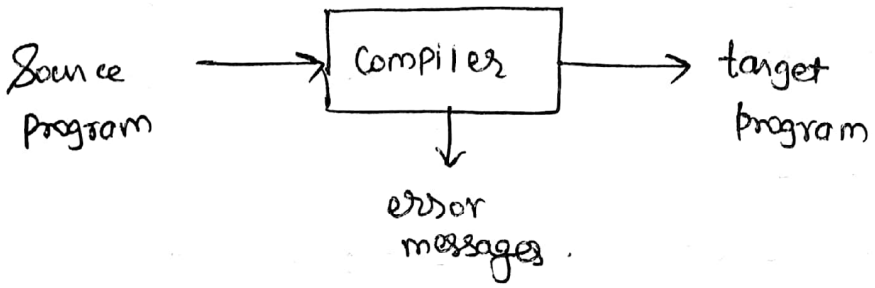
Introduction to Compilers:

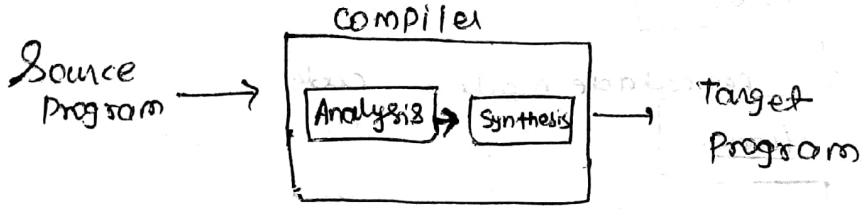
- A compiler is a program that reads a program written in one language (i.e the Source lng) and convert it into an equivalent program in another language i.e the target lng.



- The Compiler takes a Source program such as higher level lng such as C, PASCAL, FORTRAN & converts it into low level lng. or machine level lng such as assembly lng.

Structure of Compiler:

- The compilation can be done in two parts: Analysis and Synthesis in analysis part the Source program is read and broken down into consituent pieces, and creates an intermediate representation of the Source program.
- The Synthesis part constructs the desired target program from the intermediate representation.



Properties of Compiler:

→ The compiler itself must be bug free.
→ it must generate correct machine code.
→ The generated machine code must run fast.
→ The compiler must be portable

* Analysis of the Source program:

The Source program can be analyse in three phase.

1) Linear analysis:

In this type of analysis the Source String is read from left to right and grouped into tokens. eg. Token for the lng. can be identifiers, Constants, relational operators, Keywords

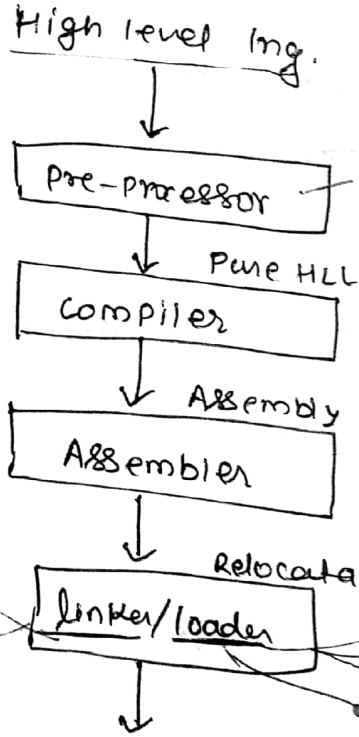
2) Hierarchical Analysis:

In this analysis, character or tokens are grouped hierarchically into nested collections for checking them Syntactically.

3) Semantic Analysis:

This kind of analysis ensure the correctness of meaning of the program.

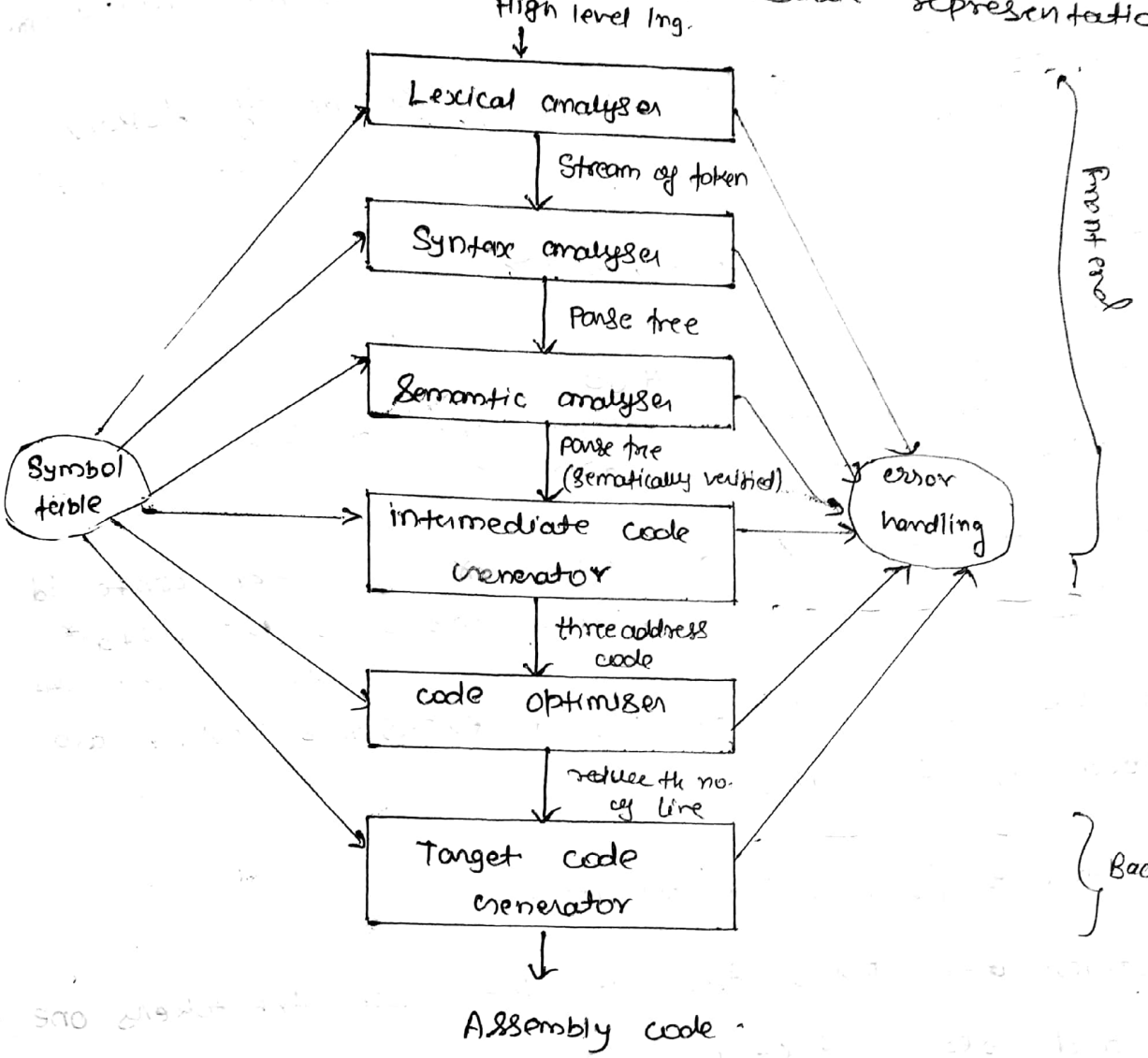
Language processing System:



first linking happen than loading.

Phase of a Compiler

- Basically two phases of Compilers, Analysis & Synthesis phase.
- Analysis Phase creates an intermediate representation from the given Source code. Synthesis Phase creates an equivalent target program from the intermediate representation.



Symbol table :

- it is a data structure being used and maintained by the Compiler, consist of all the identifiers name along with their types.

Insert, Look up → Link list, Hash table

int a = 5;
a 5 int
<name value attribute>

error handling

- Each Phase can encounter errors after detecting an error, a Phase must somehow deal with that error. Syntax & Semantic analysis Phases usually detect/handle a large fraction. The lexical phase can detect errors where the characters remaining in

the i/p don't form any token of the lng.

* Lexical Analyser: (lex tool)

- it reads the program and convert it into tokens. The chara- -cter sequence forming a token is called the lexeme for the token. token for the lng. can be identifier (varname, funname), constant & relation opr. keyword, separators (; ,) etc.
- it convert stream of lexeme into a stream of tokens
- non tokens (comments, preprocessor directive, macro, blanks, tabs, newline etc).

token:

<token name, attribute value>

eg. position = initial + rate * 60

Here position is the lexeme that could be mapped into token <id1> ....

<id1> <=> <id2> <+> <id3> <*> <60>

- if alphabet followed by alphabets or digit then consite id (identifier) otherwise same. regular expression for id = l(l+d)* letter followed by letter or digit
- Tokens are defined by the regular expressions which are understood by the lexical an

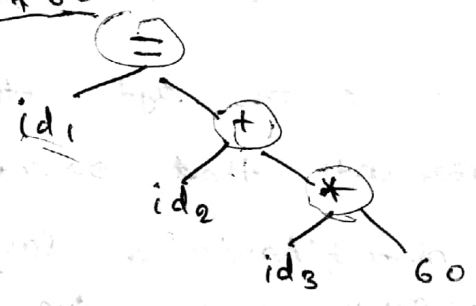
IMD

* Syntax Analyzer/(Parser): (tool Yacc)

- It construct the parse tree. it takes all the tokens one by one and uses context free grammar to construct a parse tree.

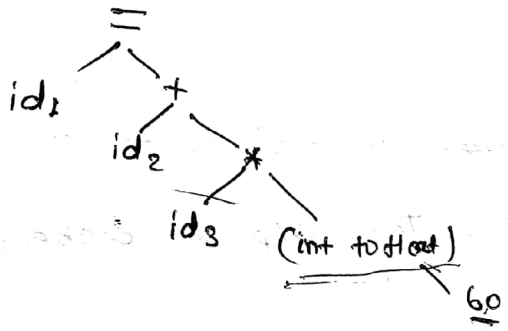
In a parse tree/Syntax tree all the interiors nodes represents an operation & children of the nodes represents argument of the operation.

eg $id_1 = id_2 + id_3 * 60$



* Semantic Analyzer:

- it verifies the parse tree, whether it's meaningful or not. it further more produces a verified parse tree
- It may be classified into category type checking, type conversion.
- it checks whether the program satisfies semantic consistency or not ? eg. like array index only represented using int. array[10] ✓ array[1.5] ✗
- The language specification may permit some type conversion like int to float, float to int etc.



* Intermediate code generator:

- it generates intermediate code that is a form which can be readily executed by machine.

eg three address code etc. imp. — means 3 line code. intermediate code is converted to machine language using the last two phases which are Platform dependent.

- it is easy to produce and easy to translate into the target program.

eg.
```
temp1 = int to real(60)
temp2 = id3 * temp1
temp3 = id2 * temp2
id1 = temp3
```

* Code optimizer:

- it attempts to improve the intermediate code so that
  - it consume the fewer resources and produces more speed.
- The meaning of the code being transformed is not altered changed.
- Optimization can be categorized into two types machine dependent and machine independent.

eg tmp1 = id3 * 60.0
id1 = id2 + tmp1

* Code generator: (TOOL LANCE)

- it's main purpose is to write a code that the machine can understand. The o/p is depends on the type of assembler.
- i.e it converts the code into Assembly lng. i-e low level lng.

eg.

```
LDF   R2, id3
MULF  R2, #60.0
LDF   R1, id2
ADDF  R1, R2
STF   id1, R1
```

eg. find the total no. of tokens.

```
int main()
{
  // 2 variable
  int a,b;
  a = 10;
  return 0;
}
```

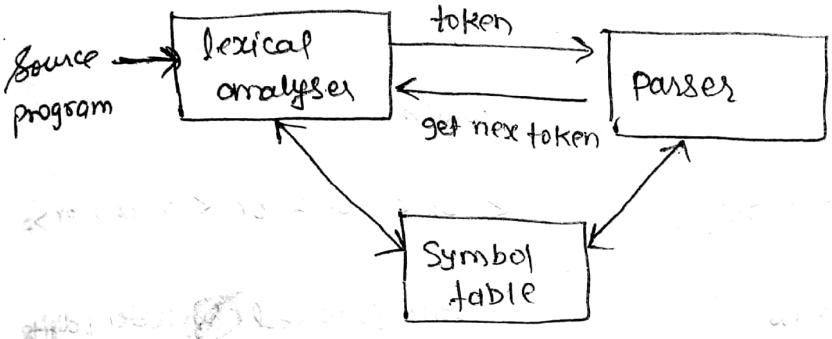
tokens are.

'int' 'main' '(' ')' 'int' 'a' ',' 'b' ';' 'a' '=' '10' ';'

'return' '0' ';' '}'

The ROLE of the Lexical Analyzer:

- it's main task is to read the i/p character and produce as O/P a sequence of token that the parser uses for Syntax analysis.
- upon receiving a "get next token" command from the parser, the lexical analyzer reads i/p characters until it can identify the next token.



talk about token.

- lexical analyzer is a part of the compiler so that it may also perform certain secondary tasks at the user interface.
- one of the such task is stripping out from the source program comments and white space in the form of blank, tab, and newline characters.
- if any error will present then LA will correlate that error with source file and line number.

Issues in lexical analysis:

- Simpler design
- Compiler efficiency is improved
- Compiler portability is enhanced.

Tokens, Patterns, and lexemes.

- A token is a set of string over the source alphabet.
- A Pattern is a rule that describes that set. (set of string is described by a rule called a Pattern.)

- A lexeme is a sequence of character matching that Pattern.

eg in Pascal for the stmt

Const Pi = 3.1416

Ethen substring Pi is a lexeme for the token identifier.

Token	Sample lexemes	informal description of Pattern
if	if	if
while	while	while
relation relop	<, <=, =, <>, >, >=	< or <= or = or <> or > or >=
id	count, sum, i, j, Pi	letter followed by letter & digits
Num	0, 12, 3.141 ...	Any numeric constant

eg printf ("sum = %d\n", total);

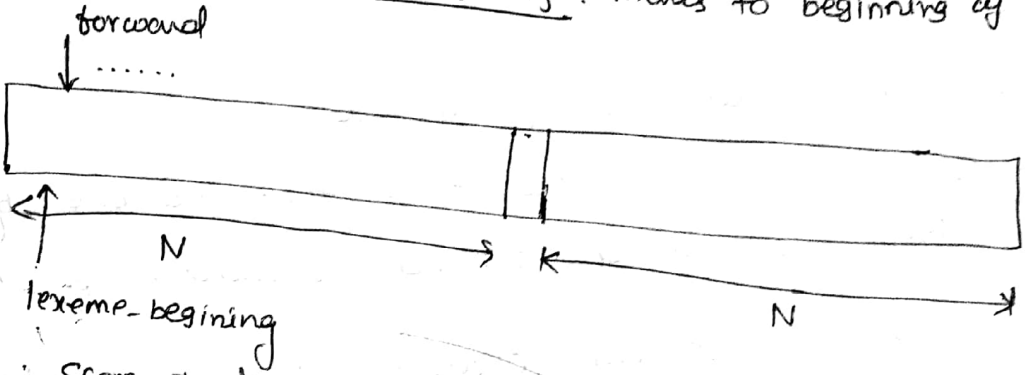
- Printf and total are lexemes matching the Pattern for token id,
"sum = %d\n" is lexeme matching literal.
- Regular expressions are used to specify lexeme Patterns.

Input Buffering :

Speed of the lexical analysis is concern.

- Lexical analysis needs to look ahead several characters before a match can be announced.
- We have two buffer i/p scheme that is useful when look ahead is necessary
  - → Buffer Pair
  - → Sentinels

- Reading the source program can be split up by a two buffer scheme. each buffer is of size N and N usually the size of disk (N is no. of char. eg 1024 or 4096)
- If lesser than N character remaining in the i/p file. th a special character eof marks the end of the source file.
- There are two pair: lexeme_begining : marks to beginning of current lexem

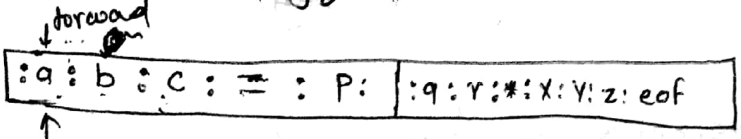


- forward : Scan a head until a pattern matches or eof

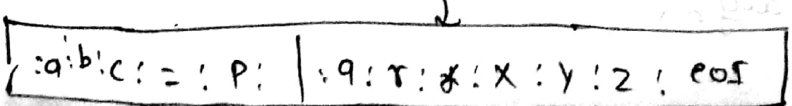
Algo

```
If (forward at end of first half ) then begin
    reload second half
    forward = forward + 1
end
elseif (forward at end of second half ) then
begin
    reload first half
    move forward to beginning of first half
end
else
    forward = forward + 1
```

eg abc = pqr * xyz



abc → identifier (when forward reaches at = then lexeme_begining st
token from time).



pqr → identifier

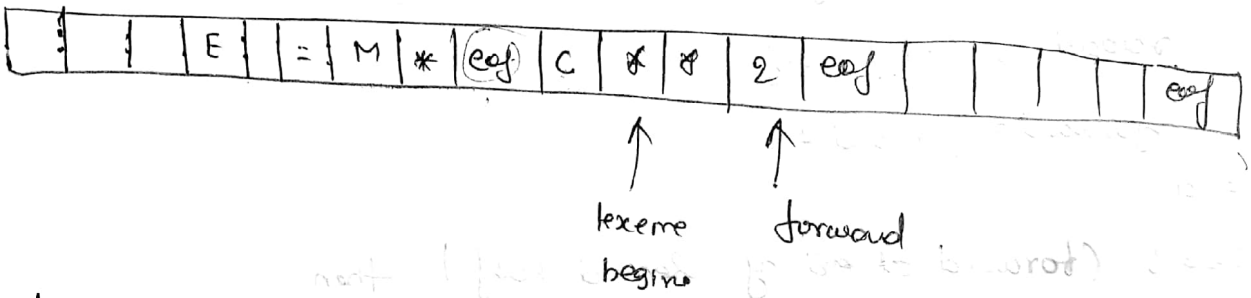
Sentinels :

- In the previous scheme each time when the forward pointer is moved, a check is done to ensure that one half of the buffer has not moved off. If it is done then other half must be reloaded.
- Therefore the ends of the buffer require two tests for each advance of the forward pointer.

Test 1: For end of buffer

Test 2 : To determine what character is read :

- The usage of sentinel reduces two test to one by extending each buffer half to hold a sentinel character at the end.
- The sentinel is a special character that can't be part of the source program.

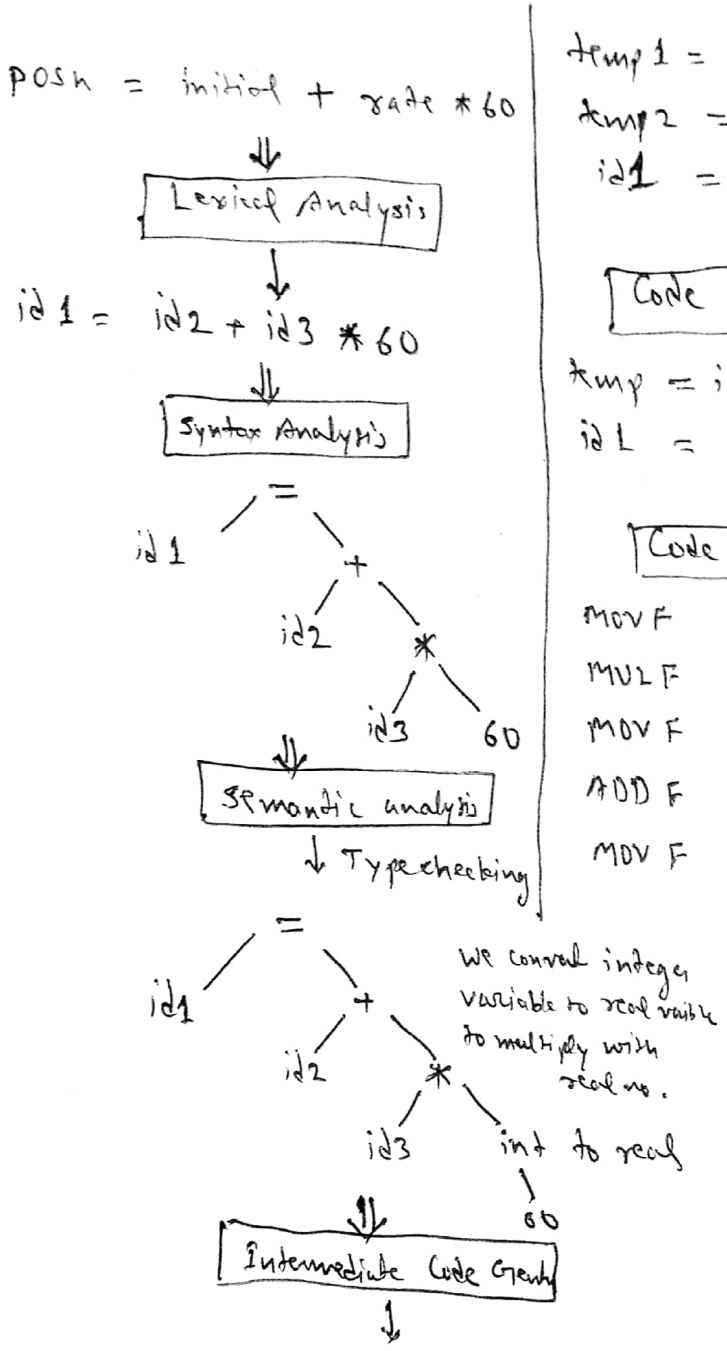


Algo

```
forward = forward + 1
If forward = eof then
begin
    reload second half
    forward = forward + 1
end
else if forward at end of second half then begin
    reload first half
    move forward to beginning of first half
end
else
    terminate lexical analysis.
```

Adv.

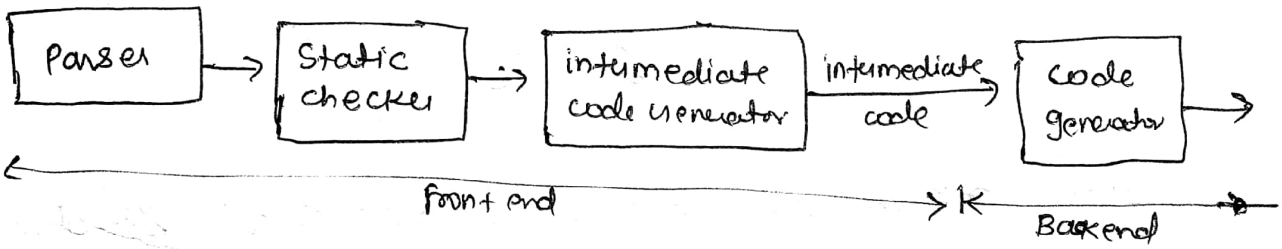
- most of the time it perform only one test to see whether forward pointer points to an eof.
- only when it reaches the end of the buffer half or eof. it performs more tests.



Intermediate code generation:

- In analysis-Synthesis model of a Compiler the front end translate a source program into an intermediate representation from which the back end generates target code.
- The benefits of using machine independent intermediate code are:

1) Because of machine independent intermediate code, Portability will be enhanced.
2) Retargeting is facilitated. (a compiler for a different machine can be created by attaching a back-end for the new machine to an existing front end.
3) it is easier to apply source code modification to improve the performance of source code by optimising the intermediate code.



- uses register names, but has unlimited Registers.
- uses control Control Structure (like jump goto.).
- uses OP-codes (eg push).

eg Three address code (it uses only 3 address)

- $x = y + z$

Intermediate languages:

- Syntax tree and postfix notation are two kind of intermediate representation. and third is called Three address code

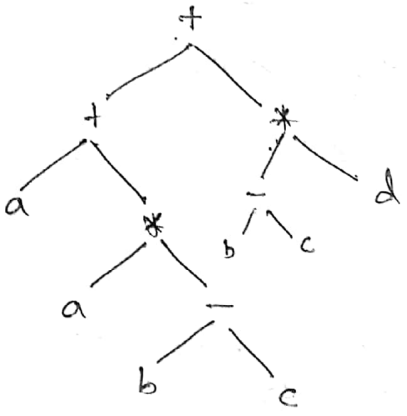
Syntax tree

Production	Semantic Rule.
$E \rightarrow E_1 + T$	E.node = new Node('+', E_1.node, T.node)
$E \rightarrow E_1 - T$	E.node = new Node('-', E_1.node, T.node)
$E \rightarrow T$	E.node = T.node
$T \rightarrow (E)$	T.node = E.node
$T \rightarrow (id)$	T.node = new Leaf(id, id.entry)
$T \rightarrow num$	T.node = new Leaf(num, num.val)

representation of IRs:

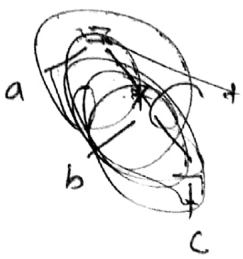
- Syntax tree
- Postfix expression
- 3 address code

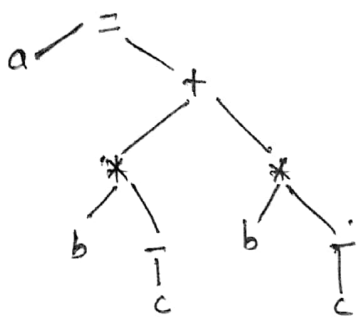
$a + a * (b - c) + (b - c) * d$



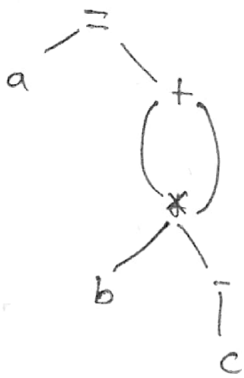
- A syntax tree depicts the natural hierarchical structure of a source program.
- A dag gives the same information but in a more compact way bcz common expressions are identified.

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





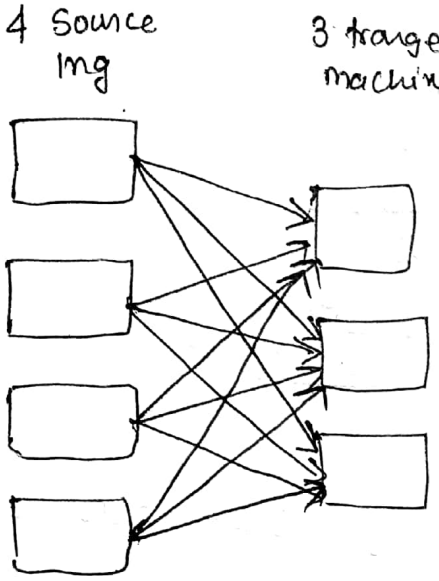
Syntax tree



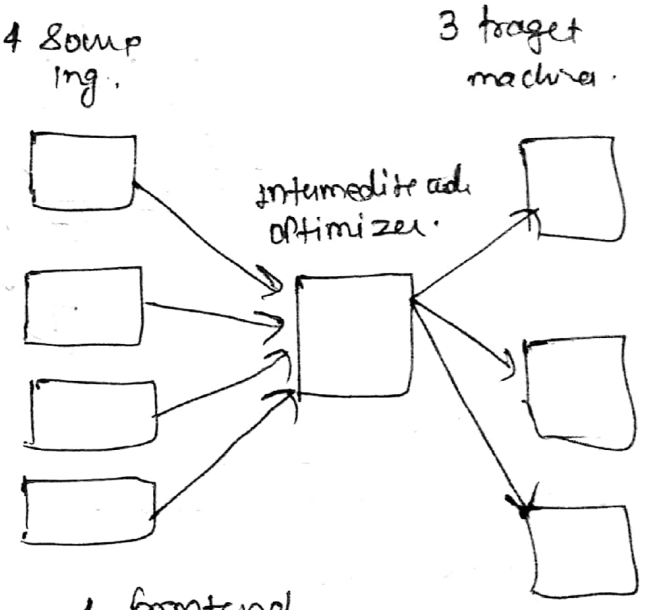
Dag (Directed Acyclic

To representation of Syntax tree.

Why we need intumediate code.



4 - frontend
+ 4*3 optimizer
+ 4x3 code generator



4 frontend
1 Optimizer
3 code generator.

1) node are allocated from an array of records and the index or position of the nodes serves as the pointer to the node.

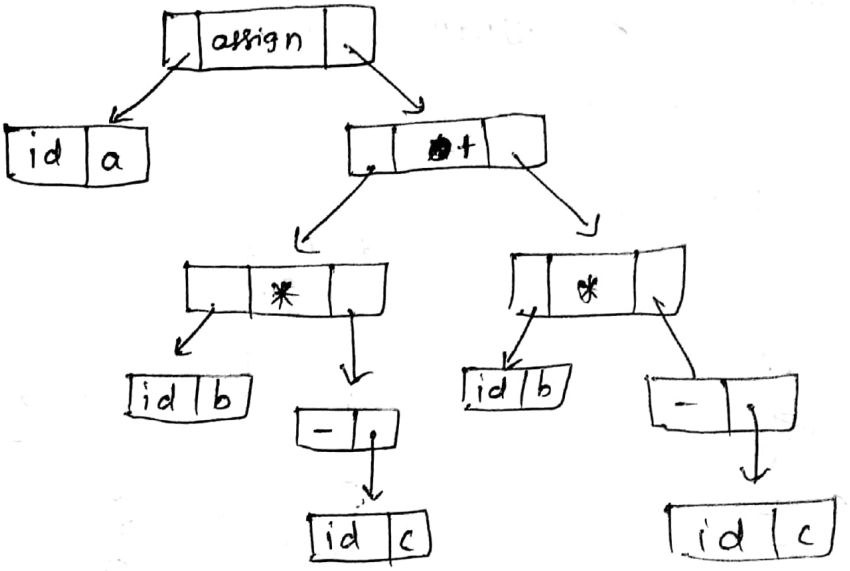
All the nodes in the Syntax tree can be visited by following pointers. Starting from the root at pos. 10.

leaves

0	id	b
1	id	c
2	unminus	1
3	*	0 2
4	id	b
5	id	c
6	unminus (-)	5
7	*	4 6
8	+	3 7
9	id	a
10	assign (=)	9 8

left and right children of an intumediate node.

② each node is represented as a record with a field for it's operator and additional fields for pointers to it's children.



3 address code:

- Three address code is a sequence of statements of the general form.

$$x := y \text{ OP } z$$

- There is at most one operator (OP) on the right side of an instruction.
- At most three addresses are used to represent any statement. (2 for operands & one address for the result).

eg $x + y * z$ (Here 3 add. is used (x, y, z).)

$t_1 = y * z$ (Assume this A/c to precedence & associativity).

$t_2 = x + t_1$

eg $a = (x + y) + z$

$t_1 = x + y$

$t_2 = t_1 + z$

$a = t_2$

- Three address code is build from two concept address & instructions.

address can be: name, constant, C

Type of three address Stmt:

- Statement can have symbolic labels and there are stmt for flow of control.

1) Assignment statements of the form $x = y \text{ op } z$

   op → binary / arithmetic / logical oprn.

2) Assignment instructions of the form $x = \text{op } y$

   op → unary operation (++, −, <<, >> ...)

3) Copy statement of the form $x = y$ where $x \leftarrow y$

4) The unconditional Jump goto L three address stmt with label L is the next to be executed.

5) Conditional Jump such as IF X relop Y goto L

6) Procedure call Param x and call P return y

   Pram x
   call P, n
   y = call (P, n)

7) Address and pointer assignment $x := \&y$ $x := *y$.

8) $x = y[i]$ & $x[i] = y$.

9) $x = *y$ and $*y = y$

Implementation of three address Stmt:

- A three address statement is an abstract form of intermediate code.
- It can be implemented as quadruples, triples, indirect triples tree or dag.

eg. Three address code for

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

3 add. code.

$t_1 = b * c$
$t_2 = a + t_1$
$t_3 = b * c$
$t_4 = d / t_3$
$t_5 = t_2 - t_4$

Quadruples: it is a record structure with 4 fields. which we call OP, arg1, arg2, result.

- The OP field contain an internal code for the operator.

for the abov eg.

OP	arg1	arg2	result
*	b	c	t_1
+	a	t_1	t_2
*	b	c	t_3
/	d	t_3	t_4
- (uminus)	t_2	t_4	t_5

Exceptions:

1) With unary operator $x = -y$ or $x = y$ don't use arg2
2) Operator like Param use neigher arg2 nor result.
3) conditional & unconditional Jump put tanget in result.

Triples: it is a record structure with three field OP, arg1, arg2.

- arg1 & arg2 are field for the arguments of OP are either pointer to symtbl or constant or Pointer to structure.
- it avoid the entering temporary names into the Symbol table.
- So it optimize the space complexity.

OP	arg1	arg2
*	b	c
+	a	(0)
*	b	c
/	d	(2)
−	(1)	(3)

bcz t1 is at index 0.

Bcz t3 present at index 2.

Indirect triples:

- it list the pointer to triples rather than listing the triples themselves.

instruction

(0)
(1)
(2)
(3)
(4)
(5)
...

	OP	arg1	arg2
(0)	*	b	c
(1)	+	a	(0)
(2)	*	b	c
(3)	/	d	(2)
(4)	−	(1)	(3)

list of pointer to triples ⇒ Optimizing Compiler can reorder instruction list, instead of affecting the triples themselves.

* SDT into three address code:

- It is constructed based on the grammar construct.
- The synthesized attribute S.code represents the three address code for the assignment S.
- The non terminal E has 2 attributes:

  E.Place : (the name that will hold the value of E)

  E.code : (the sequence of 3 address statement evaluating E)
- The function newtemp return a sequence of distinct names $t_1, t_2 \ldots$ in response to successive calls.
- newn is used to Compute the operations to exist with its parameter.

eg

Production	Semantic Rule
$S \rightarrow id : E ;$	S.code = E.code || gen(top.get(id.lexeme) ':' E.addr) or gen(id.Place ':' E.Place)
$E \rightarrow E_1 + E_2$	E.Place = new Temp() E.code = E_1.code || E_2.code || gen(E.Place = E_1.Place + E_2.Place)
$E \rightarrow E_1 * E_2$	E.Place = new Temp() E.code = E_1.code || E_2.code || gen(E.Place = E_1.Place * E_2.Place)
$E \rightarrow -E_1$	E.Place = new Temp() E.code = E_1.code || gen(E.Place = uminus E_1.Place)
$E \rightarrow (E_1)$	E.Place = E_1.Place (Here no need to generate the New temp bez parenthesis denotes the immediate addr which can be directly transfer). E.code = E_1.code
$E \rightarrow id$	E.Place = id.Place E.code = ''

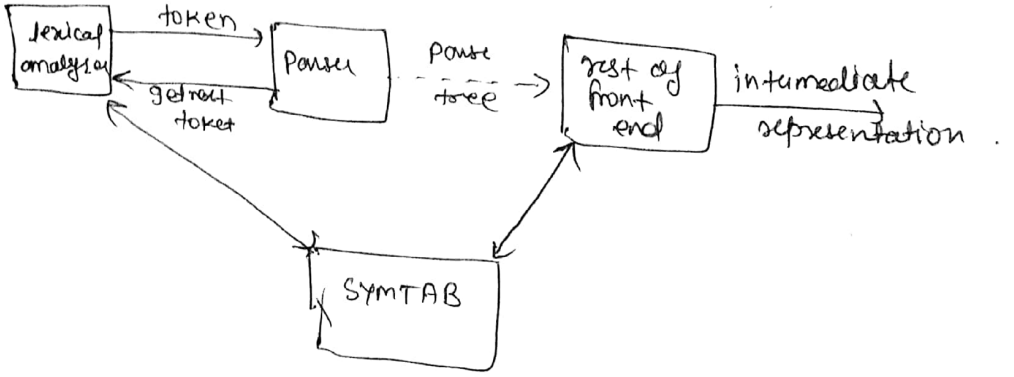
Issue in the Design of a Code generator:

① Input to the code generator:

- The input to the code generator consist of the intermediate representation of the Source program produced by the front end together with information in the SYMTAB that is used to determ the run-time address of the data objects denoted by the nam in the Intermediate representation.

Role of the Parser:

- Parser for any program take the i/p string w (which is the token) and produces as output either parse tree for w.
- If w is a valid sentences of a grammar or error message indicate that w is not valid sentence of given grammar.
- The goal of the parser is to determine the syntatic validity of a source string is valid. tree build for the use of the next phase.



Types of error can occur.

- lexical (such as misspelling an identifier, keyword or operator).
- Syntatic (such as arithmetic expression with unbalance Paranthesis)
- Semantic (operator applied to incompatible operand)
- logical (infinitely recursive call)

* Error recovery Strategies:
  - Panic mode recovery
  - Phase level
  - error Production
  - global Correction

* left recursion avoid using

$A \to A\alpha / \beta$ ← left recur.

Sol. $A \to \beta A'$

$A' \to \alpha A' / \epsilon$

* Left factoring: Procedure for converting Non-deterministic grammar to deterministic gives suitable grammar for predictive parsing.

Algo.
1) I/P is
2) An equivalent left factored grammar.
3) for each nonterminal A find the longest prefix α common to two or more of its alternatives. If $\alpha \neq \epsilon$ there is a nontrivial common Prefix, replace all of the Production $A \to \alpha B_1 | \alpha B_2 | \dots \alpha B_n | \gamma$.

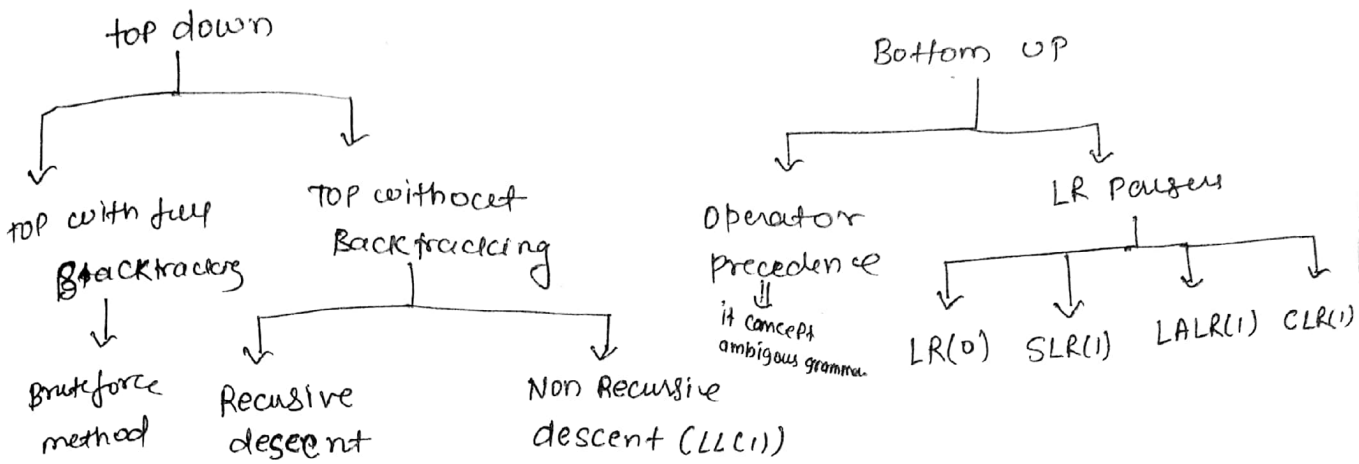
eg: $A \to aq$, $A \to ab$, $A \to ac$ } NDG.

$A \to aB$

$B \to b | c$ } left factored grammar.

Parsing

Passers ⇒ For any parser grammar shouldn't be ambiguous except Operator precedence



top down Parser:

, derive from start Symbol to last Symbol.

Steps

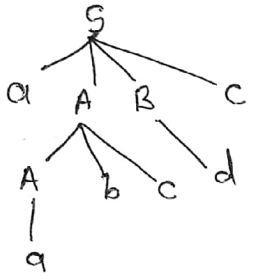
1) Starts from root
2) Expension
3) LL
→ left to right scan.
→ Left most derivation.

eg. $S \rightarrow aABC$

$A \rightarrow Abc/b$

$B \rightarrow d$

$\omega = abbcde$



Bottom up Parser:

Starts from the word to starting production.

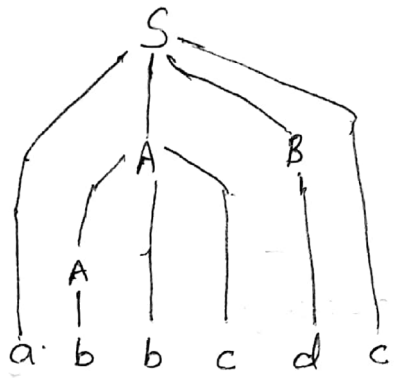
Steps:

1) Roots – starts
2) Reduction
3) LR
→ left right scan
→ Right most derivation.

eg $S \to aABC$

$A \to Abc / b$

$B \to d$ $\quad w: abbcdc$



↑ Bottom to up.

First and Follow :

- if we have a variable or string from that variable we try deriving all the strings then whatever is coming in the beginning, the first symbol is called as FIRST of that string.
- What is the terminal which could follow a variable in the process of derivation. it is called as follow.

Algorithm FIRST(X):

1) if X is terminal the FIRST(X) is $\{X\}$
2) if $X \to \epsilon$ is a production, then add ϵ to FIRST(X).
3) if x is a non terminal and $x \to Y_1 Y_2 \cdots Y_K$ is production, then place a in FIRST(X) if for some i a is in FIRST(Y_i).

Algorithm Follow(X):

- Place $ in Follow(S). where S → Start Symbol. here $ is the rightend marker of i/p buffer.
- if there is a production $A \to \alpha B\beta$ then everything in FIRST(B) except for ϵ is placed in follow(B).
- if there is a production $A \to \alpha B$ or production $A \to \alpha B\beta$ where FIRST(B) contains ϵ. then every_thing in Follow(A) = Follow(B).

8. $E \rightarrow TE'$

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

$T \rightarrow FT'$

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

$F \rightarrow (E)/id$

	First	follow
E	{ id, (}	{ $,) }
E'	{ +, ϵ }	{ $,) }
T	{ (, id }	{ $+, $,) }
T'	{ *, ϵ }	{ +, $,) }
F	{ (, id }	{ *, +, $,) }

FIRST(E) = FIRST(T) = FIRST(F) = { (, id }

FIRST(E') = { +, ϵ } FIRST(T') = { *, ϵ }

Follow(E) = Follow(E') = { $,) }

Follow(T) = Follow(T') = { $,), $ }

Follow(F) = { *, $,), $ }

9. $S \rightarrow ABCD$

$A \rightarrow a$

$B \rightarrow b$

$C \rightarrow c$

$D \rightarrow d$

	FIRST( )	Follow( )
S	a	{ $ }
A	a	{ b }
B	b	{ c }
C	c	{ D }
D	d	{ $ }

9. $S \rightarrow ABCDE$

$A \rightarrow a/\epsilon$

$B \rightarrow b/\epsilon$

$C \rightarrow c$

$D \rightarrow d/\epsilon$

$E \rightarrow e/\epsilon$

	FIRST	Follow
S	{a, b, c}	{ $ }
A	{a, ϵ}	{ b, c }
B	{b, ϵ}	{ c }
C	{c}	{d, e, $ }
D	{d, ϵ}	{ e, $ }
E	{e, ϵ}	{ $ }

eg

$S \rightarrow Bb/Cd$

$B \rightarrow aB/\epsilon$

$C \rightarrow cC/\epsilon$

	first	follow
S	{a, b, c, d}	{$}
B	{a, ε}	{b}
C	{c, ε}	{d}

LL(1) Parsing table :

eg

eg

	first	follow
$E \rightarrow TE'$	{id, (}	{$,)}
$E' \rightarrow +TE'/\epsilon$	{+, ε}	{$,)}
$T \rightarrow FT'$	{id, (}	{+,), $}
$T' \rightarrow *FT'/\epsilon$	{*, ε}	{+,), $}
$F \rightarrow id/(E)$	{id, (}	{*, +,), $}

($\epsilon' \rightarrow \epsilon$)

Parsing table : It's aim to construct ~~Pass~~ top-down parser.

	id	+	*	(	)	$ ← terminals
E	$E \rightarrow TE'$			$E \rightarrow TE'$		
E'		$E' \rightarrow +TE'$			$E' \rightarrow \epsilon$	$E' \rightarrow \epsilon$
T	$T \rightarrow FT'$			$T \rightarrow FT'$		
T'		$T' \rightarrow \epsilon$	$T' \rightarrow *FT'$		$T' \rightarrow \epsilon$	$T' \rightarrow \$$
F	$F \rightarrow id$			$F \rightarrow ($		

↑ variables

($E \rightarrow id$)

- if word start with id ~~then we~~ and production is E so we must have to use production $E \rightarrow TE'$.
  (bcz First of T is (id & ()
- Whenever the first of any variable is ε then we will place that production in follow of that variable (left hand side).
  eg in the case of $E' \rightarrow \epsilon$ we will put this production in), $
- This table is used to construct the parse tree.

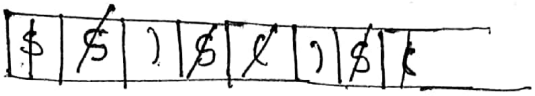
eg $S \rightarrow (S)/\epsilon$

	follow	first
S	{$,)}	{(, ϵ}

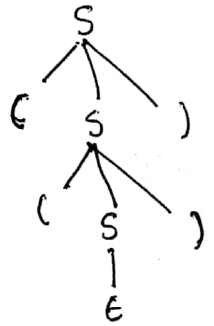
LL(1) table

	(	)	$
S	S→(S)	S→ϵ	S→ϵ

eg derive the string $w = (())\$$



parse tree



∴ it will accepted.

eg $S \rightarrow AaAb/BbBa$

$A \rightarrow \epsilon$

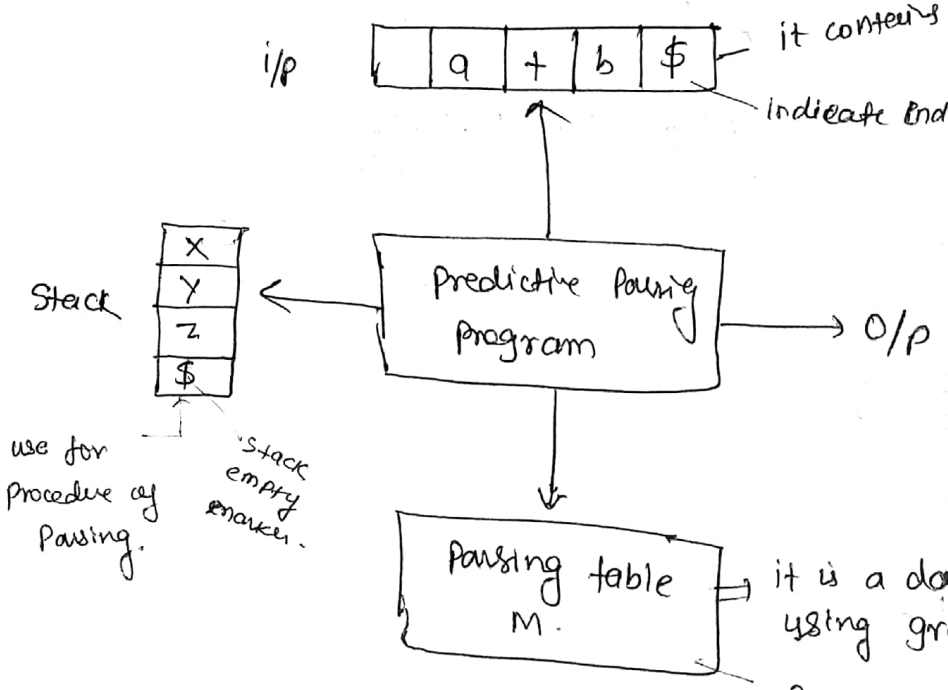
$B \rightarrow \epsilon$

	a	b	$
S	S→AaAb	S→BbBa	
A	A→ϵ	A→ϵ	
B	B→ϵ	B→ϵ	

- if two production entry will be same cell so that grammar won't be LL(1).

Non recursive predictive Passing (LL(1))

- it maintaining a Stack explicitly rather than implicitly via recursive calls.
- The key problem during predictive parsing is that of determining the production to be applied for a non terminal.



Possibilities Corresponding to 2 Symbol.

1) if $X = a = \$$ the Parser halt and announces successful Completion of Parsing.
2) if $X = a \neq \$$ the Parser POP X off the Stack and advances the i/p Pointer to the next i/p Symbol.
3) if X is a non terminal, the program consult entry $M[X, a]$ of the Parsing table M. This entry will be either an X-Production of the grammar or an error entry.

Algorithm:

set i/p to point to the first symbol of w$

repeat

1) let X be the top stack symbol & a be the symbol pointed
2) to by ip

3) if X is a terminal or $ then
4) if X = a then
5) pop X from the stack & advance i/p
6) else
error()

7) else /* X is a non terminal */
if $M[X, a] = X \rightarrow Y_1 Y_2 \cdots Y_K$ then begin
pop X from the stack
push $Y_K Y_{K-1} \cdots Y_1$ onto the stack with Y_1 on the top.
O/p the production $X \rightarrow Y_1 Y_2 \cdots Y_K$.
end
else
error()

until X = $

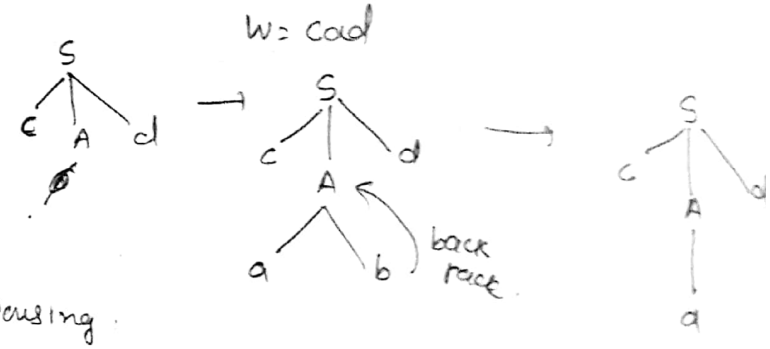
* Recursive-Descent Parsing :

- It is one of the top down parsing techniques that uses a set of recursive procedures to scan it's i/p.
- The parsing method may involve backtracking, that is making repeated scan of the i/p.

eg of backtracking.

$S \rightarrow cAd$

$A \rightarrow ab/a$



eg of Recursive descent parsing.

* A left recursive grammar can cause a recursive-descent Parser to go into an infinite loop.

Hence elimination of left recursion must be done before Parsing

eg

$E \rightarrow iE'$

$E' \rightarrow +iE'/\epsilon$

```
Procedure E()
begin
    if (l == 'i')          lookahead.
    { match('i');
      E'();
    }
end
```

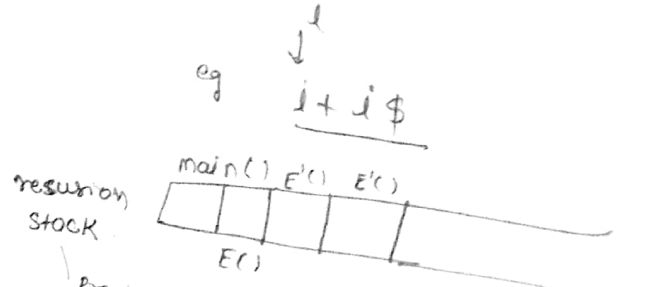
```
E'()
{ if(l == '+')
  { match('+')
    match('i')
    E'();
  }
  else
    return
}
```

Parsing of ϵ

```
match(char t)
{
  if (l == t)
    l = getchar()
  else
    Printf("error")
}
```

```
main()
{
  E();
  if(l == $)
    Printf("Parsing Success");
}
```

eg $i + i \$$



Bottom up Parsing

- Constructing a parse tree for an i/p string beginning at the leaves and going towards the root is called a bottom up parsing
- General type of bottom up parser is a shift reduce parser.