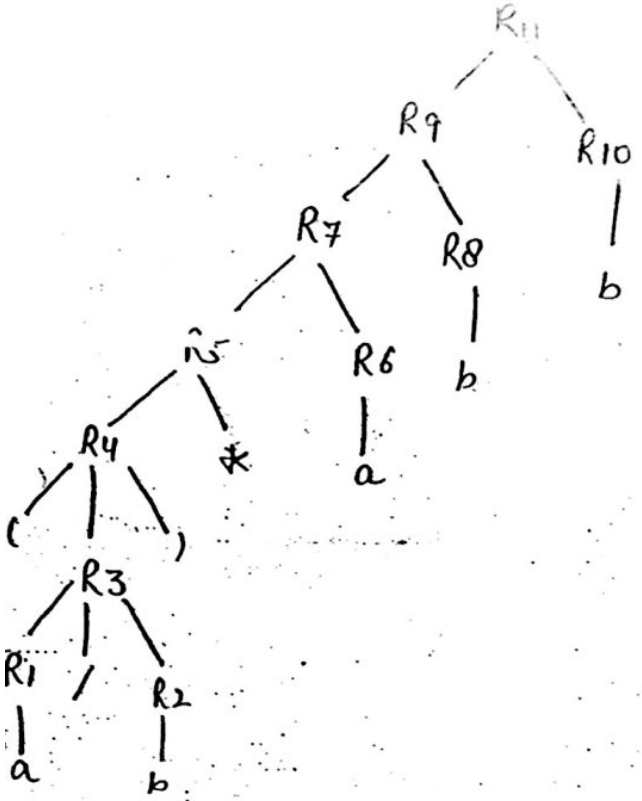


2) Let NFA for regular exp R is

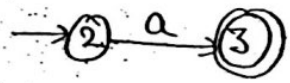
then concatenation (R).(S) is represented by



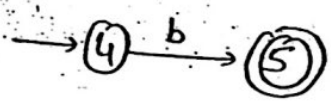
here the current state of R becomes the current state of S NFA and final state of S becomes the final state of NFA.



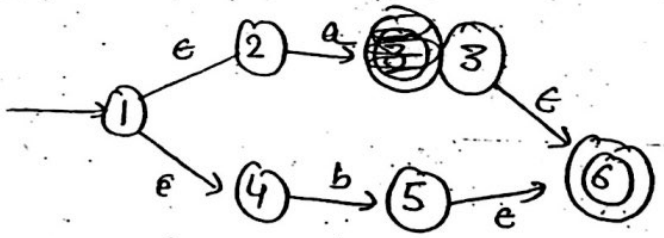
R1



R2

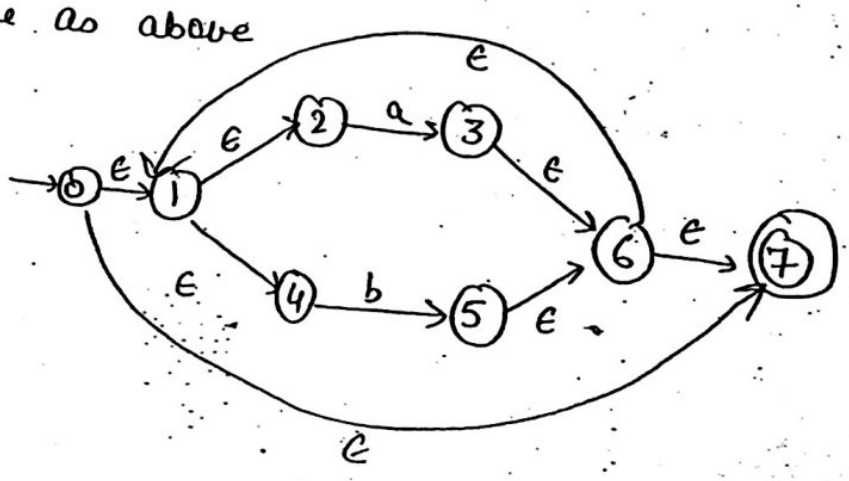


R3 - R1 | R2

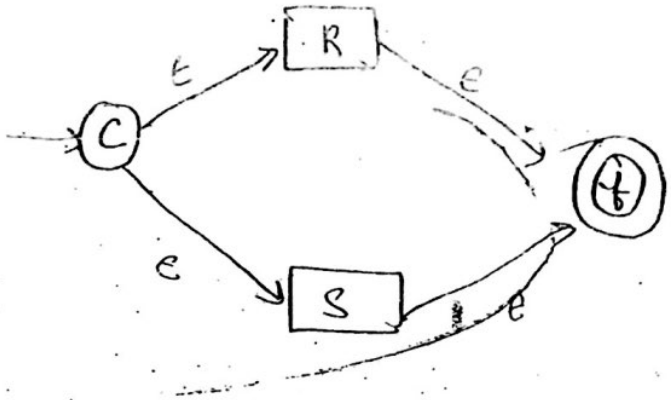


R4 → (R3) same as above

R5 = (R4)*

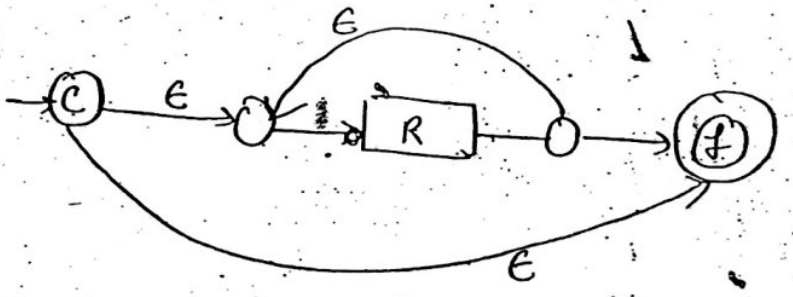


3) Alternation (R) | (S)



here the current and final states of R and S are not the current and final states of NFA. These are new current and final states.

4) NFA for regular expression R+

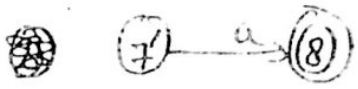


eg. Construction of NFA for regular exp.

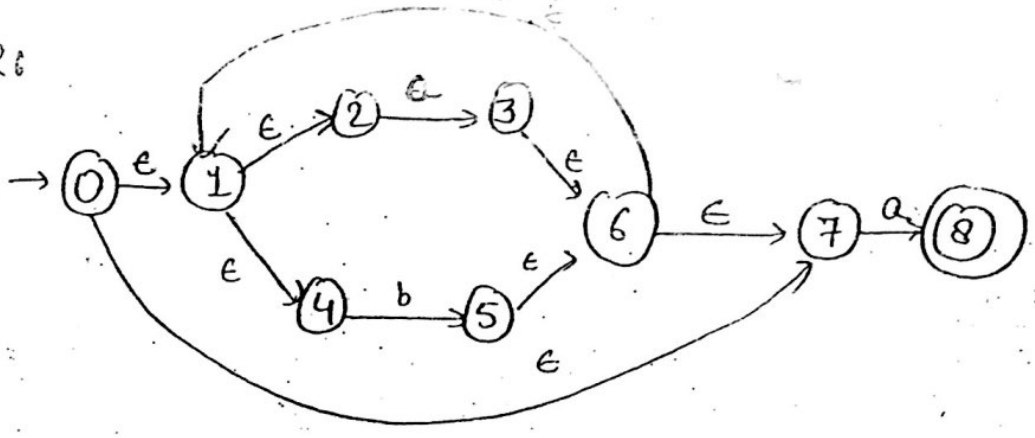
(a/b)* abb

Let decompose this R into its primitive components.

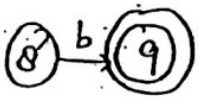
R6



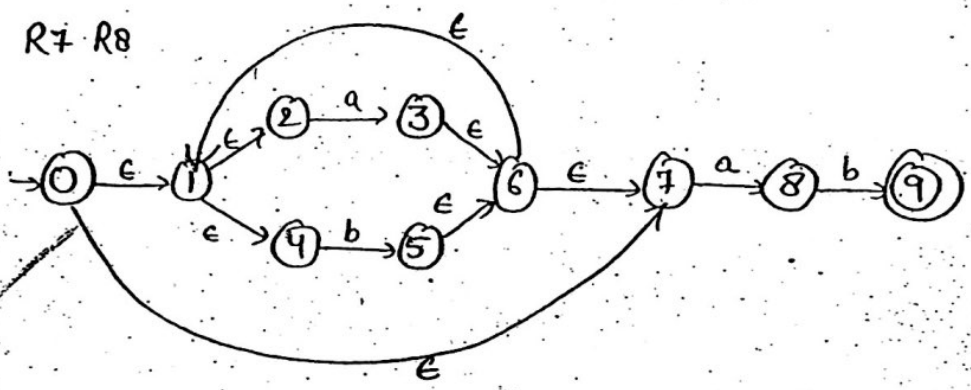
R7 = R5 R6



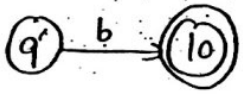
R8 =



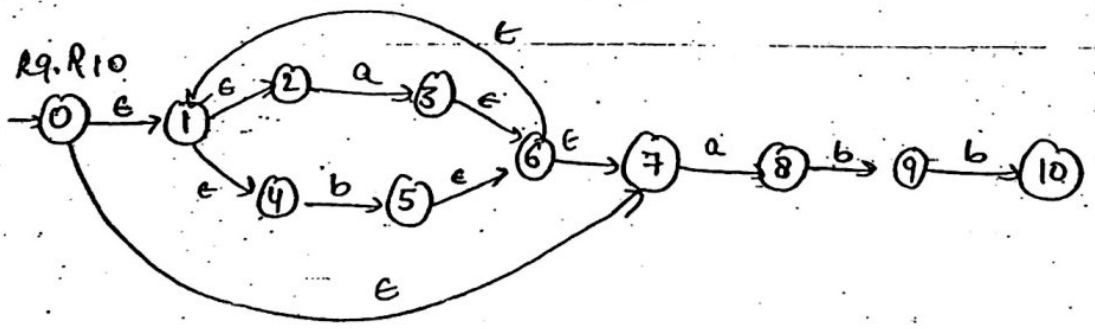
R9 = R7 R8



R10 =



R = R9.R10



representing exp (a/b)*abb

To obtain a DFA from a NFA we have to find out the states of an NFA that can be reached for a particular prefix and put together these states to form one state of a DFA.

Steps for construction of a DFA from NFA called subset construction.

Let S be a ~~set of state eqv~~ DFA state equivalent to set of states $\{s_1, \ldots s_i\}$ of NFA and T another DFA state eq. to set of states $\{t_1, t_2 \ldots t_j\}$ of NFA

If $\{s_i, a, t_j\}$ exists in the NFA, then (S, a, T) will exists in the DFA.

means for the IP symbol a a state transition from S to T will occur if there exists a state transition from s_i to t_j labelled a.

now define ϵ-closure to be the set of NFA states by following rules

1) s is added to ϵ-closure(s).

2) If t is in ϵ-closure(s), and there is an edge labelled ϵ from t to u then u is added

to ε-closure(s) Repeat it until no more states can be added.

ex. see NFA for $(a|b)^*abb$

→ The initial state for equivalent DFA is

initial state = ε-closure(0)

A = {0, 1, 2, 4, 7}

→ now find out those states from A who have transitions on a these are 2 and 7 and have such transitions to 3 and 8.

so ε-closure({3, 8}) = {3, 6, 7, 1, 2, 4, 8}

B = {1, 2, 3, 4, 6, 7, 8}

→ now find out those states from A who have transitions on b, these are 4 and transition to 5

ε-closure({5}) = 5, 6, 7, 1, 2, 4

C = {1, 2, 4, 5, 6, 7}

→ now find the states from B having transitions on a 2→3 and 7→8

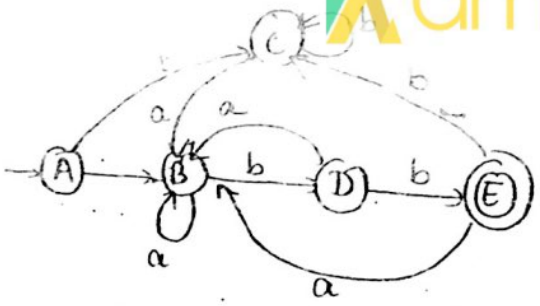
ε closure({3, 8}) = {1, 2, 3, 4, 6, 7, 8}
= B

for B b 4→5, 8→9

ε closure({5, 9}) = 5, 6, 7, 1, 2, 4, 9

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

State	a	b
→ A	B	C
B	B	D
C	B	C
D	B	E
(E)	B	C



Transition table for $(a/b)^*$ abb

Minimizing the no. of states of DFA

1) Construct a partition π of the set of states into two groups F - final states
S-F - non final states

eg:
two groups (ABCD), (E)

2) Now again split them

(E) can not be partitioned again so we place (E) in π_{new}

now partitioned - (ABCD) on the basis of I/P symbols

Let on input a each of these states go to B but on b A B C go to members of group ABCD while D goes to a member of another group E.

So new value of π is (ABC) (D) (E)

(ABC) on I/P a → no split
on I/P b (AC) (B) (D) (E)

(AC) on I/P a no split
on I/P b no split

so final partition is (AC) (B) (D) (E)

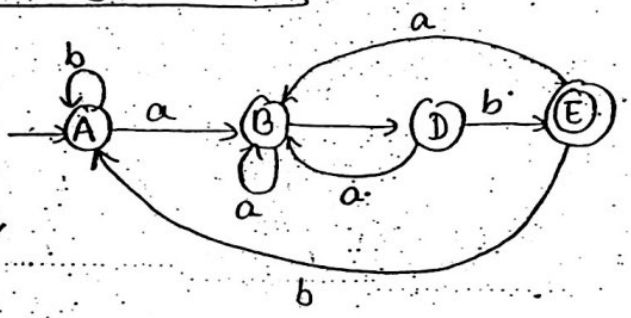
— now supplies (AC) by A

then we get

	I/P	
States	a	b
A	B	A
B	B	D
D	B	E
E	B	A

8. Reduced automaton

minimized DFA is



minimized DFA accepting (a/b)* abb

Automatic creation of lexical analyzer

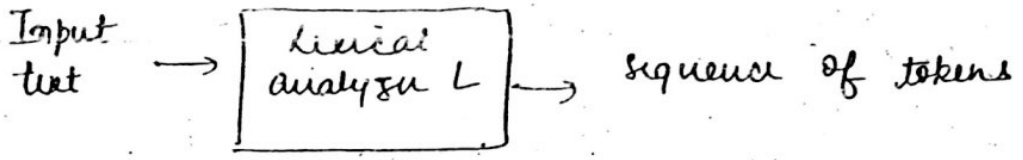
Lexical analyzer generator is used to generate lexical analyzer automatically.

Lex – is a unix tool which can automatically generate a lexical analyzer.

This LEX is a lexical analyzer generator which takes as Input the precise specification of tokens of the language, and a specification of the action to be performed on identification of each token. As an output it generates a lexical analyzer.

I/P of LEX – Input of Lex compiler is a source program which is a specification of a lexical analyser, consisting of a set of regular expressions together with an action for each regular expression. This action is a piece of code which is to be executed whenever a token specified by the corresponding regular expression is recognized. This action will pass an indication of the token found to the parser, with the side effects such as making an entry in symbol table.

O/P of LEX – is Lexical analyzer program constructed from the LEX source specification.



LEX source program consists of two parts

(a) Auxilliary Definitions - these are the statements of the form -

$D_1 \rightarrow R_1$
$D_2 \rightarrow R_2$
:
$D_n \rightarrow R_n$

where each D_i is a distinct name and each R_i is a regular expression whose symbols are chosen from $\Sigma \cup \{D_1, D_2 \ldots D_{i-1}\}$.

The four identified sequence of auxillary definitions are.

letter $\rightarrow$ A | B | ... | z
digit $\rightarrow$ 0 | 1 | ... | 9

(b) Translation Rules these are the statements of the form

P_1	$\{A_1\}$
P_2	$\{A_2\}$
:	:
P_n	$\{A_m\}$

each P_i is a regular expression called Pattern over the alphabet Σ and the auxillary definition names

Pattern describe the form of token each P_i is a pattern describing the action that the lexical analyzer should take when P_i is found.

This action can be a returning of token to parser or can be a making of entry into the symbol-table.

Ex-

Let we have following tokens

- begin
- end
- if
- then
- else
- identifier
- ~~digit~~ constants
- <
- <=
- =
- <>
- >
- >=

We have to generate the lex source program for these tokens-

Auxiliary definitions

letter → A|B...|Z

digit → 0|1|...|9

Translation Rules

BEGIN	{return (BEGIN)}
END	{return (END)}
IF	{return (IF)}
THEN	{return (THEN)}
ELSE	{return (ELSE)}
letter (letter|digit)*	{LEXVAL = install (); return (ID);}
digit+	{LEXVAL = install (); return (NUM);}
<	{LEXVAL = LT, return (ROP);}
<=	{LEXVAL = LE, return (ROP);}
=	{LEXVAL = EQ, return (ROP);}
<>	{LEXVAL = NE, return (ROP);}
>	{LEXVAL = GT, return (ROP);}
>=	{LEXVAL = GE, return (ROP);}

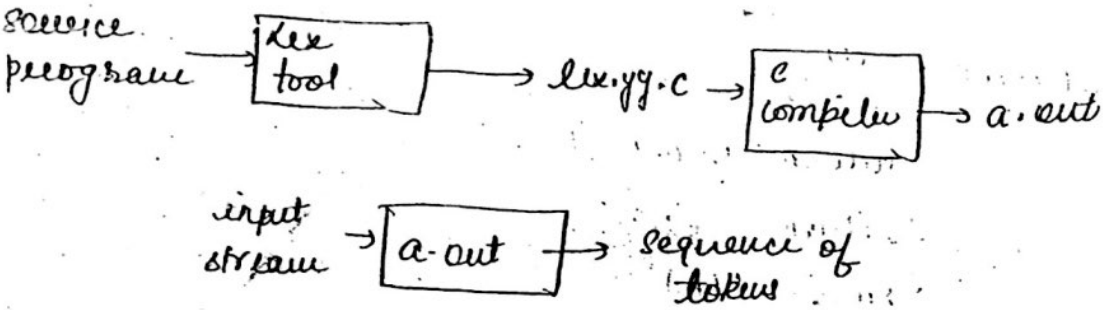
LEX Program

Conflict Resolution in LEX

If several prefixes of the input match one or more patterns then

1) Always prefer a longer prefix to a shorter prefix. eg ⇐ if <= encountered then token can be < or <= but longer one <= will be prefered.
2) If the longest possible prefix matches two or more patterns, prefer the pattern listed first in the LEX program. eg. BEGIN, if encountered will be treated as keyword not identifier

Automatic creation of lexical analyser using Lex



Compiler Construction
CS6

SYNTAX ANALYSIS

In previous ~~step~~ phase lexical analysis we had learnt about the specification of patterns using regular expressions and the construction of lexical analyser to recognize tokens using automaton.

Now the syntax analysis phase is the next phase after lexical analysis in compiler design.

Syntax Analyser (Parser) takes a token stream produced by lexical analyzer as input and create a tree like intermediate representation (Parse tree) as O/P that depicts the grammatical structure of token stream.

We all know that every programming language has some precise rules to prescribe the syntactic structure of well formed program.

Eg. In C, a program is made up of functions, a function out of declarations and statements, a statement out of expressions and so on...

This syntax of programming language constructs can be specified by CFG (context free grammar) which is also called BNF (Backus-Naur Form) description.

Grammar is used to specify the syntax of a language. This grammar has following advantages -

1) Grammar gives a precise, easy to understand syntactic specification of programming language.

2) An efficient parser can be constructed automatically from a properly designed grammar.

3) A grammar imparts a structure to a program that is useful for its translation into object code and for the detection of errors.

4) A grammar allows a language to be evolved or developed iteratively by adding new constructs to perform new tasks.

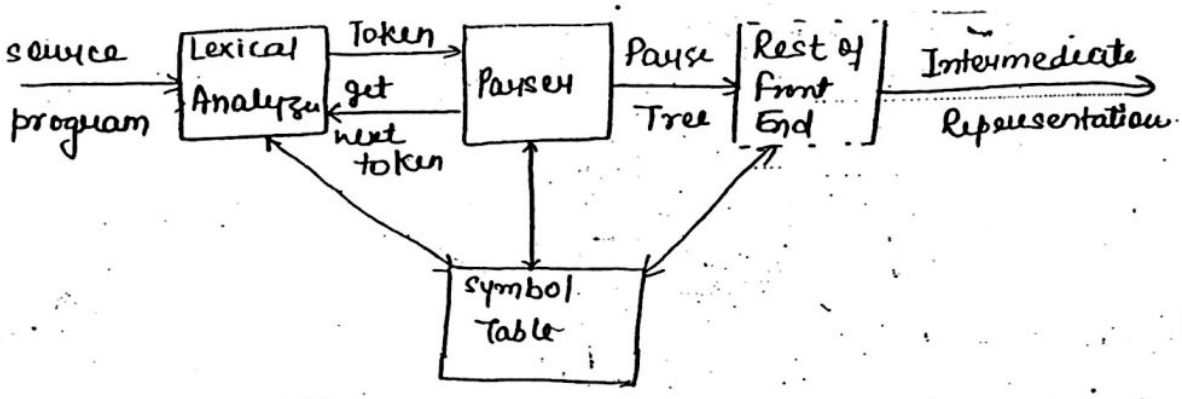
ROLE OF PARSER

→ It obtains a string of tokens from the lexical analyser.

→ It groups the tokens to identify larger structures in the program. This process is done to verify that the string can be generated by the grammar for the source language.

→ It should report any syntax error in the program.

→ It should also recover from the errors so that it can continue to process the rest of the I/P.



CONTEXT FREE GRAMMAR

The syntax of a programming language is specified by a notation called Context free grammar (CFG) which is also called as BNF (Backus Naur form).

A CFG involves four quantities

→ Terminals - these are the basic symbols of which strings in the language are composed. Tokens can be considered as terminals symbols.

→ Non terminals - these are the variables that denote a set of strings. They do not exist in the source program. They only help in defining the language generated by the grammar.

→ Start symbol - It is a non terminal and the set of strings denoted by this start symbol is the language defined by the Grammar. Means it denotes that language in which we are truly interested.

→ Production Rules - It defines the way in which the terminals and non terminals can be combined to form strings

Each production consists of

a) a non terminal on left hand side of production

b) A symbol →

c) Zero or more terminals and non terminals on the right hand side

we may define a CFG as follows

$G = (T, N, S, P)$

where T(G) → non empty set of terminals

N(G) → " " " non terminals

S(G) → single non terminal $\in N(G)$

P(G) → set of production rules.

like $A \to \alpha$

where $A \in N(G)$

& $\alpha \in (N(G) \cup T(G))^*$

Derivation produces a new string from a given string. So derivation can be used repeatedly to obtain a new string from a given string.

let $\alpha_1 \to \alpha_2 \to \alpha_3 \to \alpha_4 \dots \to \alpha_n$

then we can say

$\alpha_1 \overset{*}{\Rightarrow} \alpha_n$ means α_1 derives α_n and derivation comprise of zero or more steps.

If $\alpha \overset{*}{\Rightarrow} \beta$ and $\beta \overset{*}{\Rightarrow} \gamma$

then we can say that

$\alpha \overset{*}{\Rightarrow} \gamma$

$\overset{+}{\Rightarrow}$ is used to say that the derivation comprises of one or more steps.

---

let $S \overset{*}{\Rightarrow} \alpha$ where $\alpha \in (N(G) \cup T(G))^*$

means α may contains non terminals, then α is called sentential form of grammer.

Ex: we have a grammar for expression containing +, -, /, ↑, *, id

as follows ..

$$E \rightarrow E \text{ op } E \mid (E) \mid -E \mid id$$

$$op \rightarrow + \mid - \mid * \mid \uparrow \mid /$$

Now we have to derive a string (id + id) from the above grammar -

So
$$E \rightarrow (\underline{E})$$
$$\rightarrow (\underline{E} \text{ op } E)$$
$$\rightarrow (id \underline{op} E)$$
$$\rightarrow (id + \underline{E})$$
$$\rightarrow (id + id)$$

LEFT most and RIGHT most derivation

Each derivation step needs to choose a nonterminal to replace and a production to apply.

If at each step we replace the left most non terminal, derivation is known as left most derivation.

& If at each step we replace the right most non terminal, derivation is known as Right most derivation.

Ex-1 The grammar that describe the syntax of an expression consisting of +, - and digits -

$P(G) \rightarrow \{$ exp → exp + digit
exp → exp - digit
exp → digit
digit → 0|1|2...|9 $\}$

here $T(G) = \{0,1,...9,+,-\}$
$N(G) \Rightarrow \{exp, digit\}$
$S(G_1) = \{exp\}$

Ex-2 Now expression has following terminal symbols

id, +, -, *, /, (,) ;

then the grammar is

exp → exp + term
exp → exp - term
exp → term
term → term * factor
term → term / factor
term → factor
factor → (exp)
factor → id

grammar for +, −, *, /, ↑, ( ), id can also be modified to

Exp → Exp op Exp

Exp → (Exp)

Exp → − Exp

Exp → id

op → + | − | * | ↑ | /

by using shorthands it can be written as

Exp → Exp op Exp. | (Exp) | − Exp | id

op → + | − | * | ↑ | /

DERIVATIONS AND PARSE TREES

DERIVATION - derivation provides a mean for generating the sentences of a language. Derivation refers to replacing an instance of a non terminal in a given string's non terminal by R.H.S of the production rule whose L.H.S contains the non terminal to be replaced.

eg. $A \to \gamma$ is a production rule

then $\alpha A \beta \Rightarrow \alpha \gamma \beta$

means $\alpha A \beta$ derives $\alpha \gamma \beta$ in a derivation of one step.

eg: derivation of string 'id + (id * id)' by the same grammar –

Left most derivation

$E \rightarrow \underline{E}\ op\ E$
$\rightarrow id\ \underline{op}\ E$
$\rightarrow id + \underline{E}$
$\rightarrow id + (\underline{E})$
$\rightarrow id + (\underline{E}\ op\ E)$
$\rightarrow id + (id\ \underline{op}\ E)$
$\rightarrow id + (id * \underline{E})$
$\rightarrow id + (id * id)$

Right most derivation

$E \rightarrow E\ op\ \underline{E}$
$\rightarrow E\ op\ (\underline{E})$
$\rightarrow E\ op\ (E\ op\ \underline{E})$
$\rightarrow E\ op\ (E\ \underline{op}\ id)$
$\rightarrow E\ op\ (\underline{E} * id)$
$\rightarrow E\ \underline{op}\ (id * id)$
$\rightarrow E + (id * id)$
$\rightarrow id + (id * id)$

Parse Trees A parse tree is a graphical representation of a derivation that filters out the order in which productions are applied to replace non terminals.

Properties of parse tree

→ Root contains the start symbol.
→ Each interior node contains a non terminal.
→ If there is a production rule $A \rightarrow Y_1 Y_2 Y_3$ and there is a interior node containing a non terminal A then children of that node contains $Y_1 Y_2 Y_3$ from left to right.

Each leaf node contains a terminal symbol or an ε (Epsilon)

Ex- We have to generate a string id + id * id from the previous grammar -

$E \rightarrow \underline{E}\ op\ E$

$\rightarrow \underline{E}\ op\ E\ op\ E$

$\rightarrow id\ op\ E\ op\ E$

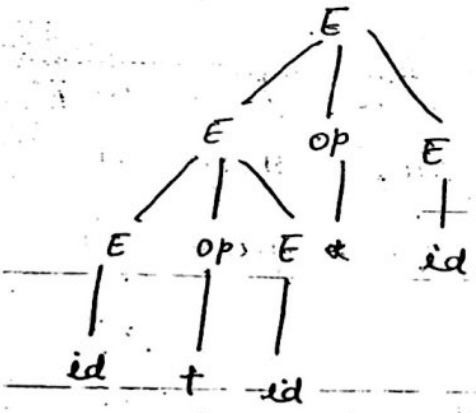
$\rightarrow id + \underline{E}\ op\ E$

$\rightarrow id + id\ op\ E$

$\rightarrow id + id * \underline{E}$

$\rightarrow id + id * id$

Parse tree for the above derivation is as



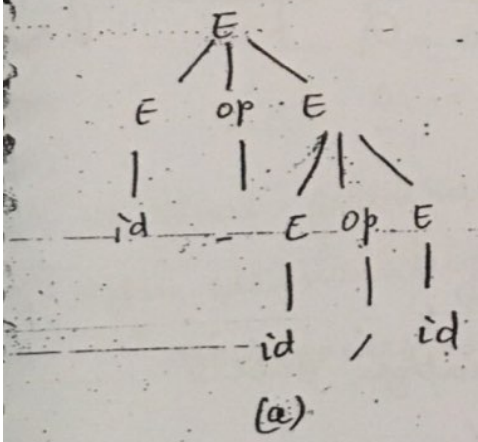
Parse tree for string id + id * id

Ambiguity - A grammar is ambiguous if it produces more than one parse tree for some sentence. Mostly a grammar is ambiguous when the same non terminal appears twice on a R.H.S of the production.

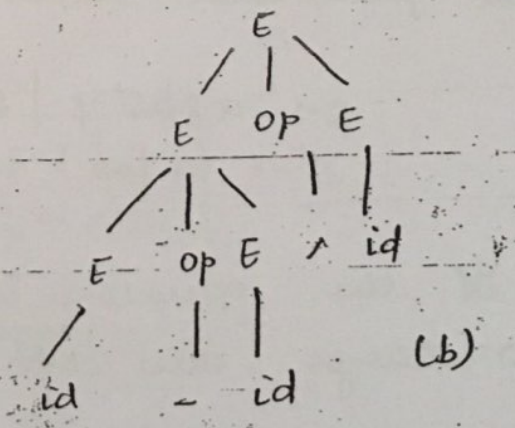
For eg. string id - id/id has two different left most derivations.

$E \to E$ op E
$\to$ id op E
$\to$ id - E
$\to$ id - E op E
$\to$ id - id op E
$\to$ id - id / E
$\to$ id - id / id

$E \to E$ op E
$\to E$ op E op E
$\to$ id op E op E
$\to$ id - E op E
$\to$ id - id op E
$\to$ id - id / E
$\to$ id - id / id



(a)



(b)

Ambiguous grammars are not good for parsing because we can not determine which production should be preferred for the expansion.

So, grammar should be rewritten to be unambiguous.

Rules for disambiguating a grammar

Generally productions are ambiguous when they have more than one occurrence of a given non terminal on their right hand side.

Rules :-

- Check the precedence of the operators involved in the productions.
- Different precedence operators treated differently for removing ambiguity
- First remove the ambiguity for minimum precedence & then so on...
- ~~If the operator is left~~

If we have a production rule like

$$S \to \alpha S B S \gamma \mid \alpha_1 \mid \alpha_2 \ldots \alpha_n$$

then ambiguity can be removed by rewriting the production rule as

$$S \to \alpha S \beta S' \gamma \mid S'$$
$$S' \to \alpha_1 \mid \alpha_2 \ldots \alpha_n$$

If the operator is left associative then change the ~~left~~ right most symbol

eg. $E \to E \uparrow E \mid id$

$$\begin{cases} E \to E' \uparrow E \mid E' \\ E' \to id \end{cases}$$

and if the operator is right associative then change the left most symbol

eg. $E \to E * E \mid id \to \begin{cases} E \to E * E' \mid E' \\ E' \to id \end{cases}$

eg.

$$E \to E+E \mid E-E \mid E/E \mid E*E \mid E \uparrow E \mid (E) \mid id$$

here the precedence of operator is as follows:

+, -	lowest
*, /	
↑	highest

and +, -, *, / are left associative and ↑ is right associative.

so first remove the ambiguity of + & - grammar

$$E \to E+T \mid E-T \mid T$$
$$T \to E/E \mid E*E \mid E \uparrow E \mid (E) \mid id$$
$$T \to T/F \mid T*T \mid T \uparrow T \mid (E) \mid id$$
$$T \to T/f \mid T*f \mid f$$
$$f \to T \uparrow T \mid (E) \mid id$$
$$f \to f \uparrow f \mid (E) \mid id$$
$$f \to G \uparrow f \mid G$$
$$G \to (E) \mid id$$

so final grammar is after eliminating ambiguity →

$$E \to E+T \mid E-T \mid T$$
$$T \to T/f \mid T*f \mid f$$
$$f \to G \uparrow f \mid G$$
$$G \to (E) \mid id$$

$i \to i \uparrow \pi$

Eliminating left recursion

Syntactic structures of a programming languages are expressed using recursive rules. These recursions can have left recursion & right recursions.

Left recursive grammars are not suitable for top down parsers.

Left recursive grammar :- A grammar is left recursive if the first symbol in the right hand side of a rule is the same non terminal as that in the left hand side

eg. $S \to S\alpha$

Top down parsing methods can not handle left recursive grammars, so the elimination of left recursion is needed.

Rules : ex if we have a production like

$S \to S\alpha \mid \beta$

then $S \to \beta S'$

$S' \to \alpha S' \mid \epsilon$

In General

$S \to S\alpha_1 \mid S\alpha_2 \ldots S\alpha_n \mid \beta_1 \mid \beta_2 \ldots$

replaced by

$S \to \beta_1 S' \mid \beta_2 S' \ldots \beta_n S'$

$S' \to \alpha_1 S' \mid \alpha_2 S' \mid \ldots \alpha_n S$

eg. $E \to E+T \mid T$

$T \to T * F \mid F$

$F \to (E) \mid id$

then $E \to TE'$

$E' \to +TE' \mid \epsilon$

$T \to FT'$

$T' \to *FT' \mid \epsilon$

$F \to (E) \mid id$

But if the elimination of left recursion involve indirectly recursive grammar. eg. –

$S \to A$
$A \to Sd \mid b$

→ $A \to Ad \mid b$
→ $S \to Sd \mid b$

written as

$S \to Sd \mid b$
$S \to bS'$
$S \to dS' \mid \epsilon$

$A \to A\alpha \mid \beta$
$A \to \beta A'$
$A' \to \alpha A' \mid \epsilon$

eg-2

$A \to Ba \mid b$
$B \to bdB' \mid B'$

$A \to Ba \mid b$
$B \to Bc \mid Ad \mid \epsilon$

↓

$A \to Ba \mid b$
$B \to Bc \mid Bad \mid bd \mid \epsilon$

$B \to \underline{Bc} \mid \underline{Bad} \mid bd \mid \epsilon$

$B \to bdA$

↓

$A \to Ba \mid b$
$B \to bdB' \mid B'$
$B' \to cB' \mid adB' \mid \epsilon$

$B \to bdA' \mid A'$
$A' \to cA' \mid adA' \mid \epsilon$

Algorithm: Eliminating left recursion

I/P :- grammar G with no cycles or ∈ production

O/P → equivalent grammar with no left recursion

1. Arrange non terminals in some order $A_1, A_2 \ldots A_n$
2. for i=1 to n do begin
   for j=1 to i-1 do begin
   replace each production of the form
   $A_i \rightarrow A_j \gamma$
   by the productions $A_i \rightarrow \delta_1\gamma \mid \delta_2\gamma \mid \ldots \delta_k\gamma$
   where $A_j \rightarrow \delta_1 \mid \delta_2 \ldots \mid \delta_k$ are all current A_j productions
   end
   eliminate the immediate left recursion among the A_i productions
   end.

Elimination of Left factoring

Left factoring is a process which isolates the common parts of two productions into a single production. After elimination of left factoring the produced grammar is suitable for predictive parsing.

Rule

$A \to \alpha\beta_1 | \alpha\beta_2 | \ldots \alpha\beta_n$

can be replaced by

$A \to \alpha A'$

$A' \to \beta_1 | \beta_2 | \ldots \beta_n | \epsilon$

eg: $S \to iEtS | iEtSeS | a$

$E \to b$

left factored grammar :

$S \to iEtSS' | a$

$S' \to eS | \epsilon$

$E \to b$

Precedence of Operators

9+5*2 = 68

will be calculated as

9+(5*2)

because *, / has higher precedence than +, -

Associativity of Operators

9-5-2

+, -, *, / are left associative

list → list-digit | digit

digit → 0|1|...|9

So — calculated as (9-5)-2

a = b = c

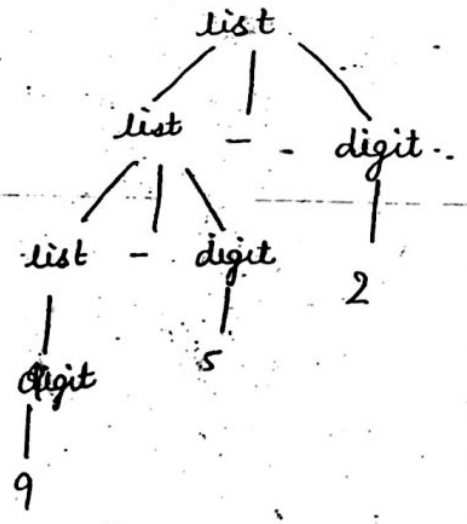
= is right associative

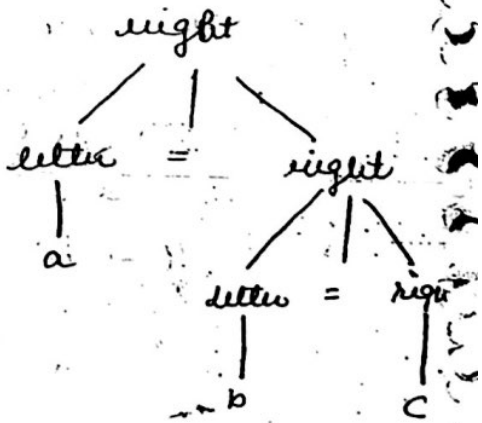
right → letter = right | letter

letter → a|b|...|z

Calculated as

a = (b = c)





Ans:- A parser scans an input token stream from left to right and groups those tokens into larger structures in order to check the syntax of an I/P string by identifying a derivation using production rules of the grammar.

Parser takes a input string w and a context free grammar Gr and produce output -

a) a parse tree for w if w is a sentence of grammar Gr ie $w \in L(G_r)$

b) Error message if w is not a sentence of grammar Gr and also provides some information like type of error and location of error.

So, Output of a Parser is a representation of a parse tree for the input, if the input is syntactically well formed.

⇒ Parsing strategies - There are two basic strategies for parsing on the basis of the way in which a parse tree can be generated.

1) Top down Parsing Strategy

By this strategy parse trees can be generated from root to leaves.

2) Bottom up Parsing

By this strategy parse trees can be generated from leaves to root

⇒ TOP DOWN PARSING

Top down parsing involves the construction of parse tree from root to leave using left most derivation.

Here the main task is the finding of a suitable production rule for each left most non terminal.

Top down parsing can be done in two ways

→ Without Backtracking

→ With Backtracking

Here we can see that at each step we have only one unique production rule for each non terminal

Top down Parsing with Backtracking

If a left most non terminal has more than one production rule

like $A \to \alpha_1 | \alpha_2 | \ldots \alpha_n$

then there is a problem to decide which A - production should be used for derivation here.

Then parser will select one of the A-production for derivation and if the derivation is done it returns the successful completion of parsing otherwise parser backtracks & reset the IIP pointer to the previous step and apply other productions

Ex→ Grammar $S \to cAd$

$A \to ab$

$A \to a$

IIP string cad

Top down Parsing without backtracking

If in top down parsing for each left most non terminal has only one unique production rule, then backtracking is not needed for derivation

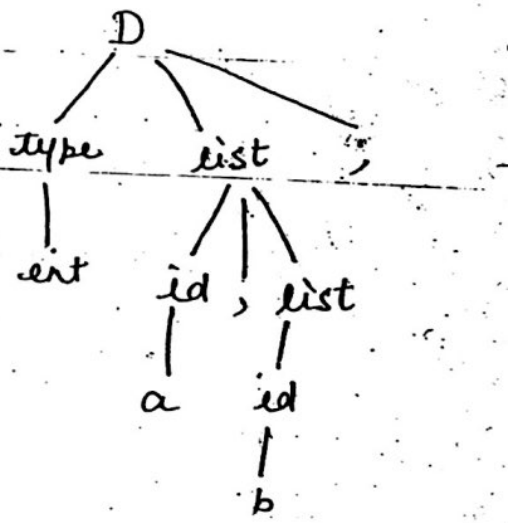
eg.

D → type list ;
type → int
type → float
list → id , list
list → id
id → a|b

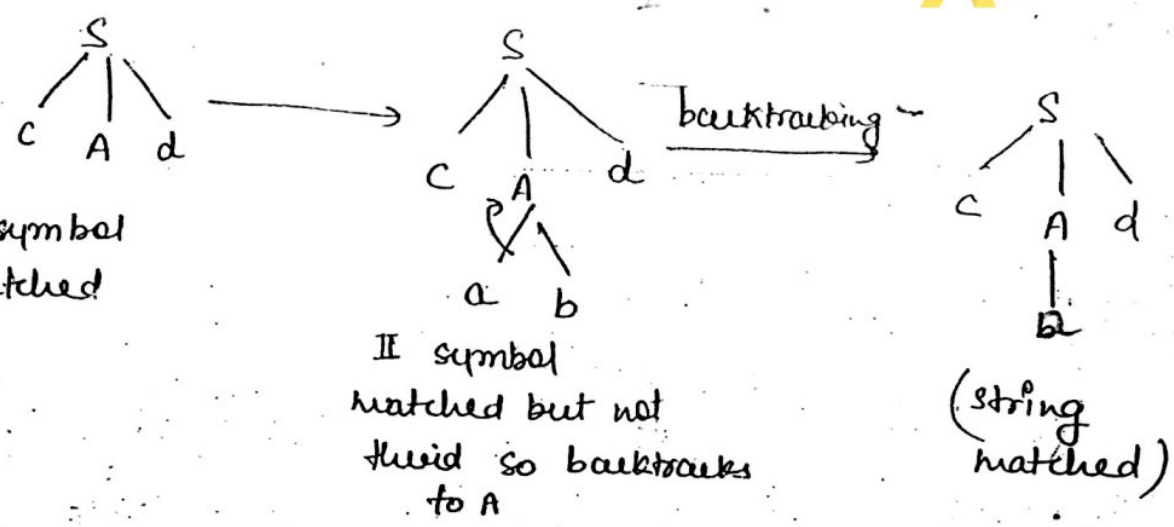
Now we have to derive a sentence int a, b ;

So

D → type list ;
→ int list ;
→ int id , list ;
→ int a , list ;
→ int a , id ;
→ int a , b ;



Derivation



Disadvantages of backtracking

Backtracking is not a good ~~too~~ ~~poo~~ way of parsing because when backtracking occures we have to undo some changes like

entries made in the symbol table have to be removed which creates a overhead.

So a top down parser having no backtracking is more reasonable.

Difficulties with Top down Parsing

1) Left Recursion - a left recursive grammar can cause a top down parser to go into an infinite loop. So, Elimination of left recursion is necessary

2) Backtracking → As already said above backtracking is not reasonable for parsing.

Recursive Descent Parsing

A recursive descent Parser is a top down method of syntax analysis which uses a collection of recursive procedures to process the input.

So "

A parser that uses a set of recursive procedures to recognize its input with no backtracking is called a 'Recursive Descent Parser'.

Recursive descent parser accepts
a left recursion free grammar
and a left factoring free grammar.

Elimination of left recursion

$A \rightarrow A\alpha \mid B$ (Left recursive grammar)

$A \rightarrow BA'$
$A' \rightarrow \alpha A' \mid \epsilon$] (Left recursion free grammar)

Elimination of Left Factoring

$A \rightarrow \alpha\beta \mid \alpha\gamma$

$A \rightarrow \alpha A'$
$A' \rightarrow \beta \mid \gamma$] Left factored free grammar

Predictive Parser → Predictive parser is a special kind of recursive descent parsing with no backtracking.

A predictive parser can perform parsing by implementing a recursive descent parsing by handling the stack of activation records. It parses a left recursion free & a left factored free grammar only.

Ex- we have grammar

$E \rightarrow E+T \mid T$

$T \rightarrow T*F \mid F$

$F \rightarrow (E) \mid id$

After eliminating left recursion from above grammar we get

$E \rightarrow TE'$

$E' \rightarrow +TE' \mid \epsilon$

$T \rightarrow FT'$

$T' \rightarrow *FT' \mid \epsilon$

$F \rightarrow (E) \mid id$

It is also called as non recursive predictive parsing because it implements the recursive descent parsing by maintaining a stack of activation records.

Predictive Parser has four main parts -

1) Input Buffer -

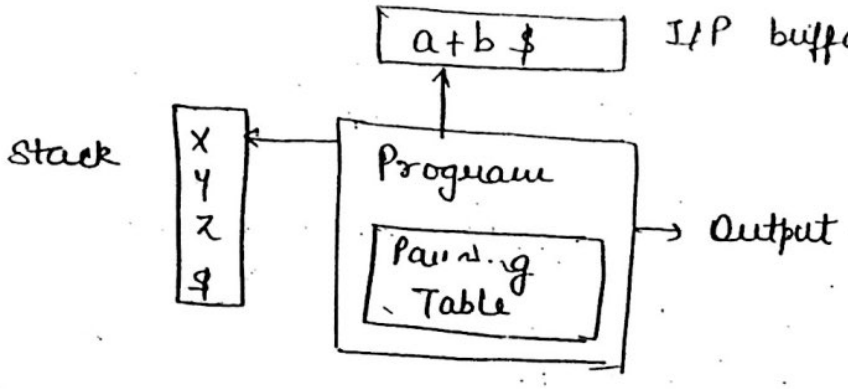
An input buffer contains the input string to be parsed followed by symbol $.

2) Stack - Stack contains a sequence of grammar symbols with $ on the bottom. Initially stack contains start symbol.

3) Parsing Table - It is a 2 dimensional array M[A, a]

where A is a non terminal and a is a terminal

4) Output - the action performed



(Model of Predictive Parser)

Parser is controlled by a program. This program determines (X) top element of stack and (a) current input symbol and performs following actions -

1) If $X = a = \$$ then parser halts & returns successful completion of parsing

2) If $X = a \neq \$$ then parser pops X off the stack and advance the input pointer to next input symbol.

3) If X is a non terminal then it checks in parsing table. If $M[X, a] = \{X \to UVW\}$ then parser replace -X by WVU with U on top of stack.

	id	+	*	(	)	$
E	$E \to TE'$			$E \to TE'$		
E'		$E' \to +TE'$			$E' \to \epsilon$	$E' \to \epsilon$
T	$T \to FT'$			$T \to FT'$		
T'		$T' \to \epsilon$	$T' \to *FT'$		$T' \to \epsilon$	$T' \to \epsilon$
F	$F \to id$			$F \to (E)$		

and the input string is id + id * id that we can derive in following manner.

moves by predictive parser →	Stack	Input	Action
	$ E	id + id * id $	
	$ E'T	id + id * id $	$E \to TE'$
	$ E'T'F	id + id * id $	$T \to FT'$
	$ E'T'id	id + id * id $	$F \to id$
	$ E'T'	+ id * id $	POP
	$ E'	+ id * id $	$T' \to \epsilon$
	$ E'T+	+ id * id $	$E' \to +TE'$
	$ E'T	id * id $	POP
	$ E'T'F	id * id $	$T \to FT'$
	$ E'T'id	id * id $	$F \to id$
	$ E'T'	* id $	POP
	$ E'T'F*	* id $	$T' \to *FT'$
	$ E'T'F	id $	
	$ E'T'id	id $	$F \to id$
	$ E'T'	$	$T' \to \epsilon$
	$ E'	$	$E' \to \epsilon$
	$	$	successful parsing.

Construction of Predictive Parsing Table

Finding FIRST and FOLLOW

Rules to calculate FIRST(X)

(1) If X is a terminal then
FIRST(X) = {X}

(2) If X → ∈ is a production then FIRST(X) = {∈}

(3) If X is a non terminal and X → aα then FIRST(X) = {a}

(4) If X → $Y_1 Y_2 \dots Y_K$ is a production for some K ≥ 1 and if
FIRST(Y_i) = {a} and
FIRST(Y_1), FIRST(Y_2), ... FIRST(Y_{i-1}) contains ∈
then FIRST(X) = {a}

If FIRST(Y_j) contains ∈
where j = 1, 2 ... i-1
then add every non ∈ symbol in FIRST(Y_i) to FIRST(X).

If ∈ is in FIRST(Y_j) for all j = 1, 2 ... k
then add ∈ to FIRST(X).

Rules to calculate FOLLOW(A)

1) If S is a start symbol

FOLLOW(S) = { $ }

2) If there is a production $A \to \alpha B \beta$

then FOLLOW(B) = FIRST(B) − ϵ

3) If there is a production

$A \to \alpha B$

or $A \to \alpha B \beta$ if FIRST(B) contains ϵ

= { ϵ }

then FOLLOW(B) = FOLLOW(A)

Ex Let we have a following grammar

$E \to TE'$

$E' \to +TE' \mid \epsilon$

$T \to FT'$

$T' \to *FT' \mid \epsilon$

$F \to (E) \mid id$

and we have to construct a parsing table for this grammar

First calculate FIRST & FOLLOW for each non terminal in grammar:

1) FIRST(E) = FIRST(T) = FIRST(F) = { (, id }
by rule 4 and 3 of FIRST

2) FIRST(E') = { +, ∈ } by rule 1 & 2

3) FIRST(T') = { *, ∈ } by rule 1 & 2

4) FOLLOW(E) = { $ } by rule 1 of follow

FOLLOW(E) = FOLLOW(E') by rule 3

$A \to \alpha B$, $E \to TE'$

FOLLOW(E') = {) } by rule 2 $A \to \alpha B\beta$, $F \to (E)$

so FOLLOW(E) = FOLLOW(E') = { $,) }

5) FOLLOW(T) = FOLLOW(T') by rule 3

$T \to FT'$

FOLLOW(T) = FIRST(E') − ∈ (by rule 2)

= { + } $A \to \alpha B\beta$, $E' \to +TE'$

~~Foll~~

and $A \to \alpha B\beta$

$E' \to +TE'$ by rule 3

because FIRST(E') contains ∈

so FOLLOW(T) = FOLLOW(E') = { $,) }

so FOLLOW(T) = FOLLOW(T') = { +,), $ }

c) FOLLOW (F)

$A \to \alpha B \beta$

$T' \to *FT'$ by rule 3

because FIRST(T') contains ϵ

so FOLLOW(F) = FOLLOW(T') = {+,), $}

and by rule 2

$A \to \alpha B \beta$

$T' \to *FT'$

~~because~~ ~~FIR~~

FOLLOW(F) = FIRST(T') − ϵ

= {*}

so FOLLOW(F) = {+, *,), $}

Algorithm for construction of a predictive parsing table ① Remove left Recursion

② ~~Remove~~ Left factor the Grammar.

1) For each production $A \to \alpha$ do following steps

a) Calculate FIRST(A)

if FIRST(A) = {a} then add $A \to \alpha$ to M[A, a]

b) If FIRST(A) contains ϵ then for each terminal b in FOLLOW(A) add $A \to \epsilon$ to M[A, b]

If ϵ in FIRST(A) and $ in FOLLOW(A) then add $A \to \epsilon$ to M[A, $] also.

Now construct the parsing table

I production $E \to TE'$

FIRST (E) = { (, id }

So add $E \to TE'$ in table for terminal (, id

II production $E' \to +TE' \mid \epsilon$

FIRST (E') = { +, ϵ }

because contains ϵ add $E' \to \epsilon$ to FOLLOW (E') = {), $ }

III production $T \to FT'$

FIRST (T) = { (, id } add $T \to FT'$ for { (, id }

	id	+	*	(	)	$
E	$E \to TE'$			$E \to TE'$	~~$E \to TE'$~~	
E'		$E' \to +TE'$			$E' \to \epsilon$	$E' \to \epsilon$
T	$T \to FT'$			$T \to FT'$		
T'		$T' \to \epsilon$	$T' \to *FT'$		$T' \to \epsilon$	$T' \to \epsilon$
F	$F \to id$			$F \to (E)$		

IV Production $T' \to *FT' \mid \epsilon$

FIRST (T') = { *, ϵ } add $T' \to *FT'$ to *

because it contains ϵ

add $T' \to \epsilon$ to FOLLOW (T') = { +,), $ }

V Production $F \to (E) \mid id$

FIRST (F) = { (, id }

add $F \to (E) \mid id$ for (and id respectively

⇒ LL(1) Grammar → A grammar is a LL(1) grammar if its LL(1) parsing table has no multiply defined entries. Ambiguous and left recursive grammar can not be a LL(1).

So LL(1) parsers are those parsers which uses only one input symbol to determine which production should be apply, and the grammar for which such parsers can be generated are called LL(1) grammar.

In LL(1) first L stands for left to right scanning of I/P.
second L stands for producing left most derivation.
1 indicates that the next I/P symbol is used to decide next parsing process. (ie length of lookahead is 1).

Properties of LL(1) Grammar

1) No ambiguous or left recursive grammar can be LL(1)
2) All the entries in the parsing tables are unique for LL(1) grammar

For a grammar to be LL(1) following conditions must be satisfied -

for production $A \to \alpha | \beta$

{ FIRST (α) $\cap$ FIRST (β) $= \phi$

and

if FIRST (β) contains ϵ and FIRST (α) does not contain ϵ

then

FIRST (α) $\cap$ FOLLOW (A) $= \phi$

}

eg. $S \to ict\,SS' \mid a$ ($\alpha\,\beta\beta$)

$S' \to eS \mid \epsilon$ ($A \to \beta\beta$)

$C \to b$

FIRST (S) : {i, a}

(S') → {e, ϵ}

C → b

Follow (S) → {e, $}

Follow (S) = Follow (S') = {e, $}

Parsing table

	a	b	e	i	t	$
S	S → a			S → ictSS'		
S'			S' → ϵ S' → eS			S' → ϵ
C		C → b				

above grammar is not a LL(1) grammar because its parsing table has multiple defined entries.

FIRST (S) = { i, a }

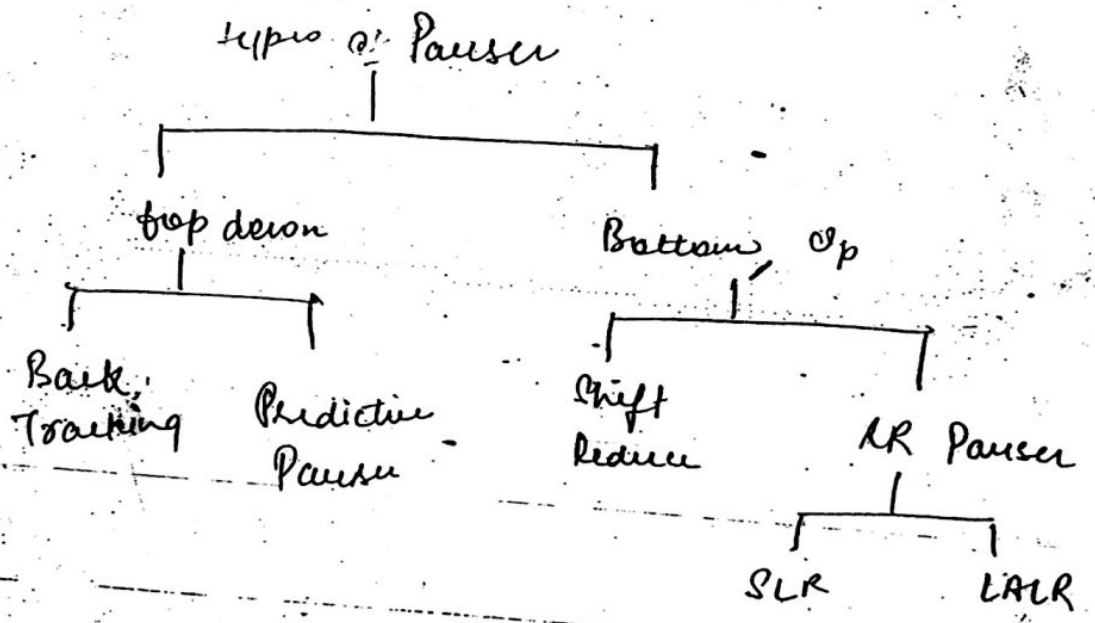
FIRST (S') = { e, ∈ }

FIRST (C) = { b }

FOLLOW (S) = FOLLOW (S') = { e, $ }

FOLLOW (S') = { e, $ }

FOLLOW (C) = { t }



LR parser ... R

... can be used ... way

entire free grammars

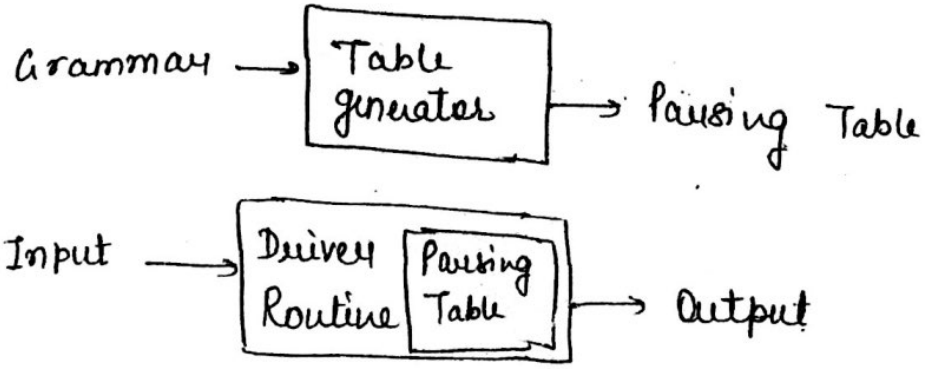
This technique is called LR(K) parsing where L stands for Left to Right scanning of I/P. R stands for constructing right most derivation in reverse and k stands for no. of input symbols of look ahead.

So, LR parsers are bottom up parsers which scan the input from left to right and construct a right most derivation in reverse.

LR parser consists of two parts

- a driver routine
- a parsing table

driver routine is same for all parsers but parsing table changes from one parser to another.



There are three different techniques for LR parsing tables.

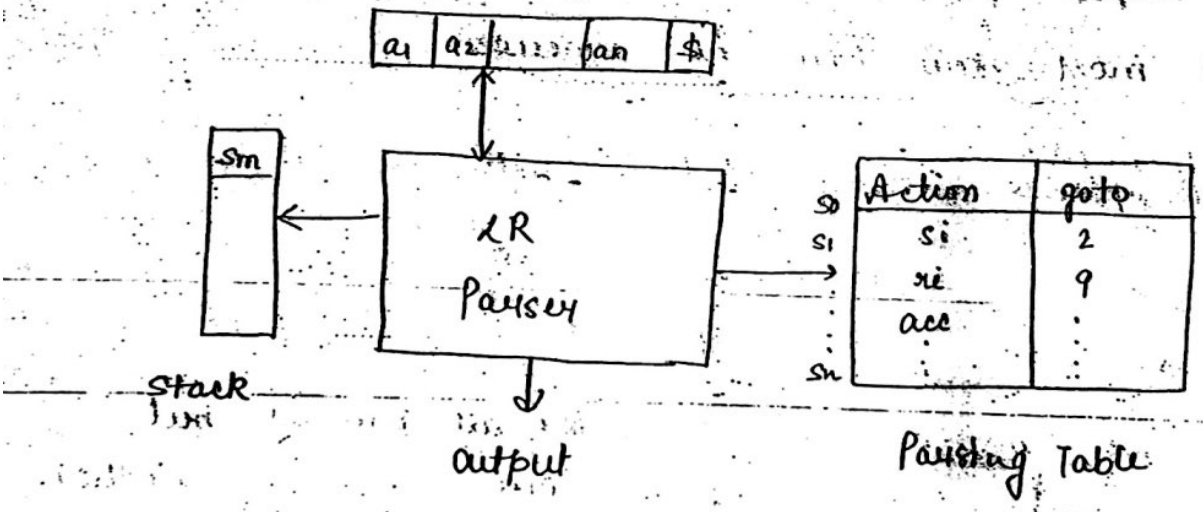
1) SLR – Simple LR

easiest to implement but fail for certain grammars.

2) Canonical LR – can work on a very large class of grammars but expensive to implement

3) LALR – LookAhead LR is intermediate in power b/w SLR and canonical LR.

Structure of LR Parser –



Configuration of LR Parser

Parser has

1) Input
2) Stack
3) Parsing Table

Input is read from left to right one symbol at a time.

Stack can be in the following configuration

$s_0 X_1 s_1 X_2 s_2 X_3 \ldots X_m s_m$

where each X_i is a grammar symbol and each s_i is a state

and s_m is on top of stack - top of stack always contain a ~~stack~~ state.

Parsing table has two parts

1) Action
2) Goto

Parsing action table entry action $[s_m, a_i]$ for state s_m and input a_i can have four possible values

a) s_i which means shift to state i.
b) r_j reduce by using the j^{th} rule.
c) acc means accept
d) error

GOTO takes a state and grammar symbol as arguments and produces a state

Configuration of LR Parser.

I component is stack contents &

II component is input

$(S_0 X_1 S_1 X_2 \ldots X_m S_m \; , \; a_i a_{i+1} \ldots a_n \$)$

next move of parser depends on current input symbol a_i and top of stack S_m

ACTION $[S_m, a_i]$

four types of moves can be done

1) If ACTION $[S_m, a_i]$ = shift S

means parser executes a shift move and generate the following configuration

$(S_0 X_1 S_1 X_2 \ldots X_m S_m \; a_i \; S \; , \; a_{i+1} \ldots a_n \$)$

here current I/P symbol a_i and state S is shifted to stack where S = GOTO $[S_m, a_i]$

then configuration will be

$(S_0 X_1 S_1 X_2 \dots X_{m-r} S_{m-r} A s , a_i a_{i+1} \dots a_n \$)$

Here parser popped off $2r$ symbols of the stack and push A, left side of production on to stack and state s.

where $s = ACTION[s_{m-r}, A]$

(3) If $ACTION[s_m, a_i] = accept$ → we are parsing completed

or

(4) If $ACTION[s_m, a_i] = error$ - calls an error recovery routine.

$E \to T$

$T \to T * F$

$T \to F$

$F \to (E)$

$F \to id$

Let the given parsing table is

States	Action						Goto		
	id	+	*	(	)	$	E	T	F
0	S5			S4			1	2	3
1		S6				acc			
2		r2	S7		r2	r2			
3		r4	r4		r4	r4			
4	S5			S4			8	2	3
5		r6	r6		r6	r6			
6	S5			S4			9	3	
7	S5			S4				10	
8		S6			S11				
9		r1	S7		r1	r1			
10		r3	r3		r3	r3			
11		r5	r5		r5	r5			

Now you have to parse a string id * id * id using above parsing table by LR parser.

using

stack	input
0	id * id + id $
0 id 5	* id + id $
0 F 3	* id + id $
0 T 2	* id + id $
0 T 2 * 7	id + id $
0 T 2 * 7 id 5	+ id $
0 T 2 * 7 F 10	+ id $
0 T 2	+ id $
0 E 1	+ id $
0 E 1 + 6	id $
0 E 1 + 6 id 5	$
0 E 1 + 6 F 3	$
0 E 1 + 6 T 9	$
0 E 1	$
accept	

Generation of Parsing Table

LR parsing tables can be generated by three methods

1) SLR
2) canonical LR.
3) LALR

(1) SLR Parser and construction of SLR parsing tables

For the construction of SLR parser, first we

→ make the augmented grammar

means: if a grammar has a start symbol S then we make a augmented grammar with new production

$$S' \rightarrow S$$

this new production indicates the parser when it should stop parsing

→ generate canonical collection of LR(0) items

eg. $A \rightarrow XYZ$ generate four items

$A \rightarrow \cdot XYZ$

$A \rightarrow X \cdot YZ$

$A \rightarrow XY \cdot Z$

$A \rightarrow XYZ \cdot$

Grammar is

$E \to E+T \mid T$

$T \to T*f \mid f$

$f \to (E) \mid id$

Follow (E) = +,), $

(T) = *, +,), $

Augmented grammar for above grammar is

0 → $E' \to E$

1 → $E \to E+T$ r_1

2 → $E \to T$ r_2

3 → $T \to T*f$ r_3

4 → $T \to f$ r_4

5 → $f \to (E)$ r_5

6 → $f \to id$ r_6

How find I_0

$E' \to .E$

$E \to .E+T$

$E \to .T$

$T \to .T*f$

$T \to .f$

$f \to .(E)$

$f \to .id$

I_0 E.

I_1 (I_0, E)

$E' \to E.$

$E \to E.+T$

I_2 (I_0, T) (I_4, T)

$E \to T.$

$T \to T.*f$

I_3 (I_0, F) (I_4, F) (I_6, F)

$T \to F.$

I_4 $(I_0, ()$ $(I_4, ()$ $(I_6, ()$ $(I_7, ()$

$F \to (.E)$

$E \to .E+T$

$E \to .T$

$T \to .T*F$

$T \to .F$

$F \to .(E)$

$F \to .id$

I_5 (I_4, id)

$F \to id.$

I_6 $(I_1, +)$ $(I_8, +)$

$E \to E+.T$

$T \to .T*F$

$T \to .F$

$F \to .(E)$

$F \to .id$

I_7 $(I_2, *)$ $(I_9, *)$

$T \to T*.F$

$F \to .(E)$

$F \to .id$

I_8 (I_4, E)

$F \to (E.)$

$E \to E.+T$

I_9 (I_6, T)

$E \to E+T.$

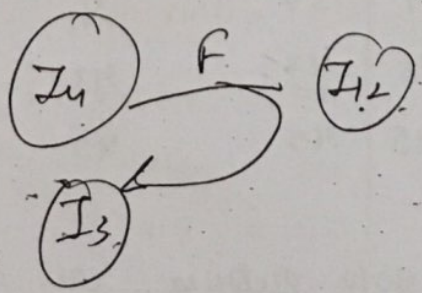
$T \to T.*F$

I_{10} (I_7, F)

$T \to T*F.$

I_{11} $(I_8,))$

$F \to (E).$



Ex Consider the following grammar and generate the SLR(1) table

$S \to L = R$
$S \to R$
$L \to *R$
$L \to id$
$R \to L$

augmented grammar is

$S' \to S$
$S \to L = R$
$S \to R$
$L \to *R$
$L \to id$
$R \to L$

I_0
$S' \to .S$
$S \to .L = R$
$S \to .R$
$L \to .*R$
$L \to .id$
$R \to .L$

$I_1 (I_0, S)$
$S' \to S.$ ✓

$I_2 (I_0, L)$
$S \to L. = R$
$R \to L.$ ✓

$I_3 (I_0, R)$
$S \to R.$ ✓

$I_4 (I_0, *)$ $(I_6, *)$ $(I_{11}, *)$
$L \to *.R$
$R \to .L$
$L \to .*R$
$L \to .id$

$I_5 (I_0, id)$ (I_6, id) (I_{11}, id)
$L \to id.$ ✓

$I_6 (I_2, =)$
$S \to L = .R$
$R \to .L$
$L \to .*R$
$L \to .id$

$I_7 (I_4, R)$
$L \to *R.$

$I_8 (I_4, L)$ (I_6, L)
$R \to L.$

$I_9 (I_6, R)$
$S \to L = R.$

Bottom Up Parsing

Bottom up Parsing - In this technique parsers build parse trees from the bottom (leaves) to the top (root).

IIP to the parser is being scanned from left to right one symbol at a time.

The Bottom up parsing method is called Shift-Reduce Parsing because it consists of shifting IIP symbols onto a stack until the right side of the production appears on top of the stack. The right side then can be replaced (reduced) by the symbol on the left side of the production.

One kind of shift Reduce Parser (SR-parser) is an Opator Precedence Parser.

Shift Reduce Parsing

SR parsing follows a process of right most derivation of a string w. This is a process of reducing a string w to the start symbol of a grammar.

eg. let we have a grammar

$S \rightarrow aAcBe$

$A \rightarrow Ab \mid b$

$B \rightarrow d$

and we have to derive the string abbcde

So in bottom up parsing the string abbcde will be scanned from L to R for some substrings that can match the right side of some production.

a Abcde
→ aAcde
→ aAcBe
→ S

Each replacement of the right side of production by left side is called Reduction.

Handle :- A handle of a right sentential form γ is a production $A \to \beta$ and a position of γ where β may be found and replaced by A to produce the previous right sentential form in a rightmost derivation of γ.

because $S \overset{*}{\underset{rm}{\Longrightarrow}} \alpha A w \underset{rm}{\Longrightarrow} \alpha \beta w$

eg. abbcde is a right sentential form whose handle is $A \to b$ at position 2.

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

Handle Pruning

Rightmost derivation in reverse often called Canonical Reductions sequence is obtained by handle pruning.

means Handle pruning is the process of obtaining R.M.D in reverse by shift and reduction sequences.

Stack implementation of Shift Reduce Parsing

In automating the SR parsing process by handle pruning there are two problems

- how to locate a handle in right sentential form
- what production to chose if there are more than one production with the same right side

To implement SR parser use - stack & IIP buffer

Stack	IIP
$	w$

IIP buffer holds the IIP string to be parsed with $ at the end.

during LtoR scanning of IIP parser shift zero or more IIP symbols onto the stack, until it is ready to reduce a string β on top of the stack. Parser then reduces string β by the left side of production and repeat this cycle until the stack contains start symbol & IIP is empty or an error has encountered.

If the grammar is

$E \to E+E$

$E \to E*E$

$E \to id_1 \mid id_2 \mid id_3$

and we have to derive a string $id_1 = id_2 * id_3$ by SR parser

Stack	I/P	action
$	$id_1 + id_2 * id_3 \$$	shift
$ id_1	$+ id_2 * id_3 \$$	reduce by $E \to id_1$
$ E	$+ id_2 * id_3 \$$	shift
$ $E+$	$id_2 * id_3 \$$	shift
$ $E + id_2$	$* id_3 \$$	shift
$ $E + E$	$* id_3 \$$	reduce by $E \to id_2$
$ $E + E *$	$id_3 \$$	shift
$ $E + E * id_3$	$\$$	shift
$ $E + E * E$	$\$$	reduce by $E \to id_3$
$ $E + E$	$\$$	reduce by $E \to E*E$
$ E	$\$$	reduce by $E \to E+E$
		Accept

Operations of SR Parser

(1) **Shift** - In this operation next I/P symbol is shifted to the top of the stack.

(2) **Reduce** - If the parser knows the right end of the handle at top of the stack and if it locate the left end of the handle then it decides that with what non terminal the stack should be replaced.

(3) **Accept** - This operation shows the successful parsing.

(4) **Error** - Parser discovers that syntax error has occurred and then calls an error recovery routine.

Example →

$E \to E+T \mid T$

$T \to T*F \mid F$

$F \to id$

For I/P string id + id * id

The R.H.D is

$E \to E+T$

$\to E+T*F$

$\to E+T*id$

$\to E+F*id$

$\to E+id*id$

$\to T+id*id$

$\to F+id*id \to id+id*id$

Stack	I/P buffer	Action
$	id + id * id $	shift
$ id	+ id * id $	Reduce by F → id
$ F	+ id * id $	Reduce by T → F
$ T	+ id * id $	Reduce by E → T
$ E	+ id * id $	shift
$ E +	id * id $	shift
$ E + id	* id $	Reduce F → id
$ E + F	* id $	Reduce T → F
$ E + T	* id $	shift
$ E + T *	id $	shift
$ E + T * id	$	Reduce F → id
$ E + T * F	$	Reduce by T → T * F
$ E + T	$	Reduce by E → E + T
$ E	$	Accept

Conflicts During Shift Reduce Parsing

There are grammars for which shift reduce parser can not be used. For such grammars SR parser reach in a situation of conflict.

There are two types of conflict

(1) **Shift Reduce Conflict** - This conflict arises when the parser reach in a configuration in which it can not decide whether to shift or to reduce.

eg. stmt → if expr then stmt
→ if expr then stmt else stmt
→ other

Let the parser is in following configuration

stack	ip
... if expr then stmt	else $

It is a situation of shift reduce conflict because we can not say whether we should reduce if expr then stmt or shift else.

(2) **Reduce-Reduce Conflict** - This conflict arises when SR parser can not decide that which of the several reduction should be used for parsing.

eg. 1. stmt → id (parameter_list)

2. stmt → expr := expr

3. parameter_list → parameter_list , parameter

4 parameter_list → parameter

5 parameter → id

6 expr → id (exp_list)

7 expr → id

8 expr_list → expr_list , expr

9 expr_list → expr

eg A(I,J)

token stream id (id, id)

stack	I/P
.... id(id	, id) $

In this situation we can't say that whether we should reduce id by production 5 or 7.

Correct choice is production 5 if A is a procedure
7 if A is an array.

Operator Precedence Parsing

This is a form of shift reduce parsing which is used to parse operator grammar. This parsing is especially used for parsing expressions.

Operator precedence grammar

these grammars should have property

- that no production right side is ϵ
- no production has two adjacent non terminals.

eg. $E \to EAE \mid (E) \mid -E \mid id$

$A \to + \mid - \mid * \mid / \mid \uparrow$

is not an operator grammar, because the right side EAE has two consecutive non terminals.

Operator grammar can be obtained by substituting the value of A in I production

we get

$E \to E+E \mid E-E \mid E*E \mid E/E \mid E \uparrow E \mid (E) \mid -E \mid id$

Operator Precedence parsing uses the information about the precedence & associativity of operators to guide the parse tree

Precedence Relations

In operator precedence parsing we use three disjoint precedence relations between pair of terminals.

$<\cdot$
$\doteq$
$\cdot>$

These relations guide the selection of handles and have following meaning.

$a <\cdot b$	a yields precedence to b
$a \doteq b$	a has same precedence as b
$a \cdot> b$	a takes precedence over b.

There are two methods of generating precedence relations

1. By determining the associativity & precedence of operators
2. By finding unambiguous grammar

(will be discussed later)

Using Precedence Relations for Parsing

Let we have operator precedence relations as

	id	+	*	$
id		$\cdot>$	$\cdot>$	$\cdot>$
+	$<\cdot$	$\cdot>$	$<\cdot$	$\cdot>$
*	$<\cdot$	$\cdot>$	$\cdot>$	$\cdot>$
$	$<\cdot$	$<\cdot$	$<\cdot$	

These precedence relations are used to delimit the handle of a right sentential form.

eg. If the string is id+id*id

then the string with precedence relations inserted is

$ <· id ·> + <· id ·> * <· id ·> $

Process of finding handle

1) Scan the string from L to R until ·> is encountered.
2) Then scans backwards (to the left) over any = until a <· is encountered. ~~Then scans backwards to~~ $
3) Handle contains everything between <· and ·>

In above string on scanning from L to R until ·> and then by scanning backwards to Left until <· we get id.

So the handle is the first id

we then reduce id to E

and we get right sentential form E+id*id

Similarly after reducing the remaining id's to E

we get E+E*E

now consider the string $+*$ by deleting non terminal from above string

then

$ <· + <· * ·> $

i.e. to the rule first handle is E+E

Rules to Calculate Operator Precedence Relations

I Method - Relations from Associativity and Precedence

Rule 1:- If operator θ_1 has higher precedence than operator θ_2

then generate $\theta_1 \gtrdot \theta_2$ & $\theta_2 \lessdot \theta_1$

eg. b/w + and *

relations will be + <· * and * ·> +

Rule 2:- If θ_1 and θ_2 are operators of equal precedence,

then generate $\theta_1 \gtrdot \theta_2$ and $\theta_2 \gtrdot \theta_1$ { if operators are left associative }

or make $\theta_1 \lessdot \theta_2$, $\theta_2 \lessdot \theta_1$ { if they are right associative }

eg. relation b/w + and + and + and - can be

+ ·> +

+ ·> -

- ·> +

- ·> -

because they are left associative

↑ is right associative so

↑ <· ↑

it shows that in E-E+E, (E+E) will be selected first as handle.

and in E↑E↑E will have the last E↑E first

E↑E selected first.

Rule 3: Make $\theta \lessdot id$

$id \gtrdot \theta$

$\theta \lessdot ($

$(\lessdot \theta$

$) \gtrdot \theta$

$\theta \gtrdot)$

$\theta \gtrdot \$$

$\$ \lessdot \theta$

and also

$(\doteq)$	$\$ \lessdot ($	$\$ \lessdot id$
$(\lessdot ($	$id \gtrdot \$$	$) \gtrdot \$$
$(\lessdot id$	$id \gtrdot)$	$) \gtrdot)$

By these rules operator precedence relations by following grammar can be created

Grammar $E \to E+E \mid E-E \mid E*E \mid E/E \mid E \uparrow E \mid (E) \mid -E \mid id$

	+	-	*	/	↑	(	)	id	$
+	$\gtrdot$	$\gtrdot$	$\lessdot$	$\lessdot$	$\lessdot$	$\lessdot$	$\gtrdot$	$\lessdot$	$\gtrdot$
-	$\gtrdot$	$\gtrdot$	$\lessdot$	$\lessdot$	$\lessdot$	$\lessdot$	$\gtrdot$	$\lessdot$	$\gtrdot$
*	$\gtrdot$	$\gtrdot$	$\gtrdot$	$\gtrdot$	$\lessdot$	$\lessdot$	$\gtrdot$	$\lessdot$	$\gtrdot$
/	$\gtrdot$	$\gtrdot$	$\gtrdot$	$\gtrdot$	$\lessdot$	$\lessdot$	$\gtrdot$	$\lessdot$	$\gtrdot$
↑	$\gtrdot$	$\gtrdot$	$\gtrdot$	$\gtrdot$	$\lessdot$	$\lessdot$	$\gtrdot$	$\lessdot$	$\gtrdot$
(	$\lessdot$	$\lessdot$	$\lessdot$	$\lessdot$	$\lessdot$	$\lessdot$	$\doteq$	$\lessdot$	—
)	$\gtrdot$	$\gtrdot$	$\gtrdot$	$\gtrdot$	$\gtrdot$	—	$\gtrdot$	—	$\gtrdot$
id	$\gtrdot$	$\gtrdot$	$\gtrdot$	$\gtrdot$	$\gtrdot$	—	$\gtrdot$	—	$\gtrdot$
$	$\lessdot$	$\lessdot$	$\lessdot$	$\lessdot$	$\lessdot$	$\lessdot$	—	$\lessdot$	—

id * (id ↑ id) - id / id

Stack		Input
$	<·	id * (id ↑ id) - id / id $
$ <· id	·>	* (id ↑ id) - id / id $
$ E	<·	* (id ↑ id) - id / id $
$ <· E *	<·	(id ↑ id) - id / id $
$ <· E * <· (	<·	id ↑ id) - id / id $
$ <· E * <· (<· id	·>	↑ id) - id / id $
$ <· E * <· (E	<·	↑ id) - id / id $
$ <· E * <· (<· E ↑	<·	id) - id / id $
$ <· E * <· (<· E ↑ <· id	·>	) - id / id $
$ <· E * <· (<· E ↑ E	·>	) - id / id $
$ <· E * <· (E	≐	) - id / id $
$ <· E * <· (≐ E)	·>	- id / id $
$ <· E * E	·>	- id / id $
$ E	<·	- id / id $
$ <· E -	<·	id / id $
$ <· E - <· id	·>	/ id $
$ <· E - E	<·	/ id $
$ <· E - <· E /	<·	id $
$ <· E - <· E / <· id	·>	$
$ <· E - <· E / E	·>	$
$ <· E - E	·>	$
$ E		$

Accept

Rules

(1) $a \doteq b$, if there is a right side of a production of the form $\alpha a \beta b \gamma$ where β is either ϵ or a single non terminal.

means $a \doteq b$ if a appears immidiately to the left of b , or if they appear seperated by one non terminal. eg. $S \to iCtSeS \Rightarrow i \doteq t$ & $t \doteq e$

(2) $a <\cdot b$, if for some non terminal A there is a right side of the form ~~αA~~ $\alpha a A \beta$ and $A \to \gamma b \delta$ where γ is either ϵ or single Non terminal

means $a <\cdot b$ if a non terminal A appears immidiately to the right of a and derives a string in which b is the first terminal symbol.

eg. $S \to iCtS$ & $C \to b$

so $i <\cdot b$

(3) $a \cdot> b$ if for some non terminal A there is a right side of the form $\alpha A b \beta$ and $A \to \gamma a \delta$, where γ is either ϵ or single non terminal.

means : $a \cdot> b$ if a non terminal appear immidiately to the left of b derives a string whose last terminal is a.

eg. $S \to iCtS$ and $C \to b$.

$b \cdot> t$

relation among all the terminals involved in the grammar.

An operator precedence grammar is an ϵ free grammar in which the precedence relations $<\cdot$, $\doteq$ and $\cdot>$ are disjoint.

means for any pair of terminals a and b, never more than one of the relations $a <\cdot b$, $a \doteq b$ and $a \cdot> b$ true.

eg: $E \to E+E \mid E*E \mid (E) \mid id$

this grammar is not an operator precedence grammar as by the rule (ii) and (iii) two precedence relations hold b/w + and +

by rule (ii) in $E \to E+E$

$\alpha a A B$

$a = +$

$A = E$

and $E \to E+E$

so E derives string in first terminal is +

so, $+ <\cdot +$

by rule (iii) in $E \to E+E$

$\alpha A b B$

$A = E$

$b = +$

$E \to E+E$

last of E is +

so $+ \cdot> +$

Therefore above grammar is not an operator precedence grammar.

Transform this grammar into operator precedence & unambiguous grammar

$E \to E+T \mid T$

$T \to T*F \mid F$

$F \to id \mid (E)$

Now find the precedence relations by those three rules. These rules can be modified as follows:-

Calculate FIRST or LEADING & LAST or TRAILING all the non terminals.

LEADING $(A) = \{a \mid A \overset{+}{\Rightarrow} \gamma a \delta$, where γ is either ϵ or single non terminal

LAST $(A) = \{a \mid A \overset{+}{\Rightarrow} \gamma a \delta$, where δ is either ϵ or single non terminal.

now find precedence relations

1) $\doteq$: in above grammar and by rule (i) this relation holds b/w (and) only

so $(\doteq)$

2) $<\cdot$ - we find a, terminal a immediately to the left of a non terminal to play the rules of a & A then make $a <\cdot$ LEADING(A) also make $\$ <\cdot$ LEAD(start symbol)

3) $\cdot>$ we find a terminal b immediately to the right of a non terminal A then make

~~$b > LAST(A)$~~

$LAST(A) > b$ also make LAST(start symbol) $> \$$

further. If a is in LEADING(B), and there is a production of the form $A \to B\alpha$ then a is in LEADING(A)

Consider the grammar $E \to E+T \mid T$
$T \to T*F \mid F$
$F \to (E) \mid id$
and find precedence relations

so by rule (1) $\doteq$ ($\doteq$)

by rule 2 $\lessdot$

$+ \lessdot$ leading(T)
$* \lessdot$ leading(F)
$(\lessdot$ leading(E)
$\$ \lessdot$ leading(E)

	LEADING	TRAILING
E	*, +, (, id	*, +,)
T	*, (, id	*,), id
F	(, id	), id

so we get

$+ \lessdot *$ $+ \lessdot ($ $+ \lessdot id$
$* \lessdot ($ $* \lessdot id$
$(\lessdot *$ $(\lessdot +$ $(\lessdot ($ $(\lessdot id$
$\$ \lessdot *$ $\$ \lessdot +$ $\$ \lessdot ($ $\$ \lessdot id$

by rule 3 for $\gtrdot$

last(E) $\gtrdot$ +
last(T) $\gtrdot$ *
last(E) $\gtrdot$)
and last(E) $\gtrdot$ $

we get

$* \gtrdot +$ $) \gtrdot +$ $id \gtrdot +$
$* \gtrdot *$ $) \gtrdot *$ $id \gtrdot *$
$* \gtrdot)$ $+ \gtrdot)$ $) \gtrdot)$ $id \gtrdot)$
$* \gtrdot \$$ $+ \gtrdot \$$ $) \gtrdot \$$ $id \gtrdot \$$

now we get the table as

row \ col	+	*	(	)	id	$
+	⋗	⋖	⋖	⋗	⋖	⋗
*	⋗	⋗	⋖	⋗	⋖	⋗
(	⋖	⋖	⋖	≐	⋖	
)	⋗	⋗		⋗		⋗
id	⋗	⋗		⋗		⋗
$	⋖	⋖	⋖		⋖	

Operator Precedence Parsing Algorithm

repeat forever

if only $ is on the stack and $ is on the input then accept and ~~blank~~ break

else

begin

let a be the top most terminal symbol on the stack, and let b be the current input symbol

if a ⋖ b or a ≐ b then shift b on to the stack

else if a ⋗ b then /* reduce */

repeat pop the stack

until the top stack terminal is related by ⋖ to the terminal most recently popped

else call the error correcting routine

end

Operator precedence parsers have no need to store the large table of precedence relations because this table can be encoded by only two functions f and g, which map terminal symbols to the integers.

We select f and g for symbol a and b, like

$f(a) < g(b)$ whenever $a <\cdot b$

$f(a) = g(b)$ " $a \doteq b$.

$f(a) > g(b)$. " $a \cdot> b$

Method for finding precedence functions for a table

① Create symbols f_a and g_a for each a that is a terminal or $.

② Partition the created symbols into as many groups as possible.

means if $a \doteq b$. then f_a and g_b are in same group

if $a \doteq b$ and $c \doteq b$

then f_a and f_c are in same group as they are same group as g_b

③ Created a directed graph whose nodes are the groups found in step 2.

if $a <\cdot b$ then create an edge from g_b to f_a

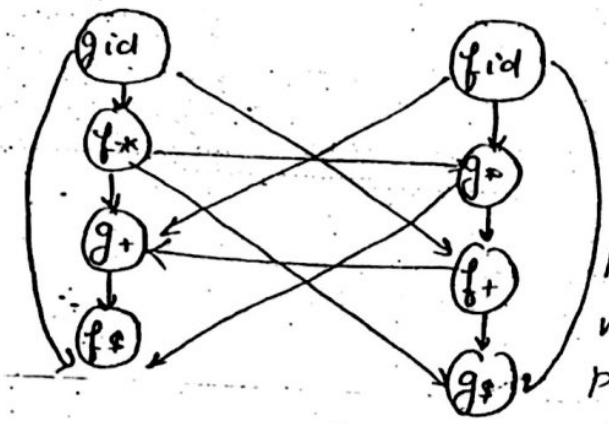
if $a \cdot> b$ then create an edge from f_a to g_b.

4) If the graph constructed in 3 has a cycle then no precedence function exists If there are no cycles then $f(a)$ be the length of the longest path beginning at the group of a ; let $g(a)$ be the length of the longest path from the group of a.

eg. Generate the precedence function table for

	id	+	*	$
id		⋗	⋗	⋗
+	⋖	⋗	⋖	⋗
*	⋖	⋗	⋗	
$	⋖	⋖	⋖	

because there are no $\doteq$ relationships, so each symbol is in group by itself



then generate directed graph.

as there are no cycles, so precedence function exists

now find out the longest path from each group and create the precedence function table

	id	+	*	$
f	4	2	4	0
g	5	1	3	0

(6)

As we know that after syntax analysis, the semantic analysis of input program is needed.

To perform semantic analysis in a syntax directed fashion, we attach some semantic rules with each productions in context free grammar.

So, we can say

"Syntax directed translation scheme is a process of allowing semantic actions to be attached to the productions of context free grammar.

Two notations are mainly used for associating semantic rules with productions

- Syntax directed Definition
- Translation Scheme

Syntax Directed Definition (SDD)

Syntax directed definition is a generalisation of a context free grammar in which each grammar symbol either terminal or non terminal has associated set of attributes.

An attribute can be anything like a string a number, a type or a memory location etc.

Definition - A syntax directed difinition is a generalization of a context free grammar where each production $A \to \alpha$ is associated with a set of semantic rules of the form

$x := f(a_1, a_2 \dots a_k)$

where x is an attribute

attributes are associated with grammar symbols and rules are associated with productions.

these attributes are of two types

1. **Synthesized attributes**

If $A \to \alpha$ is a production and $x := f(a_1, a_2 \dots a_k)$ is a semantic rule then x is a synthesized attribute of A and $a_1, a_2 \dots a_k$ are attributes belonging to the grammar symbols of α.

We can say that synthesized attribute that can be evaluated during a single bottom up traversal of the parse tree.

Production	Semantic Rule
eg. $E \to E^{(1)} + E^{(2)}$	$\{ E.VAL := E^{(1)}.VAL + E^{(2)}.VAL \}$
$T \to F$	$\{ T.VAL := F.VAL \}$

A synthesized attribute is defined in terms of attribute values at the children of node and node itself.

135 The evaluation of the semantic rules proceed (4) along with the construction of the parse tree. It defines the values of the attributes at the nodes in the parse tree for the input string.

Annotated Parse tree – A parse tree containing the values of attributes at each node of the tree is called an annotated parse tree.

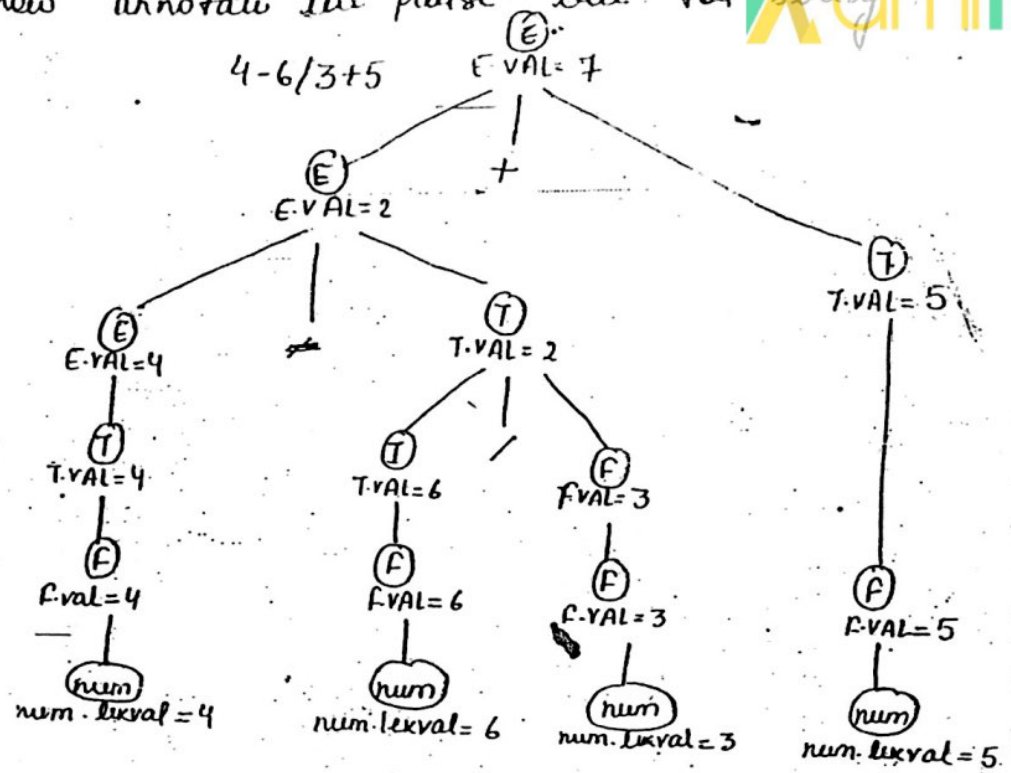
e.g.

Productions	Semantic Rules
$E \rightarrow E_1 + T$	{ E.VAL := E_1.VAL + T.VAL }
$E \rightarrow E_1 - T$	{ E.VAL := E_1.VAL − T.VAL }
$E \rightarrow T$	{ E.VAL := T.VAL }
$T \rightarrow T_1 * F$	{ T.VAL := T_1.VAL * F.VAL }
$T \rightarrow T_1 / F$	{ T.VAL := T_1.VAL / F.VAL }
$T \rightarrow F$	{ T.VAL := F.VAL }
$F \rightarrow (E)$	{ F.VAL := E.VAL }
$F \rightarrow$ num	{ F.VAL := num.lexval }

S-attributed Definition – In above syntax directed definition, every grammar symbol has synthesized attributes only so it is called S-attributed definition.

here E.VAL is called the translation

now annotate the parse tree for string



annotated Parse tree for string 4-6/3+5

2) Inherited Attributes -

If $A \to \alpha$ is a production and

$x := f(a_1, a_2 \ldots a_r)$ is a semantic rule

then x is an inherited attribute of one of the grammar symbols in α and $a_1 a_2 \ldots a_r$ are one attributes belonging to A or α.

eg. $A \to XYZ$ { Y.VAL = 2 * A.VAL }

we can say that inherited attributes are those whose initial value at a node in the parse tree is defined in terms of the attribute of the parent and or siblings

Inherited attribute at node N is defined only in terms of attribute values at N's parent, N itself and N's siblings

eg.

Production	Semantic Rules
$T \to FT'$	$T'.inh = F.val$ $T.val = T'.syn$
$T' \to *FT'$	$T_1'.inh = T'.inh \times F.val$ $T'.syn = T_1'.syn$
$T' \to \epsilon$	$T'.syn = T'.inh$
$F \to digit$	$F.val = digit.lexval$

Implementation of Syntax Directed translations

S-attribute definitions may be implemented with a bottom up parsing strategy.

Syntax directed translations can be implemented by using extra fields in the parser stack entries corresponding to the grammar symbols

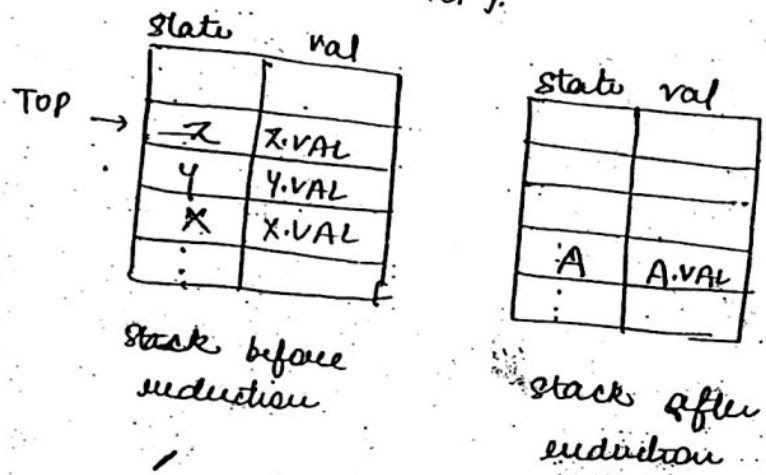
So stack is implemented by two pair of arrays - STATE and VAL

If ith STATE symbol is E then VAL[i] will hold the value of the translation E.VAL

TOP is a pointer to the current TOP of the stack.

eg. $A \rightarrow XYZ$

before the reduction of XYZ to A, the value of the translation of Z is in VAL[TOP]
and that of Y is in VAL[TOP - 1]
X is in VAL[TOP - 2]
after the reduction the value of A.VAL will appear in VAL[TOP].



Stack before reduction

Stack after reduction top = top - 2

Ex.- Let we have a syntax directed translation scheme of "desk calculator".

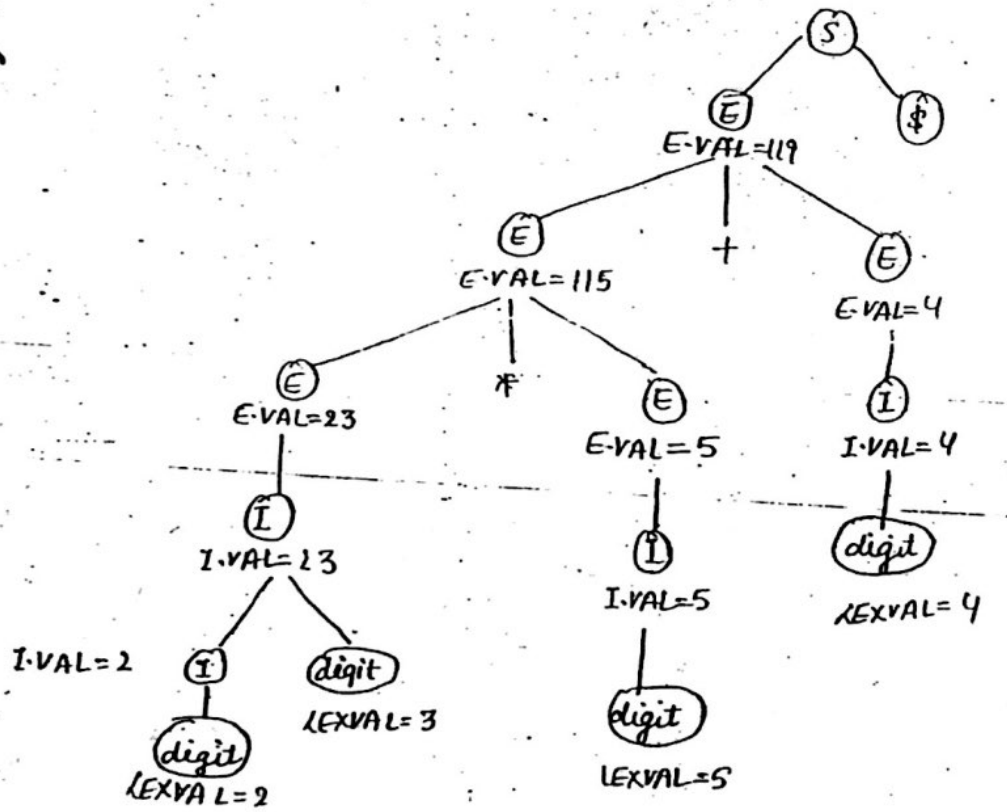
$S \rightarrow E$
$E \rightarrow E+E$
$E \rightarrow E*E$
$E \rightarrow (E)$
$E \rightarrow I$
$I \rightarrow I\,digit$
$I \rightarrow digit$

Syntax directed translation scheme for desk calculator

	Production	Semantic action
(1)	$S \to E\$$	{print E.VAL}
(2)	$E \to E^{(1)} + E^{(2)}$	$\{E.VAL := E^{(1)}.VAL + E^{(2)}.VAL\}$
(3)	$E \to E^{(1)} * E^{(2)}$	$\{E.VAL := E^{(1)}.VAL * E^{(2)}.VAL\}$
(4)	$E \to (E^{(1)})$	$\{E.VAL := E^{(1)}.VAL\}$
(5)	$E \to I$	{E.VAL := I.VAL}
(6)	$I \to I^{(1)}$ digit	{I.VAL := 10 * $I^{(1)}$.VAL + LEXVAL}
(7)	I → digit	{I.VAL := LEXVAL}

23 * 5 + 4$

Parse tree with translation



Implementation of desk calculator

	Production	Program Fragment
(1)	$S \to E\$$	print (stack[TOP-1]) val
(2)	$E \to E^{(1)} + E^{(2)}$	top = top-1 stack[TOP-2].val = stack[TOP-2].val + stack[TOP] top = top-2
(3)	$E \to E^{(1)} * E^{(2)}$	stack[top-2].val = stack[top-2].val * stack[top] top = top-2
(4)	$E \to (E^{(1)})$	stack[top-2].val = stack[top-1].val
(5)	$E \to I$	
(6)	$I \to I^{(1)}$ digit	
(7)	$I \to$ digit	VAL[TOP] = LEXVAL

Sequence of moves

	Input	state	Val	Production used
(1)	23*5+4$	—	—	
(2)	3*5+4$	2	2	
(3)	3*5+4$	I	2	I → digit
(4)	*5+4$	I3	2_	
(5)	*5+4$	I	23	I → I digit
(6)	*5+4$	E*	23$	E → I
(7)	5+4$	E*	23_	
(8)	+4$	E*5	23__	
(9)	+4$	E*I	23_5	I → digit
(10)	+4$	E*E	23_5	E → I
(11)	+4$	E	115	E → E*E
(12)	4$	E+	115_	
(13)	$	E+4	115__	
(14)	$	E+I	115_4	I → digit
(15)	$	E+E	115_4	E → I
(16)	$	E	119	E → E+E
(17)		E$		

Implementation of desk calculator

Productions	Program Fragment
(1) $S \rightarrow E\$$	print (stack[top-1].val)
(2) $E \rightarrow E^{(1)} + E^{(2)}$	stack[top-2].val = stack[top-2]. val + stack[
(3) $E \rightarrow E^{(1)} * E^{(2)}$	stack[top-2]. val = stack [top-2].val + stack[top-1
(4) $E \rightarrow (E^{(1)})$	stack [top-2]. val = stack [top-1] . val
(5) $E \rightarrow I$	
(6) $I \rightarrow I^{(1)}$ digit	stack[top-1].val = 10 * stack[top-1] + stack[top
(7) $I \rightarrow$ digit	

$E^{(2)}$
+
$E^{(1)}$

Q Give the parse tree and translations for the expression (4*7+19)*2 by following grammar

$S \to E\$$
$E \to E+E$
$E \to E*E$
$E \to (E)$
$E \to I$
$I \to I$ digit
$I \to$ digit

Syntax directed translation scheme -

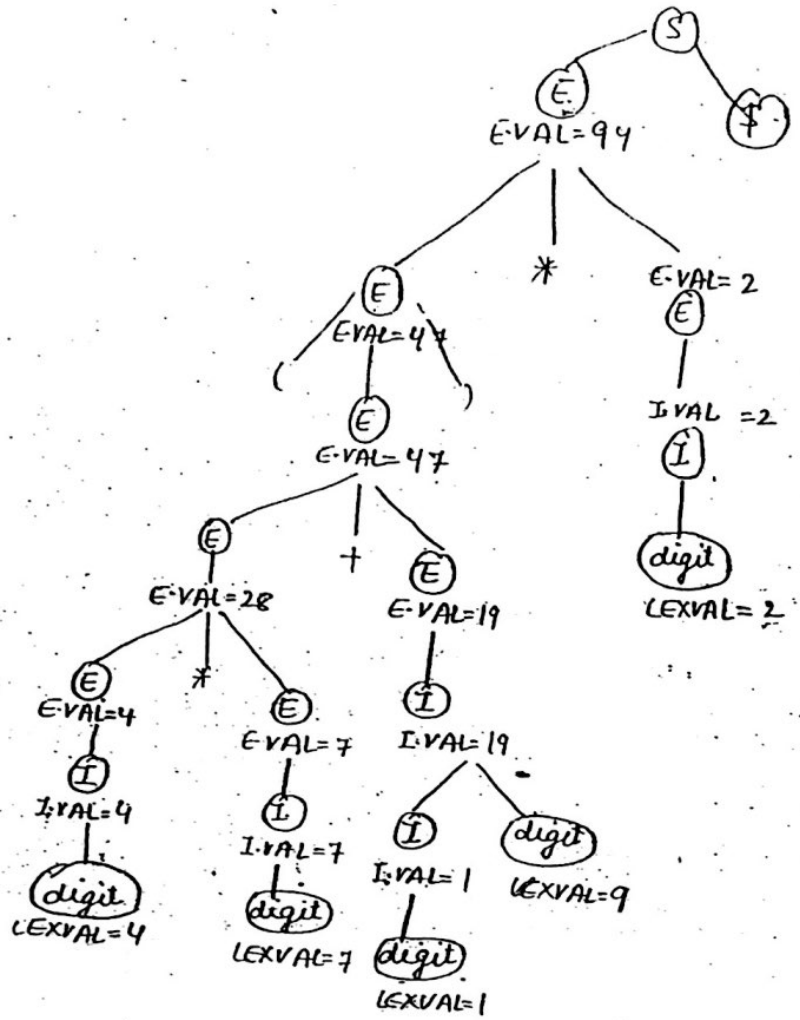
$S \to E\$$	{print E.VAL}
$E \to E^{(1)} + E^{(2)}$	$\{E.VAL := E^{(1)}.VAL + E^{(2)}.VAL\}$
$E \to E^{(1)} * E^{(2)}$	$\{E.VAL := E^{(1)}.VAL + E^{(2)}.VAL\}$
$E \to (E^{(1)})$	$\{E.VAL := E^{(1)}.VAL\}$
$E \to I$	$\{E.VAL := I.VAL\}$
$I \to I^{(1)}$ digit	$\{I.VAL := I^{(1)}.VAL + LEXVAL\}$
$I \to$ digit	$\{I.VAL := LEXVAL\}$

Implementation

Program fragment.

$S \to E\$$	print stack[TOP]
$E \to E+E$	stack[TOP] = stack[top-2] + stack[top].val
$E \to E*E$	stack(top-2).val = stack(top-2).val + stack(top-1).val
$E \to (E)$	stack(top-2).val = stack(top-1).val
$E \to I$	
$I \to I$ digit	stack(top-1).val = stack(top-1).val * 10 + stack(top)
$I \to$ digit	

Parse tree → (4*7+19)*2$

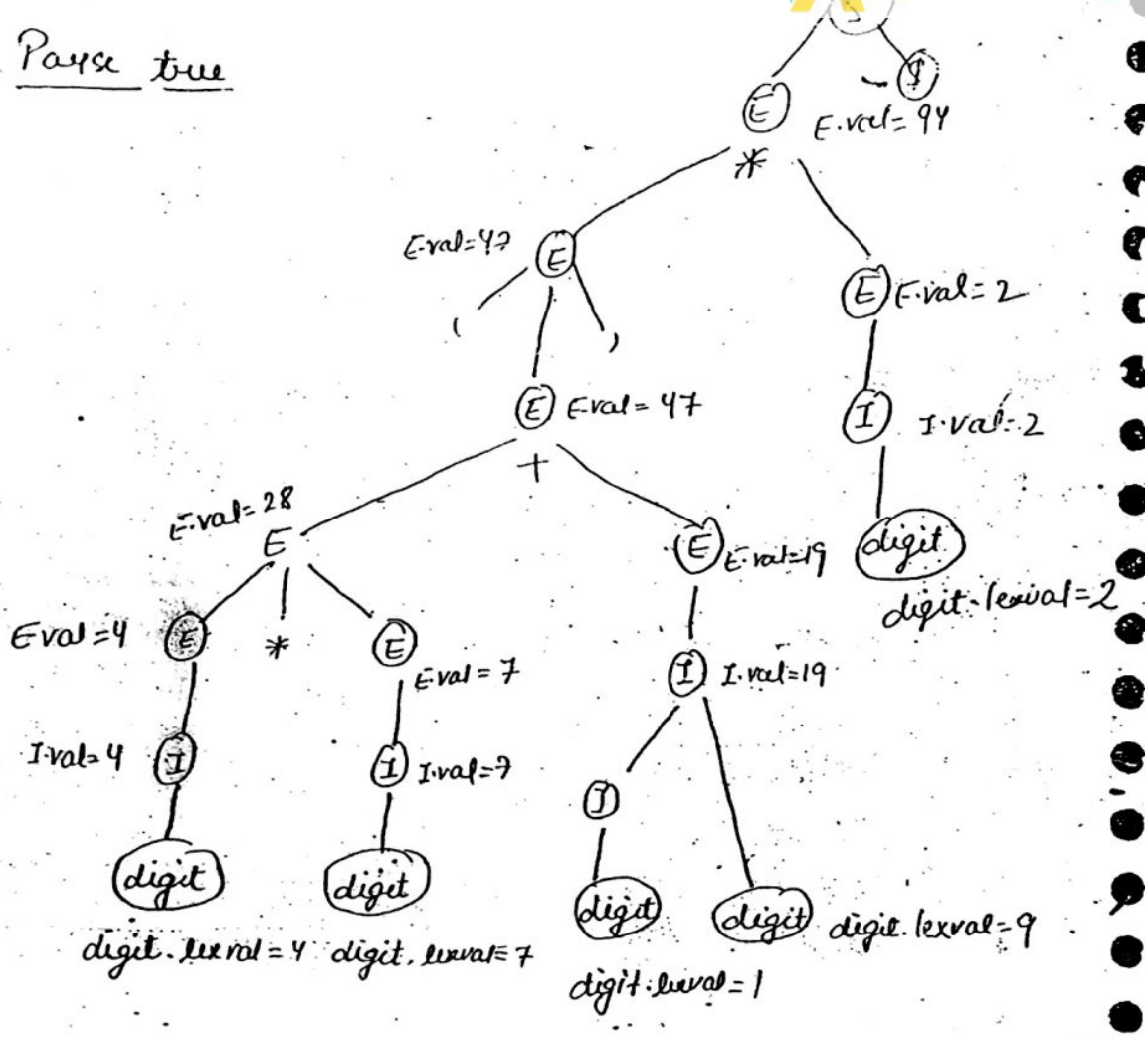


Sequence of moves

	Input	State	value	Productions used
1)	(4*7+19)*2$	(		
2)	*7+19)*2$	(4		
3)	*7+19)*2$	(I	4	I → digit
4)	*7+19)*2$	(E	4	E → I
5)	7+19)*2$	(E*	4	
6)	+19)*2$	(E*7	4	
7)	+19)*2$	(E*I	4-7	I → digit
8)	+19)*2$	(E*E	4-7	E → I
9)	+19)*2$	(E	28	E → E*E
10)	19)*2$	(E+	28	
11)	9)*2$	(E+1	28--	
12)	9)*2$	(E+I	28-1	E → digit
13)	)*2$	(E+I9	28-19	I → I digit
14)	)*2$	(E+I	28-19	E → I
15)	)*2$	(E+E	47	E → E+E
16)	)*2$	(E	47	
17)	*2$	(E)	47	E → (E)
18)	*2$	E	47	
19)	2$	E*	47--	
20)	$	E*2	47-2	I → digit
21)	$	E*I	47-2	E → I
22)	$	E*E	94	E → E*E
23)	$	E	94	
24)	-	E$		
25)		E		

(4*7+19)*2$

Parse tree



Intermediate code - In most of the compilers source code is translated into a language which is intermediate in complexity between a high level programming language and machine code. Such code is called intermediate code or intermediate text.

Most common representations of source code are -

- postfix notation
- Syntax trees
- quadruples
- Triples -

Postfix Notation → postfix expressions contain operators at their right end.

eg. for $e_1 \oplus e_2$ infix expression

postfix expression is $e_1 e_2 \oplus$

eg. $(a+b) * c$

$ab+c*$ is postfix expression for above infix expression

eg. If e then x else y

by using ? ternary postfix operator

postfix form for above is exy ?

eg. If a then if c-d then a+c else a*c else a+b

postfix for above if expression is

acd-ac+ac*?ab+?

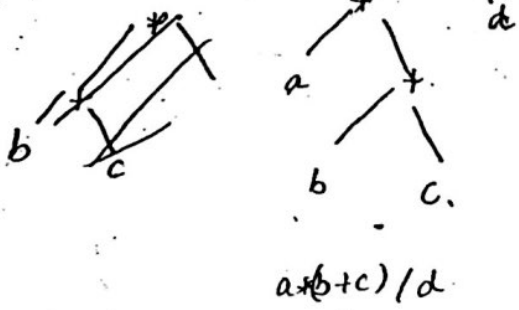
Syntax directed translation to postfix code

Production	Semantic Action
$E \rightarrow E^{(1)}$ op $E^{(2)}$	E.code = $E^{(1)}$.code || $E^{(2)}$.code || op
$E \rightarrow (E^{(1)})$	E.code = $E^{(1)}$.code
$E \rightarrow$ id	E.code = id

SYNTAX TREES

Syntax tree is basically a variant of parse tree in which each leaf represents an operand and each interior node represents an operator.

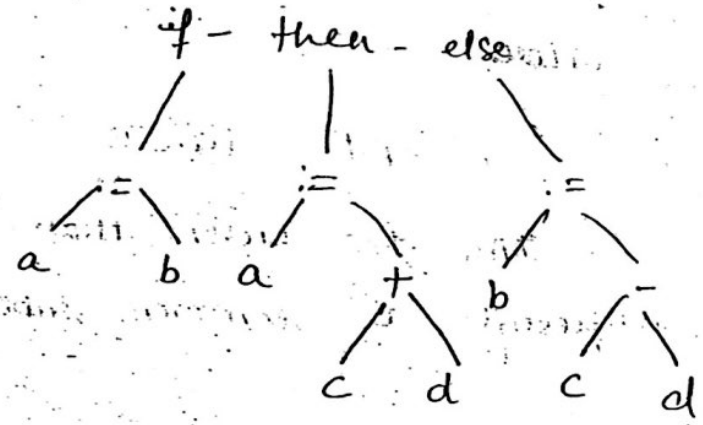
eg. Syntax tree for expression a*(b+c)/d is



a*(b+c)/d

Ex: if a=b then a := c+d else b := c-d (2)

Syntax tree



Syntax directed translation scheme to construct syntax tree

	Semantic actions
$E \rightarrow E^{(1)} + T$	E.Node := newNode ('+', $E^{(1)}$.node, T.node)
$E \rightarrow E^{(1)} - T$	E.Node := newNode ('-', $E^{(1)}$.node, T.node)
$E \rightarrow T$	E.Node := T.Node
$T \rightarrow (E)$	T.Node := E.Node
$T \rightarrow id$	T.Node := newLeaf (id, id.entry)
$\rightarrow num$	T.Node := newLeaf (num, num.val)
$\rightarrow -E^{(1)}$	E.node := UNARY ('-', $E^{(1)}$.node)

Directed ~~Acyclic~~ Acyclic graph for expressions

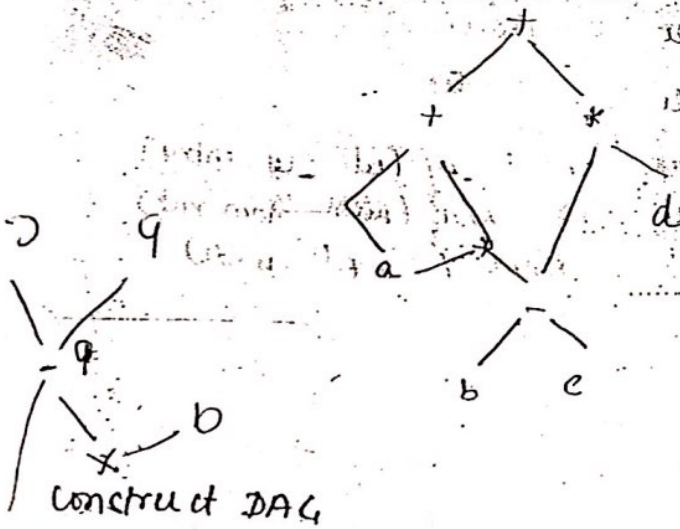
DAG is similar to syntax trees having leaves corresponding to operands and interior nodes corresponding to operators.

Difference b/w DAG and syntax tree is that a node N in a DAG has more than one parent if N represents a common subexpression in a syntax tree.

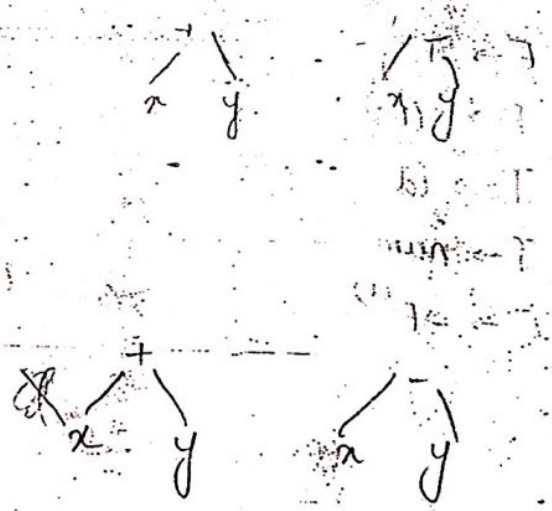
means DAG represents the generation of efficient code to evaluate the expressions.

$a + a * (b - c) + (b - c) * d$

its DAG



construct DAG



$((x+y) - ((x+y) * (x-y))) + ((x+y) * (x-y))$

Three address code -

Three address code is a linearized representation of a syntax tree or a DAG in which explicit names corresponding to the interior nodes of the graph are used.

Three address code is a sequence of statements of the form. $A = B$ op C. Each statement generally contains 3 addresses two for operands and one for the result. Each statement does not involve not more than three addresses, therefore it is called three address statement, and sequence of such statements is referred as <u>three address code</u>.

eg. three address code for expression

$a+b*c+d$ is

$T_1 := b*c$

$T_2 := a+T_1$

$T_3 = T_2+d$

---

here T_1, T_2 and T_3 are temporary names which are generated by the compiler

* Some common forms of Three address Statements

1) Arithmetic or logical statement

$$A = B \text{ op } C$$

2) Assignment instructions of the form

$$A := \text{op } B$$

where op is a unary operation like unary minus, logical negation, shift operators and conversion operators etc.

eg. ~~$T_1 := \text{inttofloat}(T_2)$~~

$T_1 := \text{int to float } T_2$

3) Copy statements of the form

$$A := B$$

where B is assigned to the value of A.

4) Unconditional jump statement

Goto L

means the three address statement with label L is next to be executed.

5) Conditional jump statement

if A relop B goto L

means if A relop B is true statement with label L is executed else the next statement in the normal flow sequence is executed

3) conditional jump statement

— if a goto L

means if a is non zero then statement with label is executed and if a is false means zero or else then next statement in normal flow sequence executed.

Procedure calls and return statement

for parameters param x

in procedure and function call p, n

calls or y = call p, n

for returning return y.

eg. p(x1, x2, ... xn)

three address → param x1
param x2
.....
param xn
call p, n

Indexed assignment statement

x = y[i]
x[i] = y

Address and pointer assignment

x = *y
*x = y

Implementations of Three address statement

A three address code is an abstract form of intermediate code. These statements can be implemented as record with fields for the operator and operands. Three such representations are-

Quadruple

Triples

Indirect Triple

Quadruple - Quadruple is a record structure using four fields-

OP	ARG1	ARG2	RESULT

OP field contains an internal code for the operator and ARG1, ARG2, RESULT are pointer to the symbol table entries. Three address statement,

A = B OP C is represented by using B in ARG1
C in ARG2
A in RESULT

A = OP B

B in ARG1
nothing in ARG2
A in RESULT

→ Param use neither ARG 2 nor RESULT

→ Conditional and unconditional jumps put the target label in Result field.

Ex A := -B * (C+D)

	OP	ARG1	ARG2	RESULT
(0)	unminus	B	-	T_1
(1)	+	C	D	T_2
(2)	*	T_1	T_2	T_3
(3)	:=	T_3		A

three address code is

$T_1 := -B$

$T_2 := C+D$

$T_3 := T_1 * T_2$

$A := T_3$

$T_1 \Rightarrow -B$

$T_2 = C+D$

$T_3 = T_1 * T_2$

$A = T_3$

To avoid entering temporary names into the symbol table. temporary value can be referred by the position of the statement that computes it.

So, in this representation, three address statements can be represented by record structure with only three fields

OP	ARG1	ARG2

where ARG1, ARG2 are either pointers to the symbol table or pointers into the structure itself

Eg. A := −B * (C+D)

Three address code

T1 := −B
T2 := C+D
T3 := T1 * T2
A := T3

	OP	ARG1	ARG2
(0)	uminus	T1	
(1)	+	C	D
(2)	*	(0)	(1)
(3)	:=	A	(2)

Indirect Triples - consists of a listing of pointers to triples rather than a listing of triples themselves in desired order

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

	statement
(0)	(9)
(1)	(10)
(2)	(11)
(3)	(12)

	OP	ARG1	ARG2
(9)	unminus	B	-
(10)	+	C	D
(11)	*	(9)	(10)
(12)	:=	A	(11)

Indirect Triple representation

Comparision of quadruples Triples and Indirect Triples

Quadruples

- ↳ direct access of the location for temporaries
- ↳ easier for optimization

Triples

- Space efficiency

Indirect Triples

- ↳ easier for optimization
- ↳ space efficiency

some examples of generating three address code

→ Assignment statements of integer types.

eg. $a = b + c + d$

three address code is

$T_1 := b + c$

$T_2 := T_1 + d$

$a := T_2$

→ Assignment statements with mixed types

eg. $X = Y + I * J$

where x and y are real and I & J are of integer types.

$T_1 := I * J$

$T_2 := \text{int to float } T_1$

$T_3 := Y + T_2$

$X := T_3$

eg. $X := Y * Z + Y * R$

where R, y are integers and x, z are of type float.

$T_1 := \text{int to float } y$

$T_2 := T_1 * Z$

$T_3 := y * R$

$T_4 := \text{int to float } T_3$

$T_5 := T_2 + T_4$

$X := T_5$

Boolean expressions

For generating the three address code for Boolean expressions, some branching statements of the following forms are used.

goto L

if A goto L

if A relop B goto L

eg.1 three address code for expression A or B and C

$T_1 := B$ and C

$T_2 := A$ or T_1

eg:2 A<B

is similar to if A<B goto then 1 else 0

three address code is

```
    if A<B goto L1
    T=0
    goto L2
L1: T=1
L2:
```

or

```
(1) if A<B goto (4)
(2) T=0
(3) goto (5)
(4) T=1
(5)
```

eg 3 If A<B or C

(1) If A<B goto (4)
(2) $T_1 := 0$
(3) goto (5)
(4) $T_1 := 1$
(5) $T_2 := T_1$ or C

4 If (A<B || C<D) X=Y+Z

If A<B goto

4 If a>b then x = a+b

If a > b goto L_1
goto L_2
L_1: $T_1 := a+b$
$X := T_1$
L_2:

5 If a>b then x = a+b
else x = a-b

If a>b goto L_1
goto L_2

L_1: $T_1 := a+b$
$X := T_1$
goto L_3
L_2: $T_2 := a-b$
$X := T_2$
L_3:

Three address code for Boolean Expressions

eg.1 A or B and C

(TAC) three address code is

$T_1 := B$ and C

$T_2 := A$ or T_1

eg.2 A<B

it is similar to if A<B then 1
else 0

TAC is

if A<B goto L_1

T=0

goto L_2

L_1: T=1

L_2: -

eg.3 if A<B or C

TAC is

if A<B goto L_1

T_1=0

goto L_2

L_1: T_1=1

L_2: T_2= T_1 or C

L_3:

Three address code for array reference

Relative location of a[i] is

base + i × w

Relative location of a[i][j]

is base + [i × n2 + j] × w

eg 1 three address code for x = a[i][j] where a is a 2×3 array, where size of each element is 4 bytes

location of a[i][j] can be represented by

base + [i × n2 + j] × w

= 0 + [i × 3 + j] × 4 = 12i + 4j

T1 = 12 * i

T2 = 4 * j

T3 = T1 + T2

x = a[T3]

= Ans

write address code for looping constructs with statements

1. while a > b do a = a + b

```
L1   If a > b goto L2
     goto L3
L2:  T1 := a + b
     a := T1
     goto L1
L3
```

on statements

```
a = 3
b = 4
for (i=0; i<n; i++)
{ B  a = b + 1;
     a = a * a;
} c = a;
```

TAC is

```
a = 3
b = 4
i = 0
L1: if i<n goto L2
    goto L4
L2: T1 := b + 1
    a = T1
    T2 = a * a
    a = T2
    goto L3
L3: T3 = i + 1
    i = T3
    goto L1
L4: T4 := a
    c = T4
```

eg.2

```
a = 0
b = 2
for (i=1; i<n; i++)
{ a = a+b * x[i] + y[i]
}
```

TAC is

```
a = 0
b = 2
i = 1
L1: if i<n goto L2
    goto L4
L2: T1 := a+b
    T2 := i*4
    T3 := x[T2]
    T4 := y[T2]
    T5 := T3+T4
    T6 := T1+T5
    a := T6
    goto L3
L3: T7 := i+1
    i = T7
    goto L1
L4:
```

Q1 Generate Quadruple Representation of

$a = b * -c + b * -c$

Three address code by syntax tree representation.

$T_1 := -c$
$T_2 := b * T_1$
$T_3 := -c$
$T_4 := b * T_3$
$T_5 := T_2 + T_4$
$a = T_5$

	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

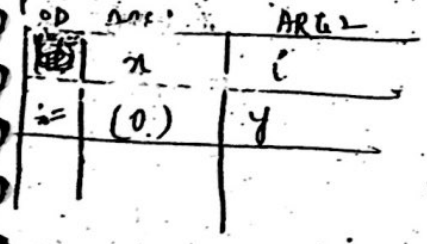
Three address code by DAG representation

$T_1 := -c$
$T_2 := b * T_1$
$T_3 := T_2 + T_2$
$a := T_3$

	OP	ARG1	ARG2	RESULT
(1)	uminus	c	–	T_1
(2)	*	b	T_1	T_2
(3)	+	T_2	T_2	T_3
(4)	:=	T_3	———	a

Q $x[i] = y$

Syntax tree representation



$x = y[i]$

	OP	ARG1	ARG2
(0)	[] =	y	i
(1)	:=	x	(0)

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	:=	a	(3)

Using syntax Tree

	OP	ARG1	ARG2
(0)	uminus	c	–
(1)	*	b	(0)
(2)	+	(1)	(1)
(3)	:=	a	(2)

Using DAG

Generate quadruple representation of

$x = a + b * c + c * d * e$

Three address code is

```
T1 := b * c
T2 := a + T1
T3 := c * d
T4 := T3 * e
T5 := T2 + T4
x := T5
```

	OP	ARG1	ARG2	RESULT
(0)	*	b	c	T1
(1)	+	a	T1	T2
(2)	*	c	d	T3
(3)	*	T3	e	T4
(4)	+	T2	T4	T5
(5)	:=	T5	–	x

Q. Generate indirect triple representation for $x := a+b*c+c*d*e$ and $y+a+b*c$

Triple representation

	OP	ARG1	ARG2
(10)	*	b	c
(11)	+	a	(10)
(12)	*	c	d
(13)	*	(12)	e
(14)	+	(11)	(13)
(15)	:=	x	(14)
16	+	y	(11)

Indirect Triple Representation

statement table for expression $x := a+b*c+c*d*e$

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

statement table for expression $y+a+b*c$

0	(10)
1	(11)
2	(16)

Generate three address code for

$X := y + I * J$

where X and y are real and
I & J are integers

$T_1 := I * J$
$T_2 := \text{int to float } T_1$
$T_3 := y + T_2$
$x := T_3$

Q Generate three address code for

$x := y * z + y * x$

where x, y are integers &
x, z are float

$T_1 := \text{int to float } y$
$T_2 := T_1 * z$
$T_3 := y * x$
$T_4 := \text{int to float } T_3$
$T_5 := T_2 + T_4$
$x := T_5$

1. If A<B or C

TAC

```
If A<B goto L1
T=0
goto L2
L1: T=1
L2: T or C
L3:
```

2. If A<B then X= a+b

```
If A<B goto L1
goto L2
L1: T1:= a+b
    X:= T1
L2:
```

3. If a<b then x=a+b
   else x= a-b

```
If a<b goto L1
goto L2
L1: T1:=a+b
    x:= T1
    goto L3
L2: T2:= a-b
    X:=T1
L3:
```

Q.4 . a || b && c < d

```
if c<d goto L1
   T1 = 0
   goto L2
L1: T1 = 1
L2: T2 := b && T1
    T3 := a || T2
L3:
```

Q.5 – a<b && c<d

```
if a<b goto L1
goto L2
L1: if c<d goto L3
L2: T=0 goto L4
L3: T=1
L4: - - -
```

Q.6 — if (x<100 || x>200 && x != y) x=0

```
if x<100 goto L1
L2: if x>200 goto L3
goto L2
L3: if x != y goto L1
    goto L2
L1: X=0
L2:
```

Q7. If a<b then x = a+b
else if a=b then x=a
else x=b

TAC is

If a<b goto L1
goto L2

L1: T1 := a+b
x := T1
goto L3

L2: If a=b goto L4
goto L5

L4: t1 := a
x := t1
goto L3

L5: t1 := b
x := t1

L3:

Q8 If (a<b || c<d) x=y+z

TAC is

If a<b goto L1
If c<d goto L1
goto L2
L1: T1 = y+z
x := T
L2:

Generate three address code for

$x = c + a[i][j]$

where a is a 2×3 array and size of each element is 4 bytes.

Sol: a[i][j] can be represented by

base + (i × n2 + j) × w

= 0 + (i × 3 + j) × 4

= 12i + 4j

So TAC is

$T_1 := 12 * i$

$T_2 := 4 * j$

$T_3 := T_1 + T_2$

$T_4 := a[T_3]$

$T_5 := c + T_4$

$x := T_5$

Generate three address code for

$x[i, j] = y$.

where x is 5×10 array and size of each element is 2 bytes.

x[i, j] can be represented by

base + (i × n2 + j) × w

$0 + (i \times 10 + j) \times 2 = 20i + 2j$

i.e.

$T_1 := 20 * i$

$T_2 := 2 * j$

$T_3 := T_1 + T_2$

2) while a>b do a=a+b

L1: if a>b goto L2
goto L3

L2: T1 = a+b
a = T1
goto L1

L3:

a=3
b=4

```
for (i=0; i<n; i++)
{ a= b+1;
  a= a*a;
} c=a
```

TAC 4

a=3
b=4
i=0

L4: if i<n goto L2
goto L4

L2: T1 := b+1
a = T1
T2 = a*a
a = T2
goto L3

L3: T3 := i+1
i = T3
goto L1

L4: T4 = a
c = T4

Q

```
a=0
b=2
for(i=1; i<n; i++)
{ a= a+b + x[i]+y[i]
}
```

TAC is

```
a=0
b=2
i=1
L4: if i<n goto L2
    goto L4
L2: T1:= a+b
    T2:= i*4
    T3:= x[t2]
    T4:= y[T2]
    T5:= T3+T4
    T6= T1+T5
    a= T6
    goto L3
L3: T7:= i+1
    i= T7
    goto L1
L4:
```

Q

```
do i+1
while (i<n);
```

TAC is

```
L1: T1:= i+1
    i=T1
    if i<n goto L
L2:
```

do i+1
while (a[i] < v)

L1: T1 := i+1
i = T1

T2 := i*4
T3 := a[T2]
if T3 < v goto L1

L2:

n = f(a[i]);

T1 := i*4
T2 := a[T1]
param T2
t3 = call f, 1
n = t3

Q

```
int i;
i=1;
while a<10 do
    if x>y then a= x+y
    else a= x-y
```

TAC

```
    t1=1
    i=t1
L1: if a<10 goto L2
    goto L4

L2: if x>y goto L3
    T2:= x-y
    a= T2 goto L1
L3: T1:= x+y
    a= T1 goto L1
L4: ;
```

Q

```
int c; int p; int i=1;
c= i
p= i
for (i=2; i<=n; i++)
{ c= c+p;
  p= c-p;
}
```

TAC

```
    T1=1
    i= T1
    T2= i
    C= T2
    p= T2
    T3= 2
    i= T3
L1: T4= n
    i= T4
    T5= i
    if T5 goto L2
    goto L3
L2: T6= C+p
    C= T6
    T7= C-p
    p= T7
    goto L3
L3: T8= i+1
    i= T8
    goto L1
```

Q while (a<c && b<d) do
if a=1 then c=c++
else
while a≤d do
a= a+3

L1: if a<c goto L2
goto L5
L2: if b<d goto L3
goto L5
L3: if a≤d goto L4
L4: T1 = a+3
a= T1 goto L3
L5:

Q P(A+B, C)

P is a procedure

do while statement

```
do i+1
while (i<n);
```

```
L1: T1=i+1
    i=T1
    if i<n goto L1
    goto L2
L2:
```

```
do i+1
while (a[i]<v)
```

```
L1: T1:= i+1
    i=T1
    T2:= i*4
    T3:= a[T2]
    if T3<v goto L1
    goto L2
L2:
```

TAC for Procedures

n = f(a[i]);

TAC is

T1 := i * 4
T2 := a[T1]
param T2
t3 = call f, 1
n = t3

fun(x1, x2 .. xn)

TAC is

param x1
param x2
:
param xn
T3 := call fun, n

Q

```
sum = 0;
for (i=1; i<=20; i++)
    sum = sum + a[i] + b[i];
```

Ans

```
T1 = 0
sum = T1
T2 = 1
i = T2
L1: if i<=20 goto L2
    goto L4
L2: T3 = 4*i
    T4 = a[T3]
    T5 = b[T3]
    T6 = T4 + T5
    T7 = sum
    T8 = T6 + T7
    sum = T8
    goto L3
L3: T9 = i+1
    i = T9
    goto L1
L4: — — — —
```

```
int b
int a;
b=3
a=12;
a=(b+2)-(a*3)/6
```

TAC

```
t1 = 3
b = t1
t2 = 12
a = t2
t3 = b+2
t4 = a*3
t5 = t4/6
t6 = t4/t5
t7 = t3+t6
a = t7
```

Q

```
void main
{ int a;
  a=23
  if (a==23)
     a=10
  else
     a=19
}
```

TAC →

```
Begin func
t0=23;
a=t0;
if (a==23) goto L1
   T1=19
   a=T1 goto L2
L1: T1=10;
   a=T1
L2: ....
```

switch A+B
begin
case 2: x:=4;
case 5:
switch x
begin
case 0: A:= B+1
case 1: A:= B+3
default: A:= 2
end
case 9: x= 4-1
default: x= 4+1
end

switch A+B
begin
case v_1: x=a
case v_2: x=b
default: x=c
end

T_1: = A+B
if T_1 = v_1 goto L_1
if T_1 = v_2 goto L_2
if

T_1 = A+B
if T_1 == v_1 goto L_1
goto L5
if T_1 == v_2 goto L_2
goto L5
L_4: x=a
L_2: x=b
L_5: x=c

T_1 = A+B
if T_1 == 2 goto L_1
goto L5
if T_1 == 5 goto L_2
goto L5
if T_1 == 9 goto L9
L_1: T_2 = 4
X = T_2
goto NEXT
L_2: T_3 = X goto NEXT
if T_3 == 0 goto L
goto L6
if T_3 == 1 goto L
goto L6
L4: A = B+1 goto NEX
L7: A = B+3 goto N
L6: A = 2 goto NEX
L5: x = 4+1
goto NEXT
L9: x = 4-1
NEXT

Code optimization

Code optimization refers to the techniques used by the compiler to improve the execution efficiency of the generated object code.

Code optimization includes various transformations on intermediate code, but these transformations must preserve the semantics of the program.

Criteria to apply optimizing transformations-

1) The optimization should capture most of the potential improvements without an unreasonable amount of effort

2) Optimization should be such that the meaning of the source program is preserved.

3) The optimization should on average, reduce the time and space expended by the object code.

Code optimization can be done in three ways-

1) Local optimization

which performed in a straight line (basic block of code means a block with no jumps in ~~exit~~ except at the beginning and no jumps except at the end

2) Loop optimization

3) Data flow analysis

↳ transmission of useful information from all parts of the program to the places where the information can be of use.

BASIC BLOCKS AND FLOW GRAPHS

A graph representation of three address statements called a flow graph. Nodes in flow graph represent computations and the edges represent the flow of control. Flow graphs can be used to collect information about the intermediate program and to find inner loops where a program is expected to spend most of its time.

BASIC BLOCK

Basic block is a sequence of consequetive statements in which

a) flow of control can only enters at the beginning. No jumps are allowed into the middle of the block.

b) flow of control can leave the block at the end without halt or possibility of branching except at the end

Algorithm for the construction of Basic Blocks

Step 1 - First determine the set of leaders, the first statements of basic blocks.

Rules for finding leaders are -

a) First three address statement in intermediate code is a leader.

b) Any statements that immidiately follows a conditional or unconditional jump (goto) is a leader.

c) Any statement that is the target of a conditional or unconditional jump is a leader

Step 2 - For each leader, its basic block consists of all instructions upto the next start of a block.

Example ①

```
t1 = a * a
t2 = a * b
t3 = 2 * t2
t4 = t1 + t3
t5 = b * b
t6 = t4 + t5
```

these sequence of statements forms a basic block.

Ex-1

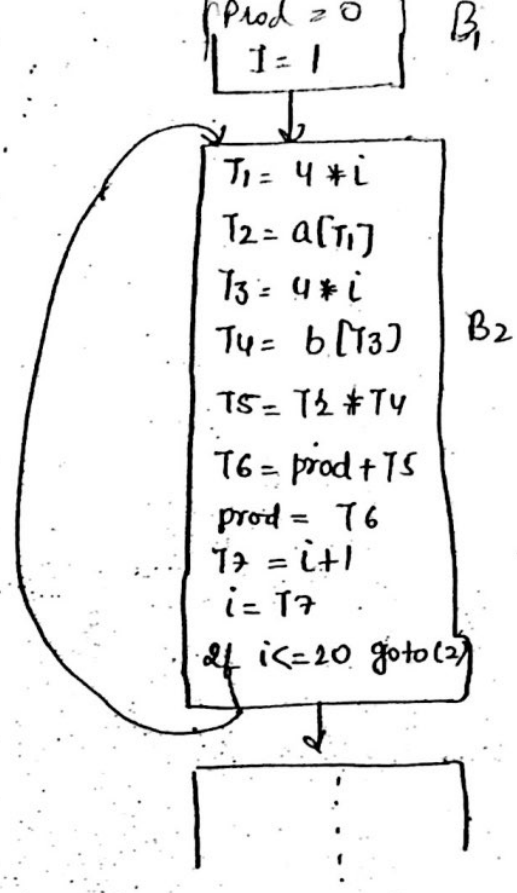
```
begin
 prod=0
 i:=1
 do begin
 prod = prod + a[i] * b[i]
 i= i+1;
 end
 while i<=20
end
```

Three address code :-

```
(0)  Prod=0          — leader
(1)  i:=1
(2)  t1:=4*i         — leader
(3)  t2 = a[t1]
(4)  t3 = 4*i
(5)  t4 = b[t3]
(6)  t5 = t2*t4
(7)  t6 = prod + t5
(8)  prod = t6
(9)  t7 = i+1
(10) i = t7
(11) if i<=20 goto (2)
(12)
```

Sequence of Basic Blocks.

Flow graph :- The relationship among basic blocks is represented by a directed graph called a flow graph. There is a directed edge from block B_1 to B_2 if B_2 immidiately follows B_1 during execution. we can say B_1 is a predicessor of B_2 & B_2 is a succesor of B_1.



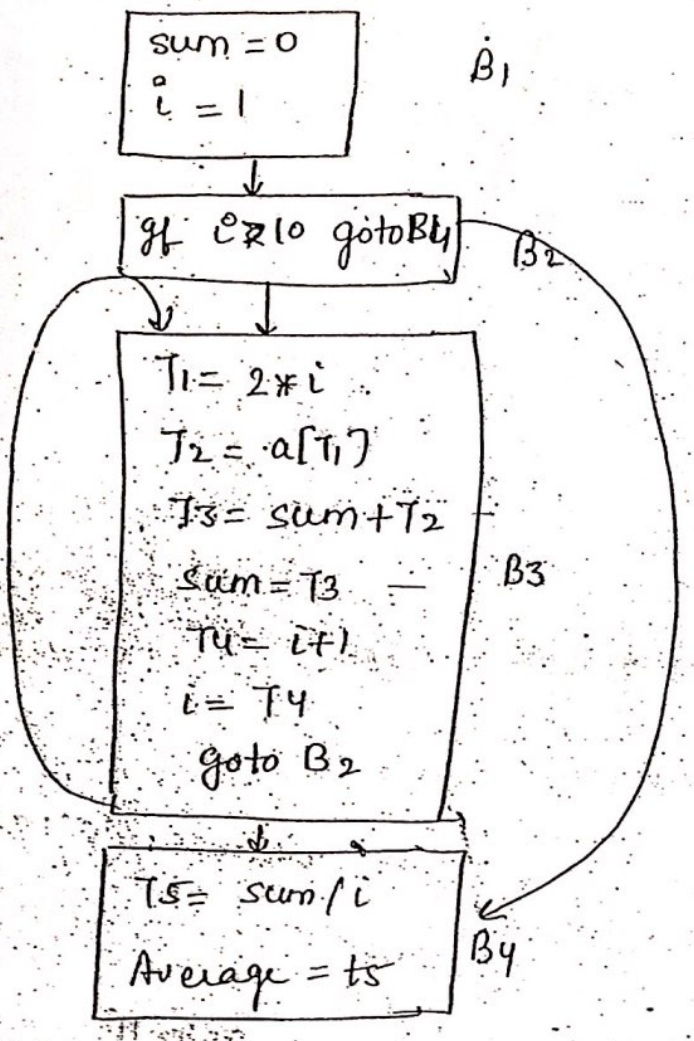
Ex-3

```
sum=0
i=1;
while i<=10 do
{
  sum= sum + a[2*i]
  i = i+1
}
average = sum/i
```

Three address statement

(1) sum=0
(2) i=1
(3) If i>10 goto 11
(4) $T_1 = 2 * i$
(5) $T_2 = a[T_1]$
(6) $T_3 = sum + T_2$
(7) sum = T_3
(8) $t_4 = i + 1$
(9) i = T_4
(10) goto(3)
(11) $T_5 = sum / i$
(12) average = T_5

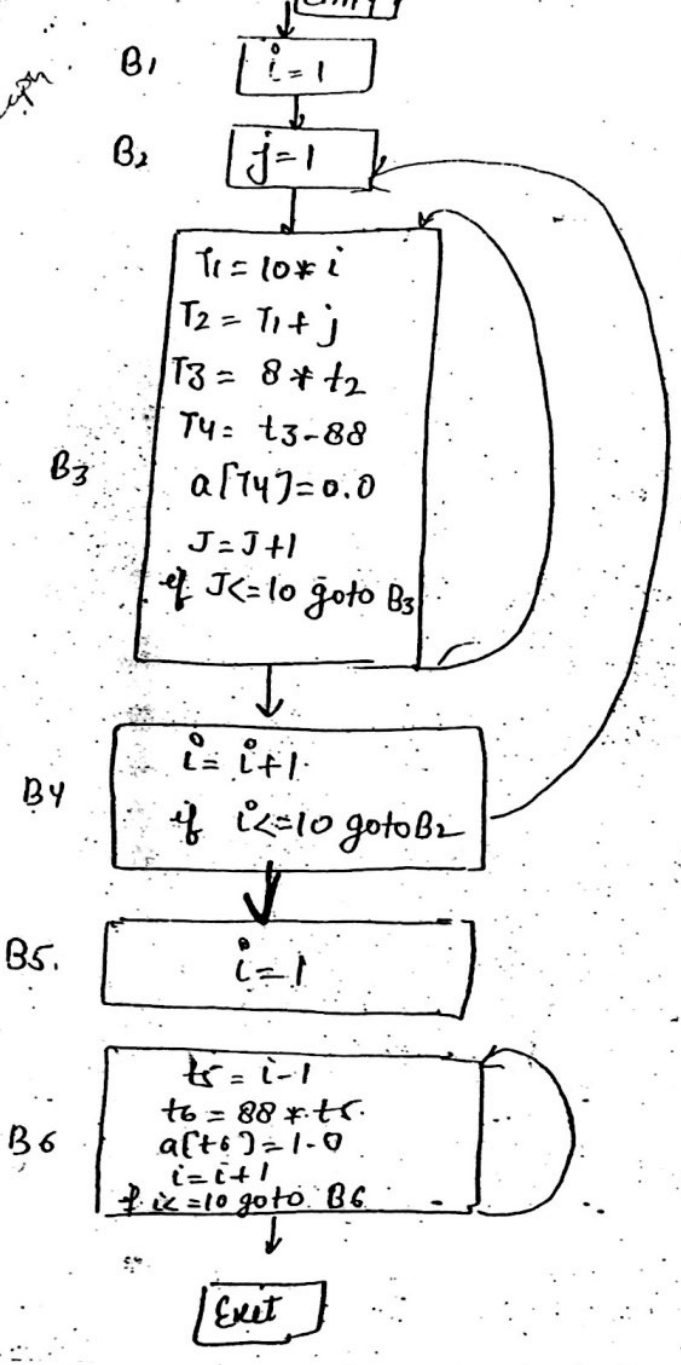
Flow graph



Ex-4 Consider the following three address code and find basic blocks.

(1) i = 1
(2) j = 1
(3) t1 = 10 * i
(4) t2 = t1 * j
(5) t3 = 8 * t2
(6) t4 = t3 - 88
(7) a[t4] = 0.0
(8) j = j + 1
(9) if j <= 10 goto (3)
(10) i = i + 1
(11) if i <= 10 goto (2)
(12) i = 1
(13) t5 = i - 1
(14) t6 = 88 * t5
(15) a[t6] = 1.0
(16) i = i + 1
(17) if i <= 10 goto (13)

Blocks & flow graph



Local Optimization

A basic block computes a set of expressions. Two basic blocks are said to be equivalent if they compute the same set of expressions.

optimization can be achieved by applying a no. of transformations within each basic block.

Two classes of local transformations can be applied to basic blocks –

1) Structure Preserving transformations
   - Common sub expression elimination
   - Dead code elimination
   - ~~Renaming temporary variables~~ Copy Propagation
   - ~~Interchange of statements~~ Constant folding
2) Algebric transformations

DAG representation of Basic Blocks

DAG representation of Basic Blocks

DAG (directed acyclic graph) is a useful data structure for analysing basic blocks. Transforming basic blocks into DAG helps in various techniques of local optimization.

Application of DAG's representation

Using DAG we can perform following actions

1) We can eliminate local common subexpressions, ie instructions that compute a value that has already been computed.

2) We can eliminate dead code, statements that compute a value that is never used.

3) We can reorder statements that do not depend on one another. This reordering may reduce the time a temporary value needs to be preserved in a register.

4) We can apply algebraic laws to simplify the computation.

Construction of DAG

We have three types of statements

(i) A = B op C (ii) A = op B (iii) A = B

If i<=2 goto will be treated as case (i) with A undefined.

1) Construct node in the DAG for each initial values of the variables appearing in the basic block.

2) For statement (i) check if we have a node labeled op with whose left child is node B and right child is node C. If not then create such node. Let n is the node.

3) For statement (ii) check if we have a node labeled op with lone child node B. If not then create such node. Let n is the node.

4) In case (iii) make n as node (B)

5) Attach A to node N.

6) An assignment from an array like x = a[i] is represented by creating a node with operator =[] with two children a & variable x becomes the label of the new node.

7) For statement like a[j] = y replaced by new node with operator []= and three children representing a, j & y. It has no variable labeling this node. Creation of this node kills currently constructed nodes whose value depends on a. This node will not receive any labels.

Local optimization (Local optimization)

Structure preserving optimization

A) Common sub expression elimination

eg.1. Consider the following basic block

1) a = b+c
2) b = a-d
3) c = b+c
4) d = a-d

Here 2nd and 4th statement compute the same expression. So transformed basic block is

a = b+c
b = a-d
c = b+c
d = b

Also 1st and 3rd statement have same expression but the value of b in statement 3rd is different from statement 1st. So 1st and 3rd statement do not compute the same expression and hence cannot be eliminated.

Ex-2

$T_6 = 4 * i$
$x = a[T_6]$
$T_7 = 4 * i$
$T_8 = 4 * j$
$T_9 = a[T_8]$
$a[T_7] = 19$
$T_{10} = 4 * j$
$a[T_{10}] = x$

after elimination we get

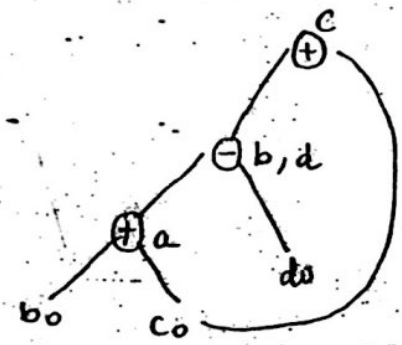
$T_6 = 4 * i$
$x = a[T_6]$
$T_8 = 4 * j$
$T_9 = a[T_8]$
$a[T_6] = T_9$
$a[T_8] = x$

Common subexpression elimination using DAG

ex-

$a = b + c$
$b = a - d$
$c = b + c$
$d = a - d$

DAG



because the DAG has only three non leaf nodes, the transformed basic block will contain only three statements

If b is not alive on exit then no need to compute b

we get

$a = b + c$
$d = a - d$
$c = d + c$

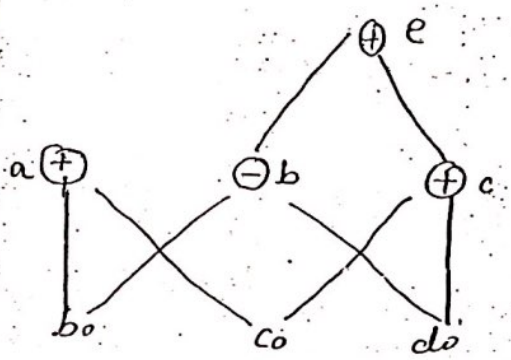
If both b & d are alive on exit, then we get

$a = b + c$
$b = a - d$
$c = b + c$
$d = b$

Ex-3 Construct DAG for following basic block

$a = b + c$
$b = b - d$
$c = c + d$
$e = b + c$

DAG



In statement 1 and 4 the value of both variables b and c changes so they does not exhibit common subexpressions

Ex-4 Construct DAG for following basic block and optimize the code.

$a = x + y$
$z = x$
$b = x + y$
$x = b$
$c = x + y$

b) Dead code elimination (Dead Code Elimination)

A variable is said to be dead if its value is not used anywhere in the program. Subsequently an assignment statement made to a dead variable can be safely removed.

Ex
```
x = t3
a[t2] = t5
a[t4] = t2
Print a[t4]
```

Here x=t3 is a dead code.

c) Copy Propagation

Let we have following expressions

a=d+e b=d+e

c=d+e

Here common sub expression in c=d+e is eliminated by using new variable t to hold the value of d+e

```
t=d+e        t=d+e
a=t          b=t
      \      /
       c=t
```

Constant folding

Deducing at compile time that the value of an expression is a constant and using the constant instead is known as constant folding

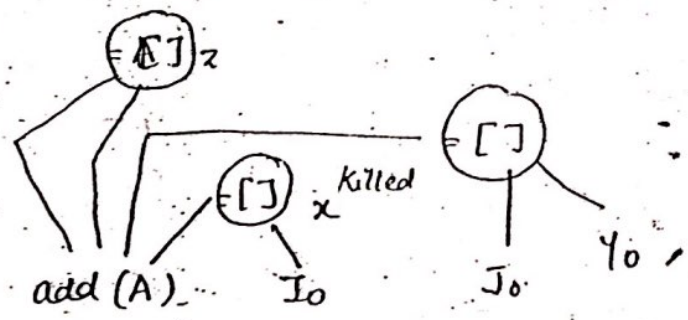
eg. a = 36 * 2
print a

Here a = 72 will be computed at compile-time.

DAG questions

Q construct DAG for

x = A[I]
A[J] = 4
z = A[I]



acc to the last rule in DAG construction algo A[J] = 4 will have 3 child and has no label, this node will kill the previous node having add A.

So for statement z = A[I] new node will be constructed.

$S_1 = 4 * I$

$S_2 = add(A) - 4$

$S_3 = S_2[S_1]$

$S_4 = 4 * I$

$S_5 = add(B) - 4$

$S_6 = S_5[S_4]$

$S_7 = S_3 * S_6$

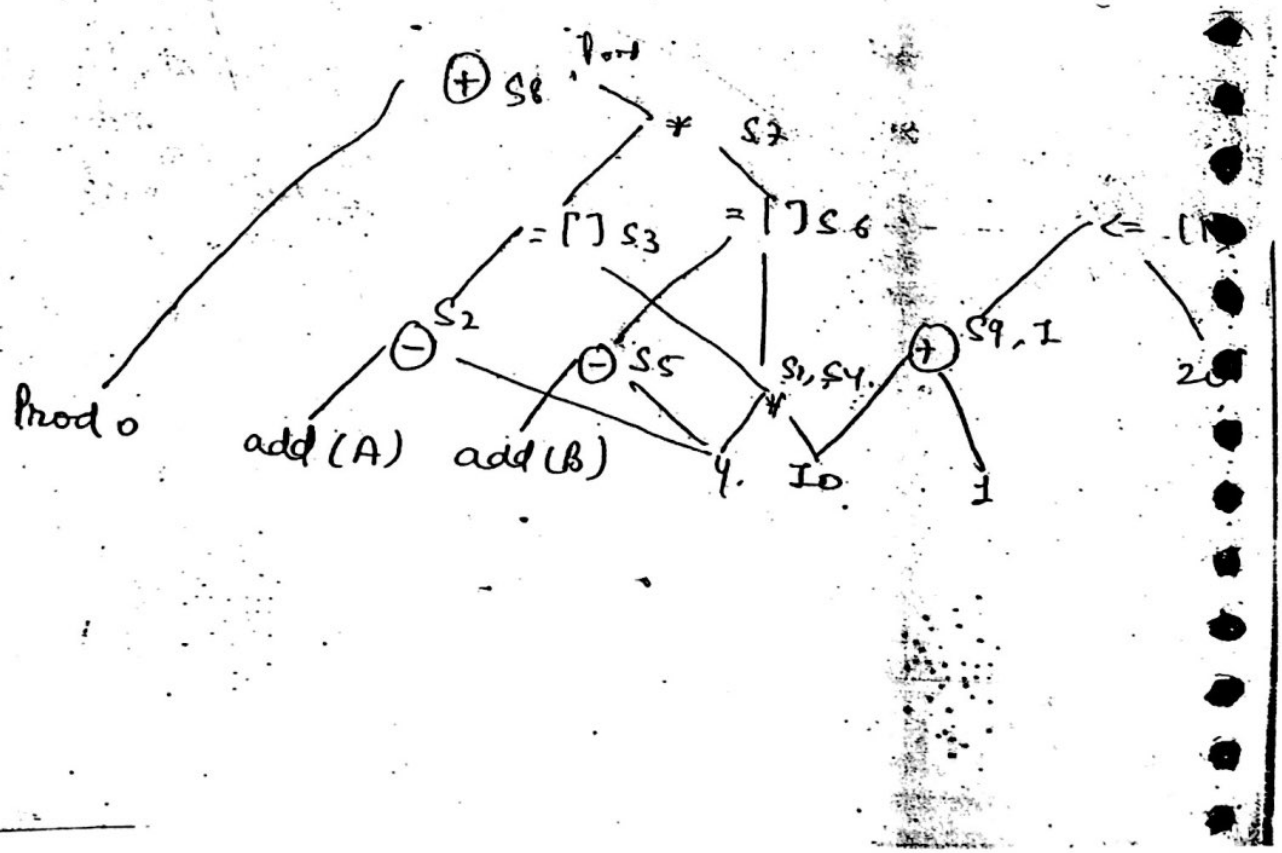
$S_8 = Prod + S_7$

$Prod = S_8$

$S_9 = I + 1$

$I = S_9$

If $I <= 20$ goto (1)



after optimization we get

```
a = b * 5
for (i=1; i<=10; i++)
{ x = a+b+i;
}
```

eg.2 while (i<= limit-2)

after code motion we get

```
t = limit-2
while (i<=t)
```

Reduction in strength

Replacement of an expensive operation by a cheaper one is called as reduction in strength.

eg. x=1

```
for (i=1; i<=10; i++)
{ x = i * 2
}
```

can be optimized as

```
x=0
for (i=1; i<=10; i++)
{ x = x+2;
}
```

Induction variable elimination

An induction variable in the loop are those variables whose assignments with in the loop are of the form $I = I \pm C$ where C is a constant.

elimination of induction variable sometimes also performs strength in reduction.

eg. we have following code in loop

```
L:  T1 = 4 * i
    T3 = T2[T1]
    I = I + 1
    if I <= 20 goto L
```

here we can see that I & T_1 are in lock step.

so when value of I increases from 1 ... 20 then T_1 will increase from 4 ... 80

so I & T_1 are induction variables

by induction variable elimination we get $T_1 = 0$

```
L:  T1 = T1 + 4
    T3 = T2[T1]
    if T1 <= 76 goto L
```

(2) Algebraic transformations

many algebraic transformations can be used to change the set of expressions computed by basic block into simplified expressions or to replace expensive operations into cheaper operations

eg. $x = x + 0$

and

$x = x * 1$

can be eliminated

eg. $x = y ** 2$

can be replaced by $x = y * y$

eg. $2 * x$

↳ $x + x$

eg. $x/2$

↳ $x \times 0.5$

eg. x^2

↳ $x * x$

(2) Loop Optimization

Generally programs spend most of their time in the execution of inner loops. The running time of program may be improved by decreasing the no. of instructions in an inner loop, or by increasing the amount of code outside the loop.

Three techniques for loop optimization

- Code motion / loop invariant
- Induction variable elimination
- reduction in strength.

Code motion / loop invariant

Code motion is a technique of moving loop invariant computations outside the loop.

A loop invariant computations is one that computes the same value every time a loop is executed.

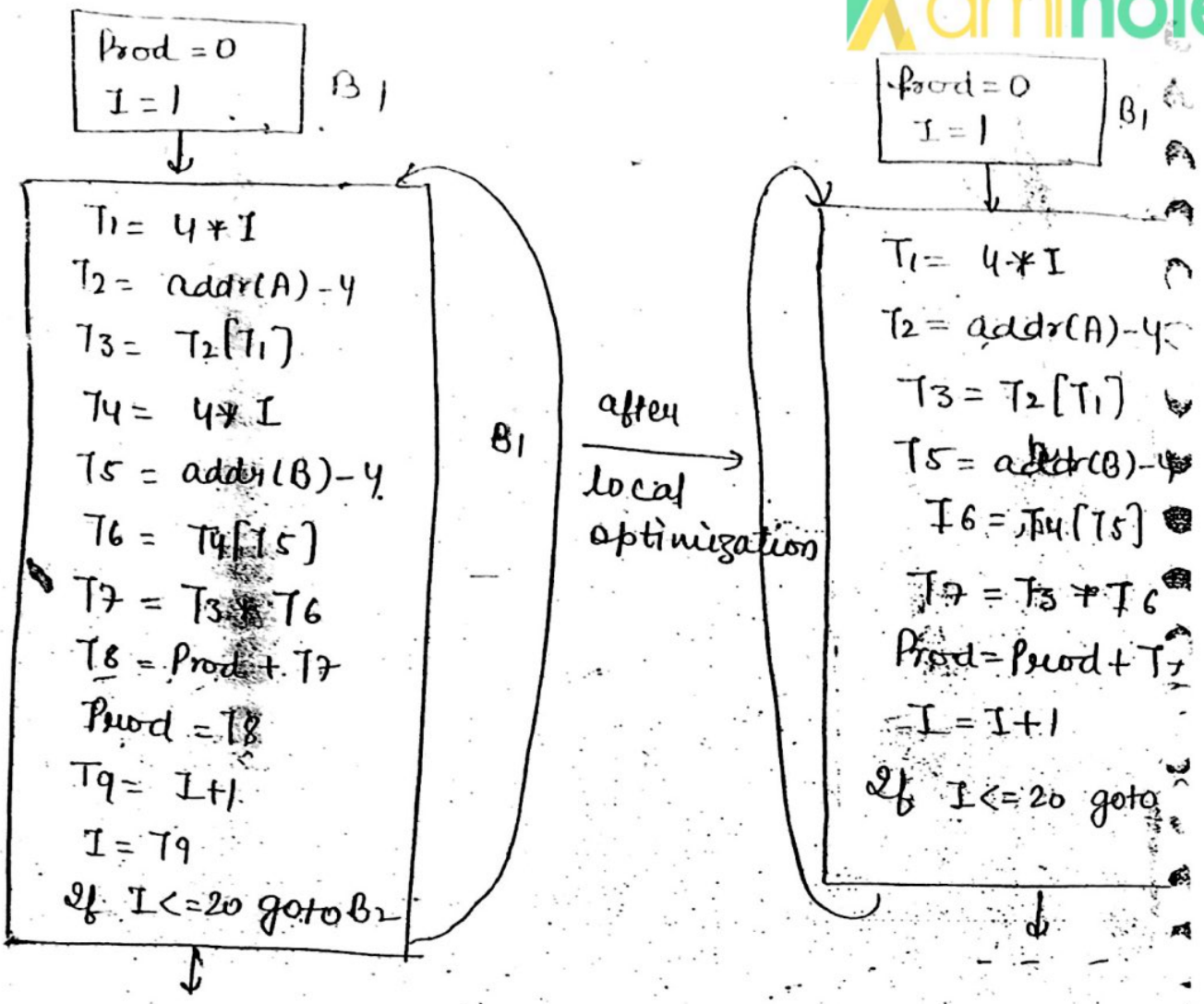
eg.
```
for (i= 1; i<=10; i++)
{ a= b*5
  x= a+ b+ i;
}
```

```
begin
prod = 0
I = 1
do
  begin
    Prod = Prod + A[I] * B[I]
    I = I + 1
  end
  while I <= 20
end
```

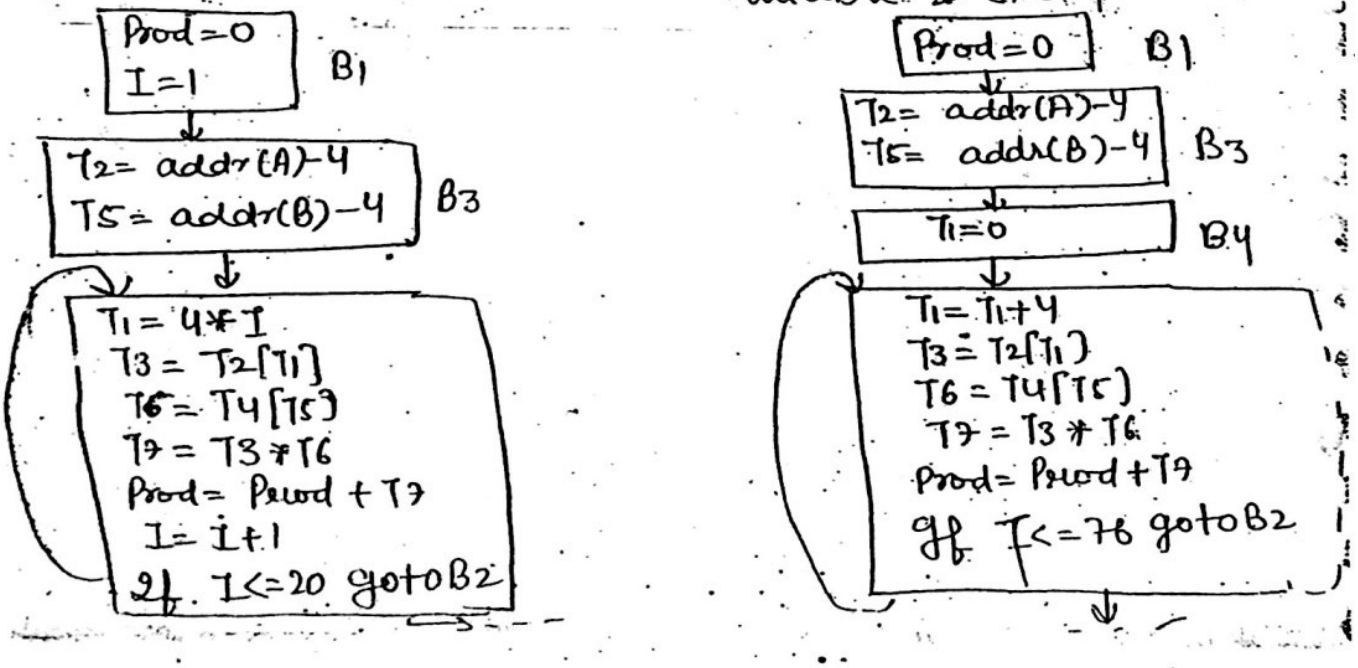
1) Generate three address code
2) Create basic blocks & flow graph
3) Perform local & loop optimization

Ans Three address code

1) Prod = 0
2) I = 1
3) T1 = 4 * I
4) T2 = Addr(A) - 4
5) T3 = T2[T1]
6) T4 = 4 * I
7) T5 = Addr(B) - 4
8) T6 = T5[T4]
9) T7 = T3 * T6
10) T8 = Prod + T7
11) Prod = T8
12) T9 = T + 1
13) T = T9
14) If I <= 20 goto (3)



Loop optimization



```
i=m-1; j=n; v=a[n]
while(1)
{ do i=i+1; while(a[i]<v);
  do j=j-1; while(a[j]>v);
  if(i>=j) break;
  x=a[i];
  a[i]=a[j]
  a[j]=x
}
x=a[i];
a[i]=a[n];
a[n]=x
```

Three address code

```
1)  T1=m-1
2)  i=T1
3)  T2=n
4)  j=T2
5)  T3=4*n
6)  v=a[T3]
```
B1

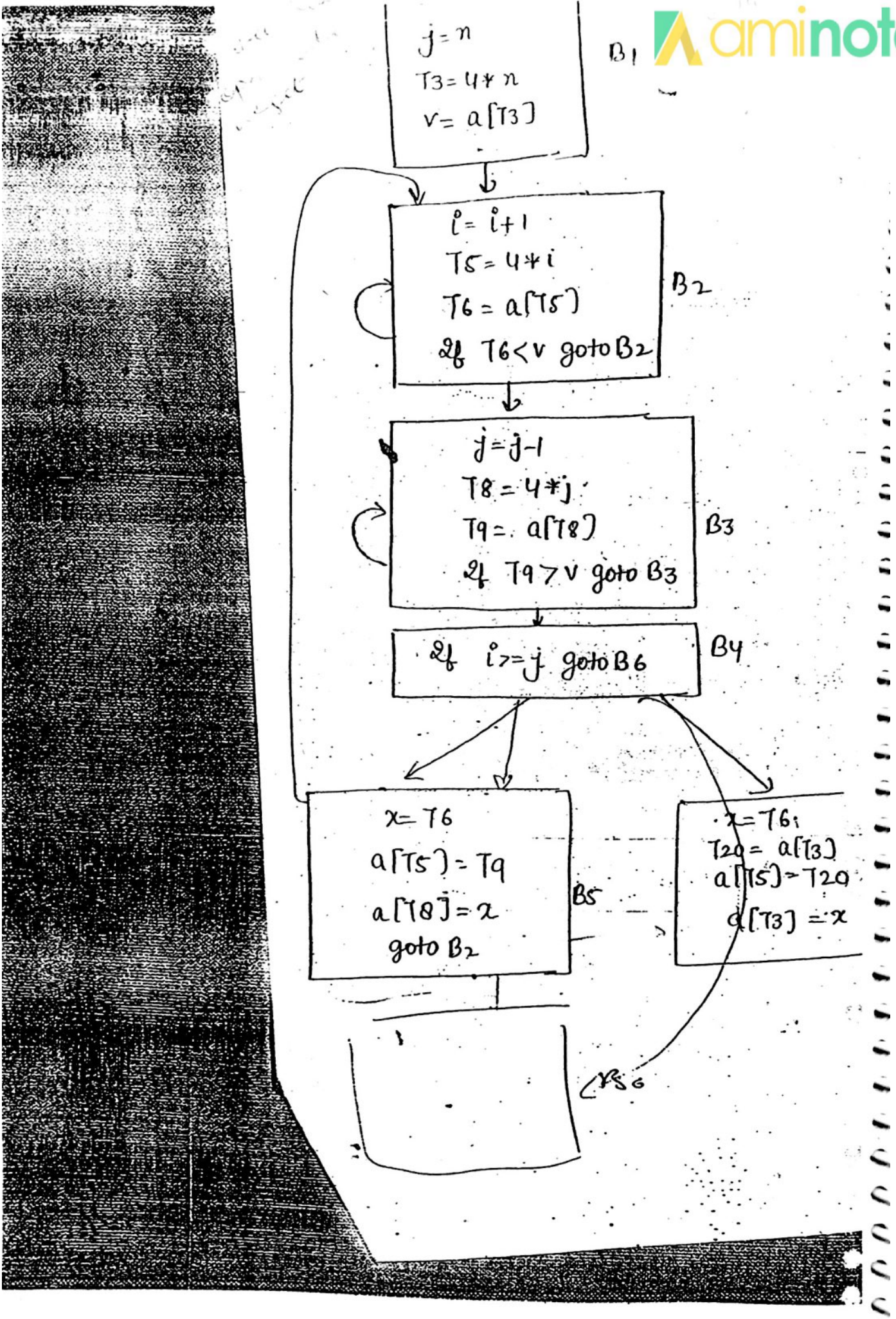
```
7)  T4=i+1
8)  i=T4
9)  T5=4*i
10) T6=a[T5]
11) if T6<v goto(7)
```
B2

```
12) T7=j-1
13) j=T7
14) T8=4*j
15) T9=a[T8]
16) if T9>v goto(12)
```
B3

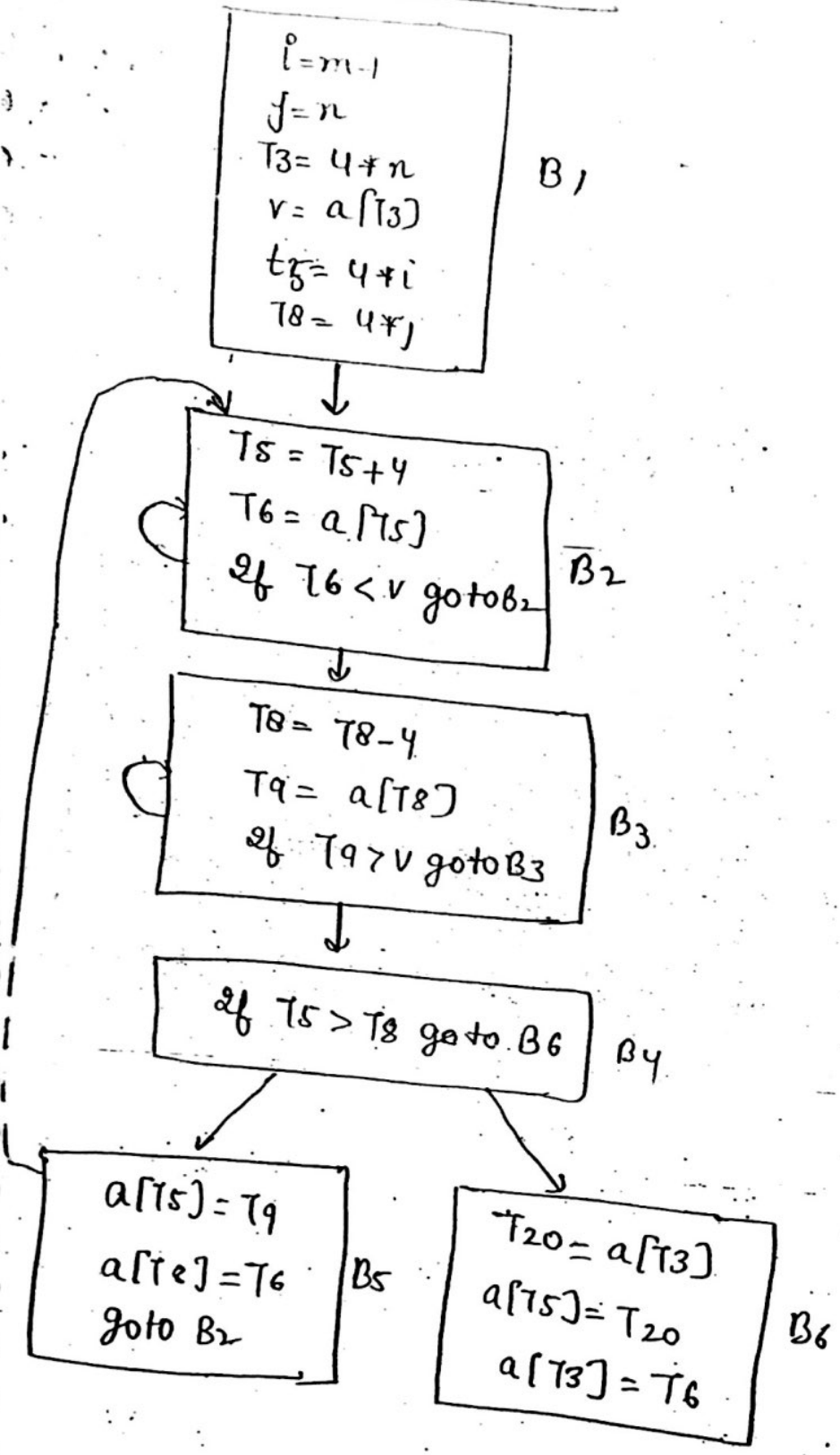
```
17) if i>=j goto(28)
```
B4

```
18) T10=4*i
19) T11=a[T10]
20) x=T11
21) T12=4*i
22) T13=4*j
23) T14=a[T13]
24) a[T12]=T14
25) T15=4*j
26) a[T15]=x
27) goto(7)
```
B5

```
28) T16=4*i
29) T17=a[T16]
30) x=T17
31) T18=4*i
32) T19=4*n
33) T20=a[T19]
34) a[T18]=T20
35) T21=4*n
36) a[T21]=x
```
B6



Loop optimization



Peephole Optimization

Peephole optimization is a method for improve the performance of the target program examining a short sequence of target instructions (called the peephole) and replacing these instructions by a shorter or faster sequence. Peephole is a small moving window on the target program.

Some peephole optimization characteristics are

1) Redundant instructions elimination
2) Flow of control optimizations
3) Strength reductions
4) Use of machine codes

Unreachable code

```
DEBUG = 0

if DEBUG = 1 then
    print debugging information
```

TAC

```
    DEBUG = 0
    if DEBUG = 1 goto L1
    goto L2
L1: print debugging information
L2: -
```