Compiler → which translates the high-level code to machine-level-code

→ Source program → **Compiler** → Target program

If the translation of source program to Target program in 1 step, it becomes complex
→ so we have 6 phases

1. lexical analyzer
2. syntax analyzer
3. Semantic analyzer
4. Intermediate code generator
5. code optimizer
6. code generator

lexical analyzer → is also called scanner
→ takes source program as the input

eg:- //A statement in source program//

sum = sum + x ;

is given to lexical analyzer

lexical analyzer

which divides all the statements into individual logical entities into

Tokens → Identifier, Keywords, Operators, Constants, Punctuation marks, Special characters, Comments

So the output of lexical analyzer is stream of tokens

* In order to recognize the tokens within the source Program a finite automata is implemented in the lexical analyzer

Syntax analyzer → takes stream of tokens as the input from the lexical analyzer

Stream of Tokens → Syntax analyzer → Converted into / combined into → Syntactic structure → hierarchial structures → If valid → Syntax Trees

These syntactic structures are represented as

* syntax analyzer is also called parser.
* So syntax analyzer gives syntax trees as output
* Syntactic trees are generated only when there is a valid syntactic structure
* we follow two parsing techniques to check the validity of syntactic structures
  1. Top-down parsing technique
  2. Bottom-up parsing technique (efficient)

Semantic analyzer → takes input from the syntax analyzer: syntax trees

* functionalities of semantic analyzer
  1. Type checking → operator compatible operands are present or not
  2. uniqueness check → we cannot declare more than one variable with same name under a data type.
  3. Name-related check
  4. flow control check

Case / Program / function /

In Ada, subprograms are represented using Procedures

Procedure name-of-Procedure
— — —
Procedure Nested-Procedure
— — —
end nested-procedure
End name-of-procedure

It checks the, whether the mentioning of procedure name in end procedure is there are not

* output of semantic analyzer is Syntax tree (with type checking functionality)

$\boxed{\text{Intermediate code generator}}$ → Takes input as Syntax trees from Semantic analyzer

* Intermediate code forms
  1. Polish Notation (Prefix) [Reverse postfix notation]
  2. Syntax tree
  3. Three Address code

* Output of Intermediate code generator is three address code, as it becomes easy to generate the code

$\boxed{\text{Three address code}}$ → A statement which is having atmost three operands.
It can have any operators (any one) including the assignment operator.

eg: $x=y+3$; ✓
$x=y$; ✓
(no assignment) $x+y*3$ ✗

* we can convert any statement which is not in three address code to three address code using operator precedence rules & Associativity rules [manually]. → But compiler uses internally a different technique "Syntax directed Translation"

eg: $x+y*3$
∴ $T_1=y*3$
$T_2=x+T_1$ ✓

* Output of Intermediate code generator is Three address code

$\boxed{\text{Code optimizer}}$ → takes Three address code as input

* Code optimizer reduces the no. of instructions by therefore optimizing the code and occupying less space
* There will be two code optimization techniques
  ① machine – Independent code optimization.
  ② machine – dependent code optimization.
* Output of code optimizer is optimized three address code

$\boxed{\text{Code generator}}$ → It takes optimized code as input and it produces the assembly code

↓ source program
[Lexical analyzer] — (finite automata)
↓ stream of tokens
[Syntax analyzer] — (Parsing)
↓ syntax tree
[Semantic analyzer] — (Type checking)
↓ syntax tree (type checking)
[Intermediate code generators]
↓ TAC
[Code optimizer]
↓ optimized TAC
[Code generator]
↓
assembly code

▸ Assume a statement in the program is given to the lexical analyzer

Position:= initial + rate * 60

↓
Then the lexical analyzer identifies each character & produces a stream of tokens using a finite automata

According to identifiers it stores identifiers & constants in symbol table
↓
+, * already exists within the operators table

within the symbol table

mem.loc	ident
1	Position
2	initial
3	rate
4	60

↓
So the given statement is converted as

id1:= id2+id3 * 60

If the identifiers repeat in statement then that particular identifier is only inserted once.

↓ Output of lexical analyzer will not be available in expression form

And id1 := id2 + id3 * 60 is given as input to syntax analyser

which constructs context free grammar for each statement by syntax analyser

then the CFG Generated will be

$S \rightarrow id := E$
$E \rightarrow E + E$
$E \rightarrow E * E$
$E \rightarrow id \,/\, num$

→ $S \rightarrow id := E$
$\boxed{E \rightarrow E+E \,/\, E*E \,/\, id \,/\, num}$
(ambiguos grammar)

Then the syntactic structure is derived either by left most derivation or right most derivation.

Therefore S, E are non terminals and deriving

$S \rightarrow id := E$
$\rightarrow id1 := E + E$
$\rightarrow id1 := id_2 + E$
$\rightarrow id1 := id2 + E * E$
$\rightarrow id1 := id2 + id3 * E$
$\rightarrow id1 := id2 + id3 * num$

Then the above syntactic structure is valid or not !: verified by two approaches Topdown approach & bottom up approach

Output of syntax analyser is $\boxed{\text{Syntax tree}}$

Syntax tree: A tree is called syntax tree, if interior nodes are represented by operators, leaf nodes are represented by identifiers (or) constants

Construction of syntactic structure: syntax tree

* we convert id1 := id2 + id3 * 60 into postfix notation

$id1 := id2 + \underbrace{id_3 * 60}_{T_1 = id_3\, 60 *}$

$\underbrace{\qquad\qquad\qquad}_{T_2 = id_2 T_1 +}$

$T_3 = id_1 T_2 :=$
$T_3 = id_1 id_2 T_1 + :=$
$T_3 = id_1 id_2 id_3 60 * + := $ (Postfix)

and expression is evaluated using a Stack

To generate tree we use 3 functions

① mknode (op, LP, RP) : for operators
② mkleaf (id, entry) : for identifiers
③ mkleaf (num, value) : for constants

entry : is place/mem.loc where identifier are stored.

∴ $id_1 id_2 id_3 60 * + :=$

* mkleaf (id, 1) creates a node and returns a address

∴ P_1 = mkleaf(id, 1)

Similarly

P_2 = mkleaf(id, 2)
P_3 = mkleaf(id, 3)
P_4 = mkleaf(num, 60)

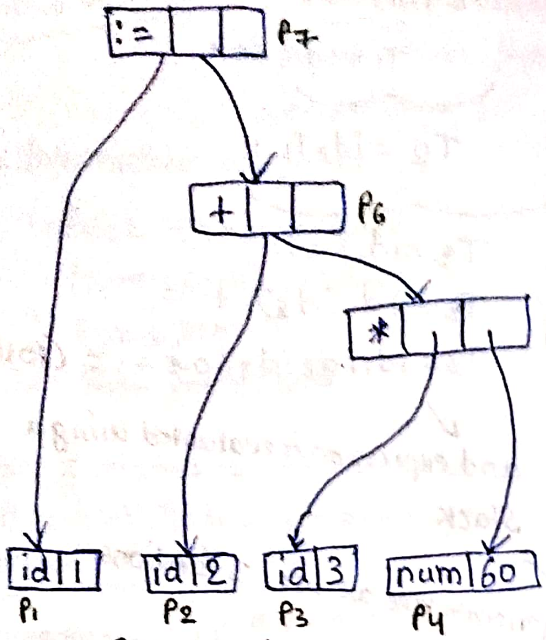
* .

P_5 = mknode (*, P_3, P_4)
id3 num

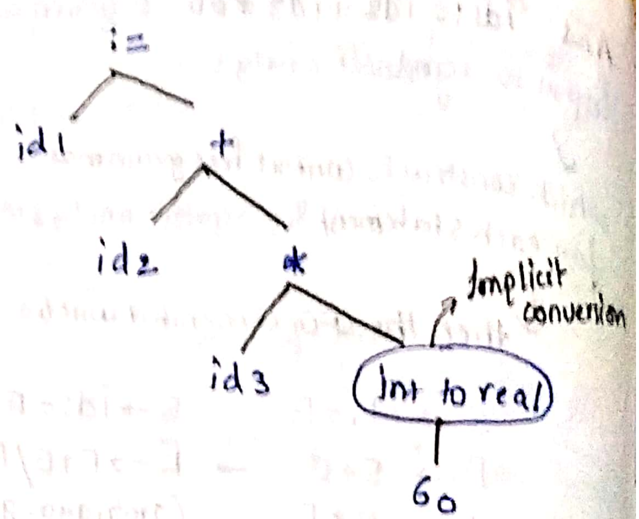
P_6 = mknode (+, P_2, P_5)
where id3 * 60 is considered

P_7 = mknode (:=, P_1, P_6)

we use functions in order of Postfix notation



Syntax tree.

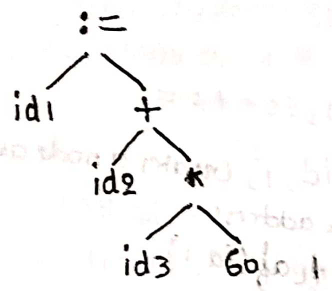


This is given as Output to Intermediate Code generator.

↓

Then it should be Converted into three address code (Intermediate code)

Abstract syntax tree : A syntax tree is called abstract syntax tree, it we represent corresponding symbol at each node.



↓

This is given as an input to Semantic analyzer.

It verifies operator compatabile Operands are applied or not (Type checking)

↓

within the output of Semantic analyzer, we reflect the implicit Conversions.

Taking the statement and reflecting implicit conversions

↓

where lower data type Calculations are converted into higher datatype automatically → As there are no mnemonics in assembly language to perform operations on mixed mode expression

float int float
$a = b * c$
float

T_1 = int to real (60)
$T_2 = id3 * T_1$
$T_3 = id2 + T_2$
$id_1 = T_3$

→ we cannot write $T_1 = id3 *$ int to real because is itself is considered as an operator

↓

Typecasting has the highest precedence when compared to other operators

This three address code is given as an input to the code optimizer.

↓

Then it carries out the Conversions and replace with the converted value

↓

$T_2 = id3 * 60.0$
$id_1 = id2 + T_2$

↓

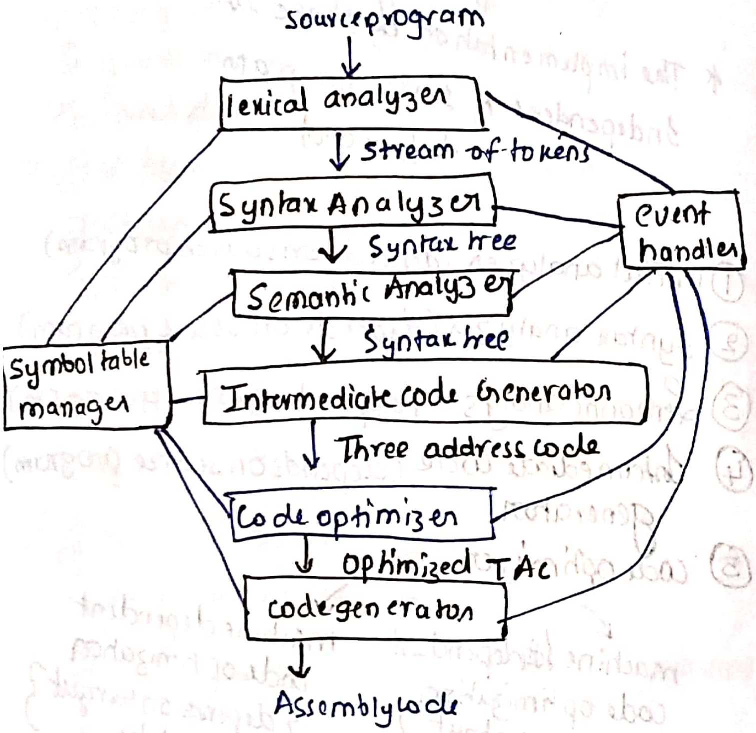
This optimized code is given as input to the Code generator.

↓

And it produces an Assembly code

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

logical phases of compiler

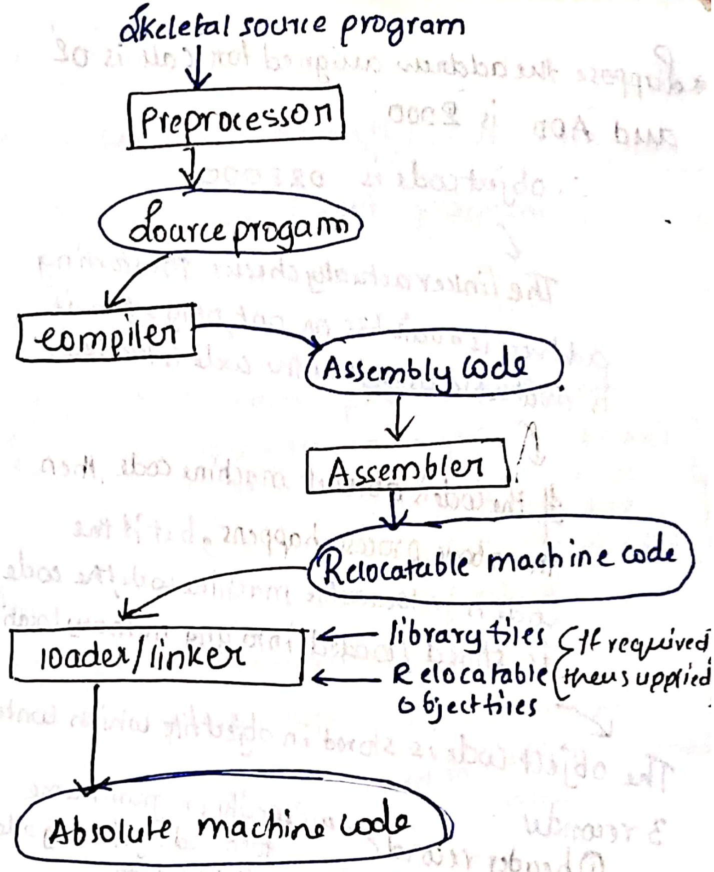


Symbol table manager: It is a datastructure which is used to store the user defined information like identifiers and constants.

> If Identifier is a variable, then it stores the information like name, address, value, type, lifetime and scope.

> If identifier is an array, it stores informations like name of array, type of array, index type, range of array.

> If identifier is a function, it stores the information like name of the function, no. of parameters, return type of the function.



* Assume a statement in Assembly code

Call ADD → function — opp code

Call ADD

ADD
begin
⋮
end

* for each statement address is allocated and object code is generated
* If the starting address is defined then it is taken as starting address. Otherwise it takes '0' as starting address.

> Call ADD
> MOV A,R → is second statement, then the assembler calculates the length of instructions on memory locations of call ADD and takes the next address and assigns it to the second statement.

If call ADD

0000	
0001	Call ADD
0002	

→ then the second statement starts occupying from 0003 → Assembler also generates the object code for each statement.

Suppose the address assigned for call is 02 and ADD is 2000

∴ object code is 022000

The linker actually checks the starting address is available or not, only when it is available only then the code is loaded.

If the code is absolute machine code, then the above process happens, but if the code is relocatable machine code, the code is stored / loaded into any memory location.

The object code is stored in object file which contains 3 records

1. header record → we specify program name followed by starting address of program
2. text record → for each statement object code is generated. (eg: call Add → 022000) and stored in text record
3. end record

end record contains the starting address of program and control is transferred to starting address and the execution starts

Relocatable machine code also consists of extra record

4. modification record → not available in absolute machine code.

If the specified starting address is unavailable, then some other free memory locations are used to store the code and this information is stored in modification record.

loader & linker actually can check which memory locations are free

grouping of phases

front end & backend | Pass

* The implementation / phase depends on the source program & independent of Target program

↓ front end

* The implementation of phase is independent of source program

↓ backend

1. lexical analyzer (depends on source program)
2. syntax analyzer (depends on source program)
3. semantic analyzer (depends on source program)
4. Intermediate code generator (depends on source program)
5. code optimizer

machine independent code optimization (donot care about machine target characteristics) → depends on source program

machine dependent code optimization {depends on target machine} → Independent of source program

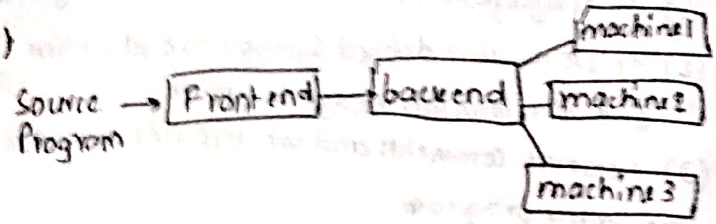
6. Code generator (depends on target machine)

Front end → Syntax analyzer, lexical analyzer, semantic analyzer, Intermediate code generator, machine Independent code optimization

back end → machine dependent code optimization, Code generator

Advantages of grouping phases

(1)



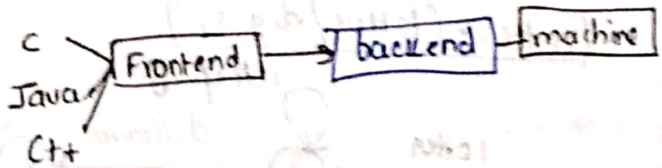
> Supposing a new programming language is being developed. Source program is given as input to front-end of the compiler design

> But by designing compiler; using grouping of phases. front end maybe similar to another already developed programming languages and you can include its API

> And the backend can be taken good care for designing it.

> If there are no grouping of phases into front end & backend, we need to design compiler (all phases) for heterogeneous machines

(2) By grouping of phases, for similar programming languages you can use the same front end.



and sometimes you can use the same backend too.

And this is very tedious to develop same compiler for similar programming languages.

Pass

* It is a physical scan over the source program i.e, number of times a source program is scanned.

(or)

* grouping one or more phases is called Pass { Can be explained using legacy rams (128 KB) & compiler needs to be loaded into memory; compiler size > ram size, therefore they grouped one (or) more phases into pass }

> Implementation of the phases, of whether it contains one pass (or) multi pass compiler depending upon features provided by source program.

Statement: goto target — is defined in symbol table

target: mov A, R_1

> Suppose goto has an object code is 03

> Target statement comes ~~after~~ before goto, so ~~Compiler doesn't know the object code of target statement yet~~ Stores target in symbol table

e.g. 1000 is assigned Address for target

∴ object code = 031000

> If target statement is placed ~~before~~ after goto statement (forward reference)

then object code = 03 $\boxed{}$ ← slot is assigned in this case

after

As target comes ~~before~~ goto, it places target statement in symbol table

target	2000

X wrong

vice versa

Case1 :

target Statement : mov A, R1 ;

Goto statement : goto target ; already defined in symbol table

In here target statement is before the goto statement.

Consider object code of statement mov A,R1 is 1000

and it is stored in the symbol table

next statement is

goto target ;

and object code of above statement is 03

∴ the object code of target statement is substituted here

031000 (backward reference)

Case2 :

goto statement : goto target already defined in symbol table

target statement : mov A, R1

In here target statement is after the goto statement

consider object code of goto statement is 03

The object code of goto statement is 03 and object code of target is not yet computed

hence a slot is created 03 $\boxed{\quad\quad}$

after target Object code is calculated the value is put in slot

(forward reference)

$\boxed{\text{Role of lexical analyzer}}$

(1) It recognizes the tokens in the source program

(2) It stores user defined symbols like identifiers and constants into the symbol table

(3) Removes comments and whitespaces within the source program.

(4) Keeps track of the line number to associate with every message

(5) Identifies the lexical errors

$\boxed{\text{Token}}$ It describes class or category of a input state

ex : identifiers, constants, keywords, operators comments etc.

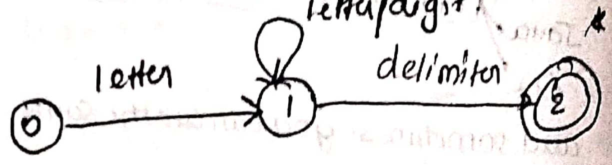
$\boxed{\text{Pattern}}$: A set of rules that describes a token

Note : Patterns are represented using regular expressions.

$\boxed{\text{lexeme}}$: Sequence of characteristics that is a form a input string which matches with the pattern of a token

$\boxed{\text{Regular expression for Identifier}}$

$\boxed{\text{letter . (letter / digit)*}}$



* when actually reading the identifier an extra character (as for =) and we perform retract operation.

* each state is considered as a procedure to write the code.

Code for the state '0' :-

1. c = getchar();
2. If LETTER(c) then
3. goto state 1; Else; FAIL();

This donot generate error immediatly, but sends control to the next finite automata

Code for the state '1' :-

1. c = getchar();
2. If LETTER(c) or DIGIT(c) then
3. goto state 1;
4. Else if DELIMITER(c) then
5. goto state 2;
6. Else Fail();

Fail() function donot immediatly generates the lexical error but transfers the control to the next finite automata.

Code for the state '2' :-

1. RETRACT();

So the RETRACT() function eliminates extra space from the identifier

2. return (id, Install());

it is used to stored the recognized identifier within the symbol table.

ex: If sum=sum+x;

∴ In symbol table and location is allotted to it

1	sum	

symbol table

then Install function returns 1

∴ return (id, 1);

Token	value	code
BEGIN	1	—
END	2	—
IF	3	—
THEN	4	—
ELSE	5	—
identifier	6	Pointer to symbol table
constant	7	Pointer to symbol table
<	8	1
<=	8	2
=	8	3
<>	8	4
>	8	5
>=	8	6

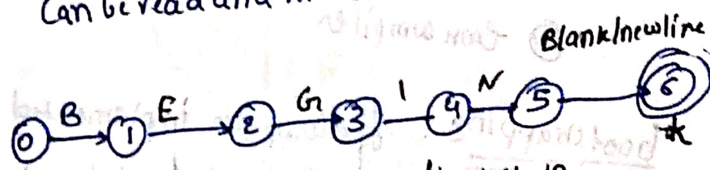
* Syntax Analyzer identifies the Tokens using their corresponding values.
* when lexical analyzer identifies the token it returns the equalent integer values to the syntax analyzer
* The values may vary for any token.

∴ for an identifier it returns (6, 1);

Finite automata for begin

> state '0' :- B
> state '1' : E
> state '2' : G
> state 3 : I
> state 4 : N

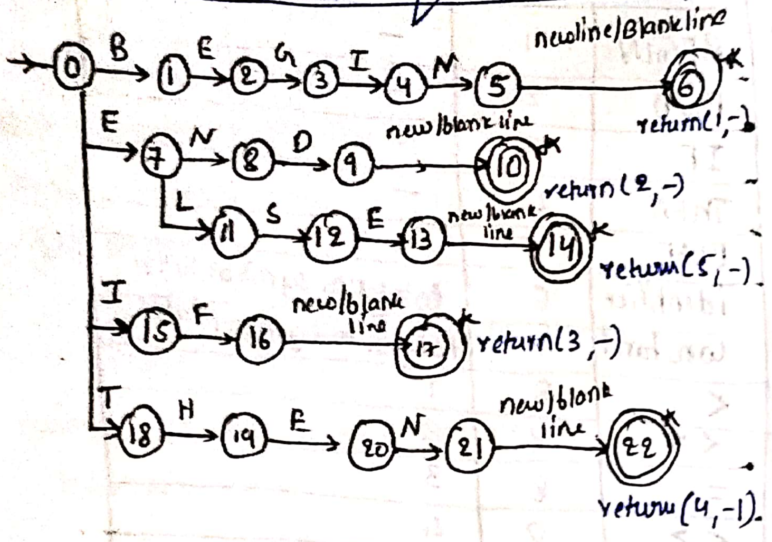
after state 4 any other character (if present) can be read and makes it as an identifier



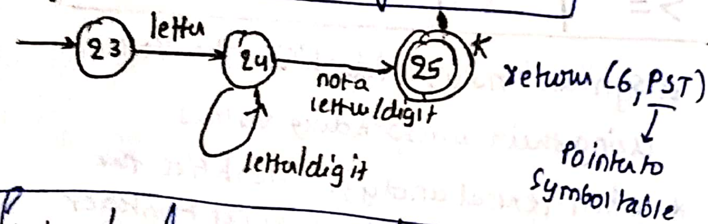
• If the control donot reach final state then it calls fail function and it moves control to next finite automata

* → represents retract operation

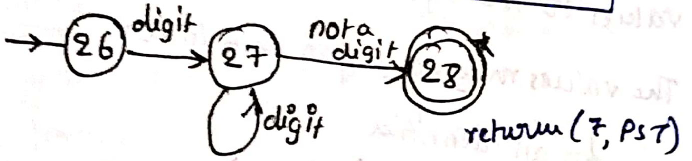
Similarly drawing the finite automata for others (Keywords)



Similarly finite automata for identifiers



Similarly finite automata for Constants



Input Buffering | Lexical analyzer generator | Regular definition

Bootstrapping:-

Consider a language 'L' that do not have any compiler available in any machine.

So we use 2 methods

① Bootstrapping

② Cross compiler

bootstrapping: If a compiler implemented in its own language is called bootstrapping.

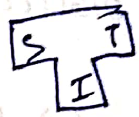
Cross compiler:- A compiler which runs on one machine produces object code for another machine.

① Source program.

② Implementation language

③ Target language

★ Characteristics of compiler represented in T form



(or) can also be represented like this

$C^{S\ T}_{\ I}$

> First we design the compiler for subset features of 'L'. (Language - source), as it is hard to develop a compiler to cover all features of language, so we take common features

$C^{S\ A}_{\ A}$ A → machine

> Now we design compiler for language L

$C^{L\ A}_{S} \rightarrow C^{S\ A}_{A} \longrightarrow C^{L\ A}_{A}$ → object code

ex: C A → machine language

we take S C C
subset features are

$C^{SC\ A}_{C} \rightarrow C^{C\ A}_{SC} \longrightarrow C^{C\ A}_{ML} \rightarrow C^{C\ A}_{A}$

compiler designed for machine language for machine A for subset features of C

compiler designed for machine language for machine A for features of C

A produces object code for A in object code cross compiling

$C^{C\ B}_{C} \rightarrow \boxed{C^{B}_{A}} \rightarrow C^{C\ B}_{B}$

II Phase of compiler

> Imagine a statement is given as an input to lexical analyzer

sum = num+x ;

↓

$\boxed{\text{Lexical analyzer}}$

↓ stream of tokens

$\boxed{\text{Syntax analyzer}}$

Syntax analyzer combines stream of tokens into syntactic structures.

In order to convert the stream of tokens into syntactic structures it uses ~~automata~~ (CA) CFG is defined

let s is a statement

$S \rightarrow id = E$

$E \rightarrow E+E$

$E \rightarrow E-E$

$E \rightarrow E*E$

$E \rightarrow E/E$

$E \rightarrow id$

for each program statement a CFG is defined.

using the CFG we must derive below result

(id1 = id2 + id3)

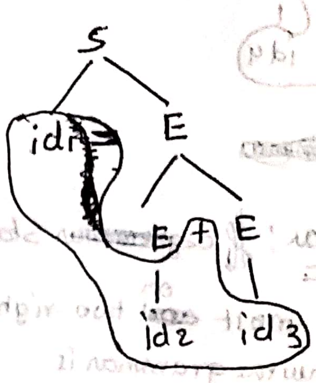
using leftmost, rightmost derivation

$S \rightarrow id_1 = E \quad (S \rightarrow id = E)$

$\rightarrow id1 = E+E \quad (E \rightarrow E+E)$

$\rightarrow id1 = id2+E \quad (E \rightarrow id)$

$\rightarrow id1 = id2+id3 \quad (E \rightarrow id)$



$id_1 = id_2 + id_3 * id_4$

$S \rightarrow id_1 = E$ (left-most)

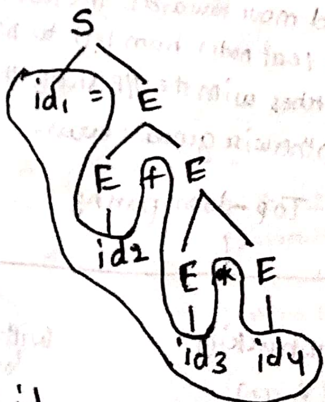
$\rightarrow id1 = E+E$

$\rightarrow id1 = id2+E$

$\rightarrow id1 = id2+E*E$

$\rightarrow id1 = id2+id3*E$

$\rightarrow id1 = id2+id3*id4$



$S \rightarrow id = E$

$S \rightarrow id = E+E$

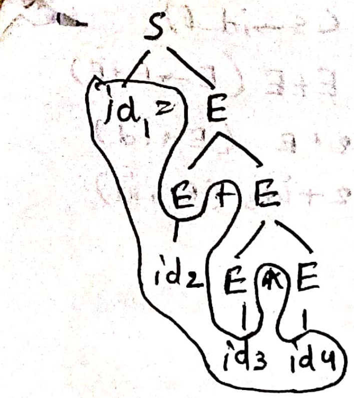
$S \rightarrow id = E+E*E$

$S \rightarrow id = E+E*id4$

$S \rightarrow id = E+id3*id4$

$S \rightarrow id = id2+id3*id4$

$S \rightarrow id_1 = id_2+id_3*id_4$



Ambigious gramar: If a ~~[illegible]~~ Statement contains two left-most or two-right most parse trees, then the grammar is called Ambigious grammar

(OR)

If there exist more than one parse trees then it is called Ambigious grammar

① Top-down-parsing :- It constructs the parse tree.

It takes the starting non-terminal as root node and move towards the leaf nodes. Scan the leaf nodes from left to right. If it matches with the i/p string then it accepts otherwise generates error.

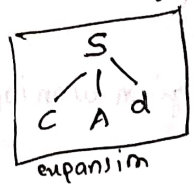
Top-down parsing
- with backtracking (Brute-force)
- Without backtracking
  - Recursive descent parser
  - non-recursive decent Parser (Predicate Parser)

Consider the grammar

$S \rightarrow cAd$

$A \rightarrow ab/a$

* within the S we take an input pointer (pointing to c) and store it in temporary ptr before expansion $\{IP = TP\}$



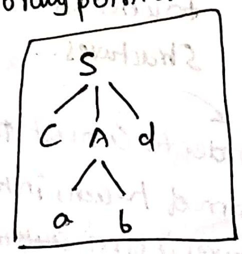
expansion

'c' in expansion is matched with i/p pointer $\boxed{cAd}$

> next symbol within the expansion is A and compared with the i/p pointer $\boxed{cAd}$ ∴ i/p pointer is incremented

and A has two Productions

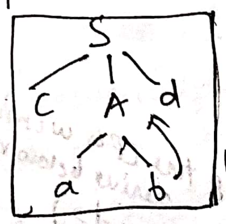
* before expansion we take an input pointer (pointing to a) and store it in temporary pointer.



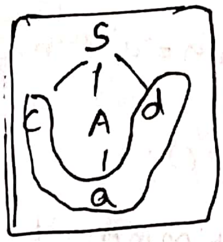
'a' in expansion matches with the i/p pointer ∴ i/p pointer is incremented

> and next time the match fails, hence we backtrack to the A (Previous)
when we backtrack to A we restore the input pointer. $\{IP = TP\}$

> expanding the a with second production



> then the next symbol is compared against the i/p pointer 'a'



and it is matching with the i/p pointer

So it accepts the syntactic structure.

ex:-

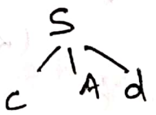
⇒ Taking the string that needs to be validated $\boxed{cad}$

CFG = { cabd, cad }

> It starts with 's' starting symbol as root node in parse tree that needs to be validated

& before expansion we store {IP = TP} input pointer in an temporary pointer. $\boxed{cad}$

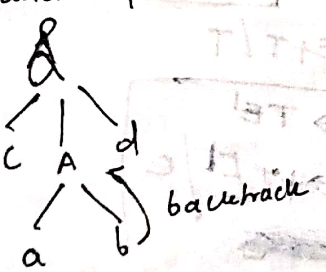
∴ expanding s using backtracking method



> input pointer is matched with the 'c' in expansion ∴ i/p pointer is incremented

> so now TP points to A then for A {IP = TP} $\boxed{cad}$

A has another productions

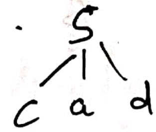


> It backtracks to root node, if there is no alternate production and i/p pointer is restored

∴ i/p pointer points to 'c'

∴ Another production of s = $\boxed{cad}$

∴



$\boxed{cad}$ {IP = TP}

'c' matches with i/p i/p pointer++

$\boxed{cad}$ {IP = TP}

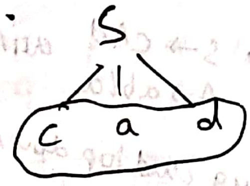
'a' matches with i/p

i/p pointer++

$\boxed{cad}$ {IP = TP}

'd' matches with i/p

∴

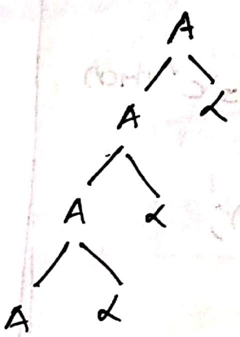


∴ it accepts

Disadvantages of backtracking :-

① Consider

A → A α / β



we only go to second production if we find unmatched result.

Infinite loop will occur in ~~loop~~ with topdown parsing backtracking and in left recursion.

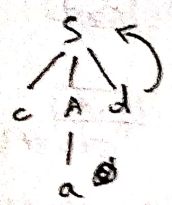
So after eliminating left recursion

$\boxed{\begin{array}{l} A \to \beta A' \\ A' \to \alpha A' / \epsilon \end{array}}$ (refer in flat)

② $S \to cAd$
$A \to a/ab$

string : cabd

Backtrace method & recursive descent parser would not predict exact right hand side of production for a non terminal having more than one productions while parsing a string but non-recursive descent parser will predict.



S

S should accept cabd, but it doesnt, because it is in different order of productions

~~If (order A → ab/a) then it accepts both cad and cabd~~
maybe true maybe not

③ It is very time taking process

⇒ For the grammar $S \to cAd$ using
$A \to ab/a$

backtracking and top down approach code

∘ each non terminal is implemented using a procedure. {S, A}   string: $\boxed{cabd}$

```
procedure S ()
begin
  if input-pointer = 'c' then
    begin
      ADVANCE ();
    if A() then
      If input-pointer = 'd' then
      begin
        Advance();
        return true;
      end
    end
  return false;
end
```

```
Procedure A()
begin
  isave := input-pointer
  If input-pointer = 'a' then
  begin
    Advance();
    If input-pointer = 'b' then
    begin
      Advance();
      return true;
    end
  end
  input-pointer := isave;
  If input-pointer = 'a' then
  begin
    Advance ();
    return true;
  end
  else return false;
end
```

Eliminate the left recursion from the following grammar

$(\alpha, \beta, \gamma) \in (V \cup T)^*$

$E \to E+T/T$
$T \to T*F/F$
$F \to (E) / id$

Sol:- rule
$$\boxed{\begin{array}{c} A \to A\alpha / \beta \\ \downarrow \\ A \to \beta A' \\ A' \to \alpha A' / \epsilon \end{array}}$$

$E \to E+T/T$

$E \to TE'$
$E' \to +TE' / \epsilon$
[redacted]

11

$T \rightarrow T*F / F$

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

after eliminating the left recursion

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

eliminate left recursion from the following grammar

bexp → bexp or bterm / bterm
bterm → bterm and bfactor / bfactor
bfactor → not bfactor / true / false

Eg: bexp → bexp or bterm / bterm

bexp → bterm bexp'
bexp' → or bterm bexp' / ∈

bterm → bterm and bfactor / bfactor

bterm → bfactor bterm'
bterm' → and bfactor bterm' / ∈

after eliminating

bexp → bterm bexp'
bexp' → or bterm bexp' / ∈
bterm → bfactor bterm'
bterm' → and bfactor bterm' / ∈

bfactor → not bfactor / true / false

multiple left recursions

$A \rightarrow A\alpha_1 / A\alpha_2 / A\alpha_3 / \ldots / A\alpha_n / \beta_1 / \beta_2 / \beta_3$

$\downarrow$

$A \rightarrow \beta_1 A' / \beta_2 A' / \ldots / \beta_n A'$
$A' \rightarrow \alpha_1 A' / \alpha_2 A' / \ldots / \alpha_n A' / \epsilon$

eliminate left recursion

$S \rightarrow Aa / b$
$A \rightarrow Ac / Sd / e$

$A \rightarrow Ac / Sd / e$

$A \rightarrow SdA'$
$A' \rightarrow cA' / \epsilon$

$A \rightarrow eA'$
$A' \rightarrow cA' / \epsilon$

$S \rightarrow Aa / b$
$A \rightarrow SdA' / eA'$
$A' \rightarrow cA' / \epsilon$

Substituting the sd in $S \rightarrow Aa / b$
it becomes $S \rightarrow Sda / b$

$\therefore$ $S \rightarrow bS'$
$S' \rightarrow daS' / \epsilon$

To avoid the confusion, we use the algorithm.

Algorithm to eliminate left recursion

step 1: Arrange non terminals in some order such as $A_1, A_2, \ldots A_n$

step 2: for $i=1$ to n do
begin
for $j=1$ to n do → no. of nonterminals
begin if $j<i$ then
replace each production $A_i \to A_j \gamma$
with
$A_i \to \delta_1\gamma / \delta_2\gamma / \ldots / \delta_k\gamma$ where
$A_j \to \delta_1 / \delta_2 / \ldots / \delta_k$ are current
A_j productions

end

eliminate immediate left recursion
among A_i productions

end

end

problem: $S \to Aa/b$
$A \to Ac/Sd/e$

Sol: rename non terminals with $A_1 \ldots A_n$

$S \cdot A$
$\downarrow \quad \downarrow$
$A_1 \quad A_2$

∴ Productions

$A_1 \to A_2a/b$
$A_2 \to A_2c/A_1d/e$

So $j<i$
∴ $A_2 \to A_1d$ (satisfies the condition)

∴ replace

$A_1 \to d$

> replacing the A_1 productions

$A_2 \to A_1d$
$A_2 \to A_2ad/bd$

∴ Productions are

$$\boxed{\begin{array}{l} A_1 \to A_2a/b \\ A_2 \to A_2c/A_2ad/bd/e \end{array}}$$

~~$A_2 \to A_2c$~~	~~$A_2 \to A_2ad$~~
~~$A_2 \to A_2'$~~	~~$A_2 \to A_2'$~~
~~$A_2' \to cA_2'$~~	~~$A_2' \to adA_2'$~~

∴ eliminating immediate left recursion

$A_2 \to bdA_2'$	$A_2 \to eA_2'$
$A_2' \to adA_2'/\epsilon$	$A_2' \to adA_2'/\epsilon$

$A_2 \to A_2c$ elimination

$A_2 \to bdA_2'$	$A_2 \to eA_2'$
$A_2' \to cA_2'/\epsilon$	$A_2' \to cA_2'/\epsilon$

∴ productions are:

$$\boxed{\begin{array}{l} A_1 \to A_2a/b \\ A_2 \to bdA_2' / eA_2' \\ A_2' \to adA_2'/cA_2'/\epsilon \end{array}}$$

⇒ Check ambiguity and write unambiguous

$R \to R+R/R \cdot R/R^*/a/b$

• deriving a+ba

LMD:

$R \to R+R$
$\to a+R$
$\to a+R \cdot R$

$\rightarrow a+b\cdot R$

$\rightarrow a+ba$

LMD2

$R \rightarrow R\cdot R$

$R \rightarrow R+R\cdot R$

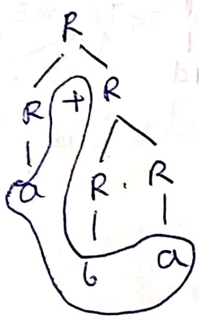
$R \rightarrow a+R\cdot R$

$R \rightarrow a+b\cdot R$

$R \rightarrow a+ba$

∴ ambigious, as there exist two left most derivations

Parse Tree LMD1



Parse Tree LMD 2

∴ The unambigous grammar is:-

Operator precedence rule

① left Associativity → we have to define in left recursion

② Right Associativity → we have to define in right recursion

* the highest precedence operators are defined away from the starting non terminal
* the lowest precedence operator are defined nearest to the starting non terminal

∴ modifying the grammar

unambigious grammar

$R \rightarrow R+T / T$

$T \rightarrow T\cdot E / E$

$E \rightarrow E^* / a / b$

operator precedence

*
·
+

* The below grammar was ambiguous, as the operator precedence was not defined clearly.

Top-down parsing
- with backtracking (Brute force method)
- without backtracking
  - Recursive Decent Parser
  - non-recursive descent Parser

→ In recursive decent parser without backtracking we use one procedure for each non-terminal.

$E \rightarrow TE'$

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

$T \rightarrow FT'$

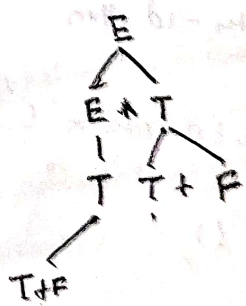
$T' \rightarrow *FT' / \epsilon$

$F \rightarrow id / (E)$

$E \rightarrow E*T / T$

$T \rightarrow T+F / F$

$F \rightarrow id$



```
Procedure E()
  begin
      T();
      EPRIME();
  end
```

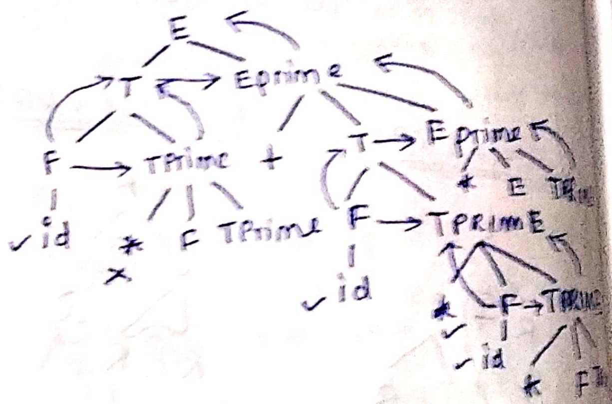
```
Procedure EPRIME()
begin
  If input-pointer = '+' then
    begin
      ADVANCE();
      T();
      EPRIME();
    end
end
```

```
Procedure T()
begin
    F();
    TPRIME ();
end
```

```
Procedure TPRIME ()
begin
  If input-pointer = '*' then
    begin
      ADVANCE();
      F();
      TPRIME();
    end
end
```

```
Procedure F()
begin
  If input-pointer ='id' then
  ADVANCE().
  else if inputpointer = 'C' then
  begin
    ADVANCE ();
    E();
    If input-pointer = ')' then
      ADVANCE();
    else
      ERROR;
  end
  else
    ERROR;
end
```

> we can generate id + id * id from above grammar



* Consider a Production

A → abcd | abce

∴ Code is

```
Procedure A ()
begin
  if input_pointer = 'a' then
      Advance()
        if Input-pointer = 'b' then
        Advance()
          if input-pointer = 'c' then
          Advance ()
            If input-pointer = 'd' then
            Advance ()
```

```
else if ip='a' then
    advance()
    if input-pointer='b' then
        advance()
        if input-pointer='c' then
            advance()
            if input-pointer='e' then
                advance
            else
                error;
    end
else
    error;
end
```

As we check the same condition twice, we reduce the grammar to

$$\boxed{\begin{array}{c} A \to abcd / abce \\ \downarrow \\ A \to abcA' \\ A' \to d/e \end{array}}$$

while using recursive descent parser

So if a production is in form $A \to \alpha\beta / \alpha\gamma$ then it is called left factoring

→ After eliminating left factoring the productions are

$$\boxed{\begin{array}{l} A \to \alpha A' \\ A' \to \beta/\gamma \end{array}}$$

→ eliminate left factoring

$C \to iCtS \mid iCtS\,eS / a$

$C \to b$

∴

$S \to iCtS\,S' / a$

$S' \to eS$

$C \to b$

∴

After eliminating

$$\boxed{\begin{array}{l} S \to iCtS\,S' / a \\ S' \to eS / \epsilon \\ C \to b \end{array}}$$

Non-recursive decent parser

* It is also called as the predictive parser

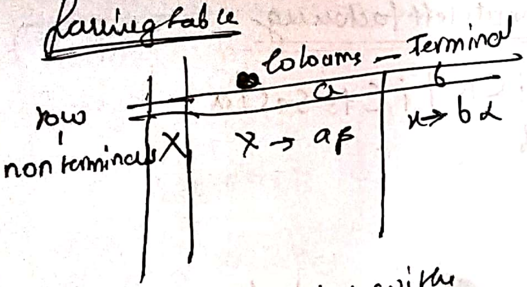
Suppose input pointer is pointing to 'a' in input string and productions are

$$X \to b\alpha / a\beta$$

* In backtracking and recursive decent parser the first production is expanded first ($b\alpha$)
* but in non-recursive decent parser, as the ip pointing to a and 'a' is available in 2nd production, it expands 2nd production first
* Parsing table is a 2 dimensional array which is used by non-recursive decent parser to predict which production to be expanded

$ab+ac = a(b+c)$

Parsing table



So X is expanded with

$X \rightarrow a\beta$

→ for predictive parser there are some mandatory prerequisites.

① Block diagram
② FIRST rules
③ Follow Rules
④ Algorithm predictive parser table

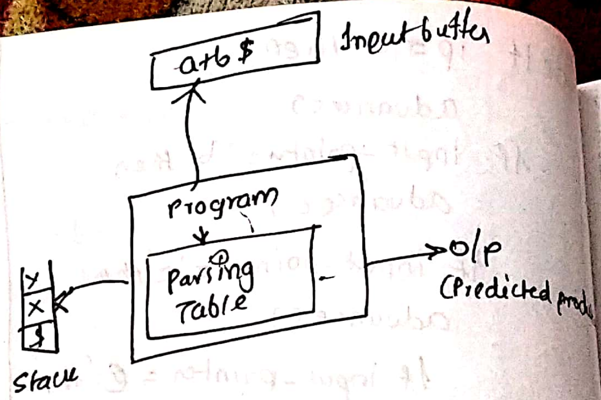
Block diagram for predictive parser / stack implementation

• Input buffer :- within Input buffer it will contain Input string & ending with $

• stack :- within the stack, we push & pop the grammar symbols (VUT) and without any symbols, stack defaultly contains $.

* Predictive parser actions are controlled by the program.

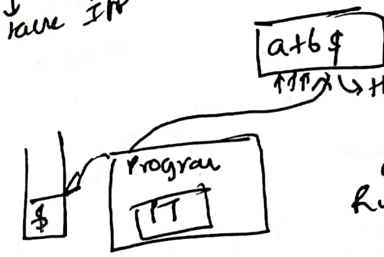
• Parsing table → used to predict which production needs to be expanded



Case(i)
Ex:

$X = a = \$$

↓ stack ↓ I/P



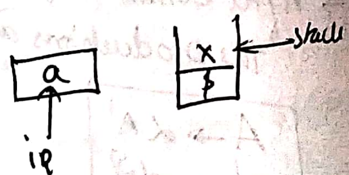
↳ then program halts & announces successful Parsing of string (If stack_top = $)

Halt
if (stack_top == '$') && (ip == '$') then
if (stack_top = 'a') && (ip = 'a') then pop()
{
increment input pointer
So that program points next symbol
}

Case(ii)
Ex: $X = a \neq \$$

then pop() && ip ++;

Case(iii)
Ex: $X \neq a \neq \$$

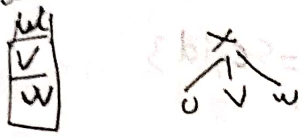


∴ use parse table to decide which production to expand.

	a
X	$X \rightarrow UVW$

Consider $X \rightarrow UVW$ production

$\therefore$ In above case (iii) push U, V, W in order into the stack



First Rules :-

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

② If $X \rightarrow a\alpha$ is a production, where X is a non terminal

a is a terminal and

$\alpha \in (T \cup NT)^*$

$FIRST(X) = \{a\}$

and if $X \rightarrow \epsilon$ is a production then $FIRST(X) = \{\epsilon\}$

③ $X \rightarrow y_1 y_2 \dots y_k$ where y_i may be terminal (or) non-terminal

For each y_i production

$FIRST(X) = FIRST(y_1)$

If $FIRST(y_1) = \epsilon$ then

$\boxed{FIRST(X) = (FIRST(y_1) - \epsilon) \cup FIRST(y_2)}$

If $FIRST(y_2) = \epsilon$ then

$FIRST(X) = (FIRST(y_1) - \epsilon) \cup (FIRST(y_2) - \epsilon) \cup (FIRST(y_3))$

and so on

If $y_1 y_2 \dots y_k \overset{*}{\Rightarrow} \epsilon$ then also X include ϵ into FIRST(X)

ex:- for 3rd rule

Consider a grammar

$S \rightarrow AB$

$A \rightarrow a$

$B \rightarrow b$

Non terminals = {S, A, B}

- $FIRST(A) = \{a\}$
- $FIRST(B) = \{b\}$
- $FIRST(S) =$

as rule ③

$FIRST(S) = FIRST(A)$

$= \{a\}$

ex: $S \rightarrow AB$

$FIRST(A) = FIRST(a)$ rule(i)

$= \{a, \epsilon\}$

$FIRST(B) = FIRST(b)$

$= \{b\}$

$FIRST(S) = First(A) - \epsilon \cup First(B)$

$= \{a, \epsilon\} - \epsilon \cup \{b\}$

$= \{a\} \cup \{b\}$

$= \{a, b\}$

ex: $S \to AB\epsilon$
$A \to a / \epsilon$
$B \to b / \epsilon$

$\therefore$ $First(A) = \{a, \epsilon\}$
$First(B) = \{b, \epsilon\}$

$First(S) = First(A)$
$= (~~\{a, \epsilon\} - \epsilon) \cup (First(B)~~))$
$= ~~(\{a\} \cup \{b, \epsilon\}~~)$

$$\boxed{FIRST(S) = FIRST(A) - \epsilon \cup FIRST(B) - \epsilon \cup \epsilon}$$

$\therefore$ $FIRST(S) = (\{a, \epsilon\} - \epsilon) \cup (\{b, \epsilon\} - \epsilon) \cup \epsilon$

$$\boxed{FIRST(S) = \{a, b, \epsilon\}}$$

Compute FIRST for following grammar

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

~~$FIRST(E) = (FIRST(T) - \epsilon) (FIRST(E') - \epsilon)$~~

~~$\Rightarrow FIRST(E) = FIRST(F) = \{(, id\}$~~

$FIRST(+) = \{+\}$
$FIRST(*) = \{*\}$
$FIRST(() = \{(\}$
$FIRST()) = \{)\}$
$FIRST(id) = \{id\}$

$FIRST(E) = \{(, id\}$
$E \to TE'$
$E \to FT'E'$
$E \to (E)T'E' / idT'E'$
so include First terminal in First(E)

$FIRST(E') = \{+, \epsilon\}$
$E' \to +TE' / E' \to \epsilon$ rule
so include in First of E'

$FIRST(T) = \{(, id\}$
$T \to FT'$
$T \to (E)T' / idT'$

$FIRST(T') = \{*, \epsilon\}$
$T' \to *FT' / \epsilon$
include in FIRST(T')

$FIRST(F) = \{(, id\}$
$F \to (E) / id$

Compute first

$S \to ACB \mid CbB \mid Ba$

$A \to da \mid BC$

$B \to g \mid \epsilon$

$C \to h \mid \epsilon$

Ans: $FIRST(C) = \{h, \epsilon\}$

$FIRST(B) = \{g, \epsilon\}$

$FIRST(A) = \{d, g, \epsilon, h\}$

$FIRST(S) = \{d, g, \epsilon, h, a, b, \}$

Follow rules :

Given a non-terminal A the set Follow(A) consisting of terminals. Apply the following rules until nothing can be added to any follow set

1. If A is start symbol, then \$ is in follow(A)
2. If there is any production $B \to aAy$ then $First(y) - \{\epsilon\}$ is in follow(A)
3. If there is a production $B \to aAy$ or $B \to aA$ such that a is in FIRST(y) then everything of Follow(B) is in Follow(A).

ex:1
$E \to TA$
$A \to +TA \mid \epsilon$
$T \to FC$
$C \to *FC \mid \epsilon$
$F \to (E) \mid id$

$First(E) = FIRST(T) = FIRST(F) = \{(, id\}$

$First(A) = (+, \epsilon)$ $\quad FIRST(C) = \{*, \epsilon\}$

$\therefore Follow(E) = Follow(A) = \{\$,)\}$

$Follow(T) = Follow(C) = \{\$,), +\}$

$Follow(F) = (+, *,), \$)$

moves made by predictive parser for above grammar (id+id*id) for above grammar

Stack	input symbol	output
\$E	id+id*id\$	
\$AT	id+id*id\$	E→TA
\$ACF	id+id*id\$	T→FC
\$ACid	id+id*id\$	F→id
\$AC	+id*id\$	
\$A	+id*id\$	C→ε
\$AT+	+id*id\$	A→+TA
\$AT	id*id\$	
\$ACF	id*id\$	T→FC
\$ACid	id*id\$	F→id
\$AC	*id\$	[illegible]
\$ACF*	*id\$	C→*FC
\$ACF	id\$	
\$ACid	id\$	F→id
\$AC	\$	
\$A	\$	C→ε
\$	\$	A→ε

Algorithm: construct the predictive parsing table

Input:- Grammar G

Output: Predictive parsing table

Step(1): For each production $A \to \alpha$ do Step ② and step ③

Step ②. for each terminal 'a' in FIRST(α) add $A \to \alpha$ into $M[A, a]$.

Step ③ If ϵ is in FIRST(α) then add $A \to \alpha$ into $M[A, b]$ for each terminal 'b' in Follow(A).

If ϵ is in First(α) and \$ is in Follow(A) then add $A \to \alpha$ into $M[A, \$]$

Step ④: make undefined entry of M as error.

ex: for step-②

$S \to AB$
$A \to a | b$
$B \to b$

	a	b
S	$S \to AB$	$S \to AB$
A	$A \to a$	$A \to b$

$S \to AB$ $\quad A \to a$
$A \to \alpha$
FIRST(AB) = FIRST(aB) ∪ FIRST(bB)
= {a, b}

ex: for step-③-(i)

$S \to Ac$
$A \to \epsilon$
Follow(C) = {c}

	c
A	$A \to \epsilon$

ex: for step-③-(ii)

$S \to Ac / A$
$A \to \epsilon$

$A \to \epsilon$
$A \to \alpha$
First(ϵ) = {ϵ}
Follow(A) = {c}

Constructing predictive parsing table for given grammar

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

Sol: $E \to TE'$: $(A \to \alpha)$ form

FIRST(α) = FIRST(TE')
= FIRST(FT'E')
= FIRST((E)T'E')
FIRST(idT'E')
= {(, id}

M[E, (] M[E, id] → In these entries we need to place $E \to TE'$ production

$E' \to +TE'/\epsilon$: $(A \to \alpha)$

FIRST(+TE') = {+}
FIRST(ϵ) = {ϵ}; Follow(E') =
($E' \to +TE'$) is added in M[E', +]
($E' \to \epsilon$) is added in M[E',)]
M[E', \$]

$\underline{T \to FT'}$ $(A \to \alpha)$

$FIRST(FT') = FIRST((E)T') \cup FIRST(id\,T')$
$= \{(, id\}$

$(T \to FT')$ Production is added in
$M[T, (]$, $M[T, id]$

$\underline{T' \to *FT' / \epsilon}$ $(A \to \alpha)$

$FIRST(*FT') = \{*\}$

$FIRST(\epsilon) = \{\epsilon\}$

$Follow(T') = \{+,), \$\}$
$\Downarrow$
$follow(T)$
as ϵ is there after T'

$T' \to *FT' / \epsilon$ production is added
where $M[T', +]$, $M[T',)]$,
$M[T', \$]$

$\underline{F \to (E)}$

$FIRST((E)) \to \{(\}$

$\underline{F \to id}$

$FIRST(id) \to \{id\}$

	id	+	*	(	)	$
E	E→TE'			E→TE'		
E'		E'→+TE'			E'→ [illegible]	E'→ [illegible]
T	T→FT'			T→FT'		
T'		T'→ε	T'→*FT'		T'→ε	T'→ε
F	F→id			F→(E)		

The initial configuration of the predictive Parser:

Stack	I/P Buffer
$ S	w$

The acceptance configuration

Stack	I/P buffer
$	$

Stack	I/P buffer	O/P
$E	id+id*id$	
$E'T	id+id*id$	E→TE'
$E'T'F	id+id*id$	T→FT'
Pop() $E'T'	+id*id*id$	
$E'	+id*id$	T'→ε
$E'T+	+id*id$	
Pop()	id*id$	-
$E'T		
$E'T'F	id*id$	T→FT'
$E'T'id	id*id$	F→id
Pop() $E'T'F*	*id$	T'→*FT'
Pop()	id$	
$E'T'F		
$E'T'id	id$	F→id
Pop		
$E'T'	$	
$E'	$	T'→ε
$	$	E'→ε
$	$	(Accept)

$S \to iCtSS'$.

$FIRST(iCtSS') = \{i\}$

This production is added to $M[S, i]$

$S \to a$

$FIRST(a) = \{a\}$

This production is added to $M[S, a]$

$S' \to \epsilon$

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

~~This production~~ i:

$Follow(S) = FIRST(S')$
$= \{\epsilon, e\}$

ϵ is ignored

$S' \to eS / \epsilon$

$Follow(S) = Follow(S')$.

$M[S', \$]$ and $M[S', e]$ we add the production $S' \to \epsilon$

$M[S', e]$ we add production $S' \to eS$

	i	a	e	b	t	\$
S	$S \to iCtSS'$	$S \to a$				
S'			$S' \to \epsilon$ $S' \to eS$			$S' \to \epsilon$
C				$C \to b$		

Follow () rules

1. $Follow(S') = Follow(S) = \{\$, \cancel{\epsilon}\}$
2. $Follow(S) = FIRST(S') = \{\cancel{\epsilon}, e\}$
3. $Follow(S) = FIRST(S')$

LL(1) Grammar :

* A grammar for which parsing table with no multipled defined entries is called LL(1) grammar

Following are the properties to check the given grammar LL(1) or not.

- $LL(K)$ → no. of lookahead symbols to take parsing decisions

L → left to right scanning of input strings

L → left most derivation

Properties

1. No ambigious, left recursion, left factoring is LL(1) grammar
2. A parsing table with no multiple definition entries is LL(1) grammar
3. For each production $A \to \alpha / \beta$ should satisfy following conditions.

(i) $\boxed{First(\alpha) \cap FIRST(\beta) \neq \{a\}}$

ex:

$S \to xy / xw$

$S \to xy$

$FIRST(xy) = \{x\}$

$M[S, x]$

$S \to xw$

$FIRST(xw) = \{x\}$

$M[S, x]$

(ii) $\boxed{FIRST(\alpha) \cap FIRST(\beta) \neq \{\epsilon\}}$

ex: $S \to A/B$

$A \to \epsilon$

$B \to \epsilon$

$S \to A$

$FIRST(A) = \{\epsilon\}$

$Follow(S) = \{\$\}$

$S \to B$

$FIRST(B) = \{\epsilon\}$

$Follow(S) = \{\$\}$

atmost one production can derive ϵ

(iii) $\boxed{FIRST(\alpha) \cap Follow(A) = \phi}$

ex:-

$S' \to eS / \epsilon$

$FIRST(eS) = \{e\}$

$S' \to \epsilon$

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

$FIRST(\alpha) \cap FOllow(S')$

$\{e\} \cap \{\$, e\}$

$\{e\}$

(iv) $\boxed{Follow(A) \cap FIRST(\beta) = \phi}$

$S' \to \epsilon / eS$

$FIRST(eS)$

- Check following grammar is LL(1) grammar

$S \to AaAb \mid BbBa$

$A \to \epsilon$

$B \to \epsilon$

Sol:- $S \to AaAb \mid BbBa$

$FIRST(AaAb) = FIRST(A)$
$= \{a\}$

$FIRST(BbBa) = FIRST(B)$
$= \{b\}$

$FIRST(\alpha) \cap FIRST(\beta) \neq \{a\}$

$FIRST(\alpha) \cap FIRST(\beta) = \{\phi\}$

∴ given grammar is LL(1) grammar

• Construct predictive parsing table for

$S \to aAbB \mid bAbB \mid \epsilon$

$A \to S$

$B \to S$

Sol: ① $S \to aAbB \mid bAbB \mid \epsilon$

$S \to aAbB$:

FIRST(S) = FIRST(aAbB) = {a}   M[S, a]

$S \to bAbB$

FIRST(bAbB) = {b}   M[S, b]

$S \to \epsilon$

Follow(S) = {a, b, \$}

$A \to S$

Follow(S) = Follow(A)

$S \to aAbB$

Follow(A) = FIRST(bB)

= {b}

Follow(A) = FIRST(aB)

= {a}

$B \to S$

Follow(S) = Follow(B)

$S \to bAaB$

M[S, a], M[S, b], M[S, \$]

② $A \to S$

FIRST(A) = FIRST(S) = {a, b, \$}

= FIRST(aAbB) + FIRST(bAaB)   M[A, a]

+ FIRST(ε)   M[A, b]

= {a} + {b} + Follow(B)   M[A, \$]

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

③ $B \to S$

FIRST(B) = FIRST(S) = {a, b, \$}   M[B, a]

"   "   M[B, b]

"   "   M[B, \$]

	a	b	\$
S	$S \to aAbB$ $S \to \epsilon$	$S \to bAaB$ $S \to \epsilon$	$S \to \epsilon$
A	$A \to S$	$A \to S$	$A \to S$
B	$B \to S$	$B \to S$	$B \to S$

∴ This is not LL(1) grammar as "n" is possible (C1).

• construct LL(1) Parser table for given grammar & also show parsing action for input

$S \to aAcd \mid BcAe$

$A \to b \mid \epsilon$

$B \to Cf \mid d$

$C \to fe$

FIRST(S) = FIRST(aAcd) ∪ FIRST(BcAe)

= {a, d, f}

FIRST(A) = First(b) ∪ FIRST(ε) = {b, ε}

FIRST(B) = {d, f}

FIRST(C) = {f}

→ Follow(S) = {\$}

Follow(A) = {c, e}

Follow(B) = {c}

Follow(C) = {f}

Parsing table for LL(1) parsing table

Non-terminal	input symbol						
	a	b	c	d	e	f	\$
S	$S \to aAcd$			$S \to BcAe$		$S \to BcAe$	
A		$A \to b$	$A \to \epsilon$		$A \to \epsilon$		
B				$B \to d$		$B \to Cf$	
C						$C \to fe$	

Questions :

• Every SLR grammar is unambiguous but not every unambiguous grammar is SLR explain

• YACC calculator program with error production

UNIT – II

Bottom-up parsing :- Constructs the parse tree starting from the leaf node (i/p string) working up towards root node (starting non terminal).

ex:- $S \to aAcBe$
$A \to Ab \mid b$
$B \to d$

i/p string: a bbcde
↓
a Abcde
↓
a A cde
↓
a AcBe
↓
S

RMD:
$S \Rightarrow aAcBe$
$\Rightarrow aAcde$
$\Rightarrow aAbcde$
$\Rightarrow abbcde.$

Reduction: A substring matching with the right hand side of the production, replace it with the left hand side of production is called reduction.

Handle:- A substring matching with the right handside of production replacing such substrings with left hand side of production eventually leads to starting non terminal is called handle.

- find appropriate handles to reduce given i/p string 'id+id*id

$E \to E+E$
$E \to E*E$
$E \to (E)$
$E \to id$

Sol.

id+id*id ⇒ $id_1 + id_2 * id_3$
↓
$E + id_2 * id_3$
↓
$E + E * id_3$
↓
$E + E * E$
↓
$E + E$
↓
E : valid string

right sentential form	Handle	Production
$id_1 + id_2 * id_3$	id_1	$E \to id$
$E + id_2 * id_3$	id_2	$E \to id$
$E + E * id_3$	id_3	$E \to id$
$E + E * E$	$E*E$	$E \to E*E$
$E+E$	$E+E$	$E \to E+E$
E		

Stack	I/P Buffer	Action
\$	w\$	
\$	$id_1 + id_2 * id_3$\$	Shift
\$$id_1$	$+ id_2 * id_3$\$	Reduce $E \to id$
\$E	$+ id_2 * id_3$\$	Shift
\$E+	$id_2 * id_3$\$	shift
\$E+id_2	$* id_3$\$	Reduce $E \to id$
\$E+E	$* id_3$\$	Shift
\$E+E*	id_3\$	shift
\$E+E*id_3	\$	Reduce $E \to id$
\$E+E*E	\$	Reduce $E*E \to E$
\$E+E	\$	Reduce $E \to E+E$
\$ E	\$	Accept

Shift: Shift the next input symbol into stack

Reduce:- The right end of string to be reduced must be on top of stack. Locate the left end of string within stack and replace the string with non terminal

Accept. successful completion of parsing

Error: Discover a syntax error & call an error recovery routine

Conflicts during shift-reduce parsing

1. There are context free grammars for which shift-reduce grammar cannot be used.
   - We cannot decide whether to reduce shift or reduce eg: dangling if-else grammar

LR parsers :-

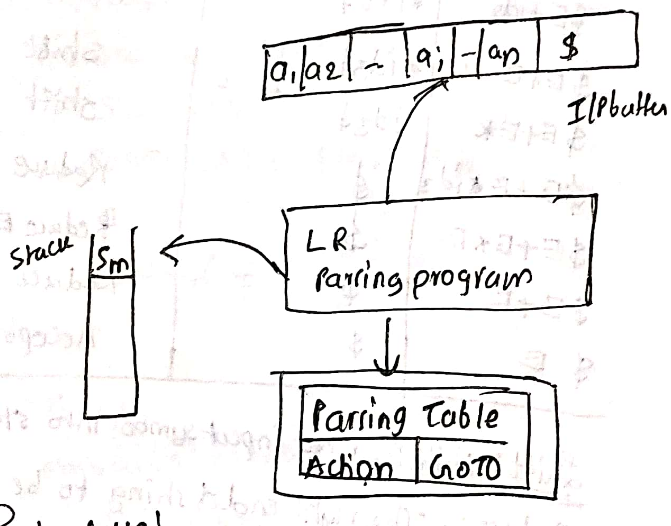
(1) SLR (Simple LR)
(2) CLR (Canonical LR)
(3) LALR (Look Ahead LR)

Stack:-

$S_0 X_1 S_1 X_2 S_2 \cdots X_m S_m$

$X_1, X_2 \cdots X_m \rightarrow$ Standard for Gramar symbol

$S_0 S_1 \cdots S_m \Rightarrow$ stands for State symbol



Parsing table

Terminal Action	Statement terminal
S,R,A,E	State symbol

Configuration of LR Parser:-

$(S_0 X_1 S_1 X_2 S_2 \cdots X_m S_m, a_i, a_{i+1} \cdots a_n \$)$

$ACTIONS[S_m, a_i] =$ shift s.

$(S_0 X_1 S_1 X_2 S_2 \cdots X_m S_m a_i s, a_{i+1}, a_{\cdots}$
$a_n \$)$.

$S = GOTO[S_m, a_i]$.

1. Shift Action
2. reduce Action
3. Accept Action:
   It halts the parser and announces successful parsing of input string.
4. $Action[S_m, a_i] =$ error:
   It identifies the error and calls the error recovery routine ().

Check the following ~~grammar~~ String

id+id*id Valid or not for the following given grammar & Parsing table

(i) $E \rightarrow E+T$
(ii) $E \rightarrow T$
(iii) $T \rightarrow T*F$
(iv) $T \rightarrow F$
(v) $F \rightarrow (E)$
(vi) $F \rightarrow id$

- For SLR, CLR, LLR, we dont eliminate ambiguity, left recursion & left factoring

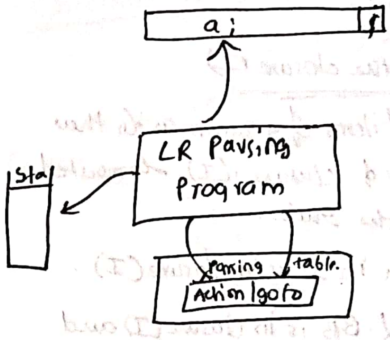
Parsing table

	id	+	*	(	)	$	E	T	F
0	s5			s4			1	2	3
1		s6				acc			
2		r2	s4		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			

*empty cells are errors

Stack	I/P buffer	Action
0	id+id*id$	shift
0 id 5	+id*id$	reduce F→id
0F3	+id*id$	reduce T→F
0T2	+id*id$	reduce E→T
0E1	+id*id$	shift
0E1+6	id*id$	shift
0E1+6id5	*id$	reduce F→id
0E1+6F	*id$	reduce T→f
0E1+6T9	*id$	shift
0E1+6T9*	id1	shift
0E1+6T9*7id5 $		

⇒



Prerequisites for LR parser

(i) Item: Placing the dot in right hand side of the production in any position
- each item describes how much production we have seen at a given point in the parsing process.

$A \to XYZ$

$A \to \cdot XYZ$ — expecting a string derived from X

$A \to X \cdot YZ$ — expecting a string derived from Y

$A \to XY \cdot Z$ — expecting a string derived from Z

$A \to XYZ \cdot$

eg:

$S \to Aa$

$A \to b$

∴ $S \to \cdot Aa$

$S \to A \cdot a$

$S \to Aa \cdot$

(ii) Augmented grammar: (G')

Adding the production $S' \to S$ to the actual productions

$S' \to S$
$S \to Aa$
$A \to b$

Purpose of adding this production is to halt the parsing process & announce the successful parsing of the input string

$F \to \cdot id$

(iii) closure (I):

Steps to compute the closure (I)

- If I is a set of items of grammar G then the set of items of closure (I) computed from I by the rules.
  1. every item in I is in closure(I).
  2. If $A \to \alpha \cdot B\beta$ is in closure (I) and $B \to \gamma$ is a production in Grammar G then add $B \to \cdot\gamma$ to I if it is not already in I.

eg: $S \to Aa$
$A \to b$

I: $S \to \cdot Aa$
$A \to \cdot b$

$\boxed{\text{closure}(I)}$

- closure ($S \to \cdot Aa$)
- $A \to \cdot b$

Compute closure of
$E' \to \cdot E$
$E \to E+T$
$E \to T$
$T \to T*F$
$T \to F$
$F \to (E)$
$F \to id$

Sol: $E' \to \cdot E$

I: $E' \to \cdot E$
$E \to \cdot E+T$
$F \to \cdot T$

$\boxed{\text{Closure}(I)}$

- closure ($E' \to \cdot E$).
$E \to \cdot E+T$
$E \to \cdot T$
$T \to \cdot T*F$
$T \to \cdot F$

Procedure closure(I)
begin
repeat
for each item $A \to \alpha \cdot B\beta$ is in closure and
each production $B \to \gamma$ is in grammar such that $B \to \cdot\gamma$ is not in I
do add $B \to \cdot\gamma$ to I
until no more set of items added to I
return I
end.

$\boxed{\text{goto}(I, x)}$ is defined as closure of all set of items of $A \to \alpha X \cdot \beta$ such that $A \to \alpha \cdot X\beta$ is in I.

I: $A \to \alpha \cdot X\beta$

goto (I, x) → dot is moved to right of x

∴ I: $A \to \alpha \cdot X\beta$

↓

closure($A \to \alpha X \cdot \beta$)

eg: I: $E' \to \cdot E$

closure($E' \to \cdot E$)
$E \to \cdot E+T$
$E \to \cdot T$
$T \to \cdot T*F$
$T \to \cdot F$
$F \to \cdot(E)$
$F \to \cdot id$

closure (goto (I, E))

$E' \to E \cdot$

$E \to E \cdot + T$

closure (goto (I, T))

$E \to T \cdot$

$T \to T \cdot * F$

closure((goto (I, ( )))

$F \to (\cdot E)$

$E \to \cdot E + T$

$E \to \cdot T$

$T \to \cdot T * F$

$T \to \cdot F$

$F \to \cdot (E)$

$F \to \cdot id$ —halt—

Sets of Item Construction:

Procedure ITEM(G')

begin

C = {closure ({S' → •S})}

repeat

for each sets of items in C &
each grammar symbol X
such that (I, X) is not empty &
goto (I, X) is not C.

Do add goto (I, X) to C

until no more sets of items added to C

end

Construct the sets of LR(0)

Items:

- LR(0) item → $A \to \alpha \cdot X\beta$ → SLR
- LR(1) item → $[A \to \alpha \cdot X\beta, a]$ ← CLR, LALR

eg:
$E \to E + T$ $S' \to S$
$E \to T$ $S \to a$
$T \to T * F$
$T \to F$
$F \to (E)$
$F \to id$

Sol: Augmented grammar is (G')

$E' \to E$.

$E \to E + T$ - - - same productions

Initial Itemset I_0: Closure ($E' \to \cdot E$)

$E \to \cdot E + T$

$E \to \cdot T$

$T \to \cdot T * F$

$T \to \cdot F$

$F \to \cdot (E)$

$F \to \cdot id$

∴ C = {$I_0, I_1, I_2, I_3, I_4, I_5, I_6, I_7, I_8, I_9, I_{10}, I_{11}$}

I_1: closure(goto (I_0, E))

$E' \to E \cdot$

$E \to E \cdot + T$

I_2: Closure(goto (I_0, T))

$E \to T \cdot$

$T \to T \cdot * F$

I_3 : closure (goto (I_0, F))

$T \to F \cdot$

I_4 : closure (goto (I_0, ( ))

$F \to (\cdot E)$

$E \to \cdot E + T$

$E \to \cdot T$

$T \to \cdot T * F$

$T \to \cdot F$

$F \to \cdot (E)$

$F \to \cdot id$ — halt —

I_5 : closure (goto (I_0, id))

$F \to id \cdot$

I_6 : closure (goto (I_1, +))

$E \to E + \cdot T$

$T \to \cdot T * F$

$T \to \cdot F$

$F \to \cdot (E)$

$F \to \cdot id$

I_7 : closure (goto (I_2, *))

$T \to T * \cdot F$

$F \to \cdot (E)$

$F \to \cdot id$

I_8 : closure (goto (I_4, E))

$F \to (E \cdot)$

$E \to E \cdot + T$

~~I_9 : closure (goto (I_4, T))~~

~~$E \to T \cdot$~~

~~$T \to T \cdot * F$~~ $= I_2$

I_9 : ~~closure (goto (I_4, F))~~

~~$F \to id \cdot$~~

$= I_3$

I_9 : ~~closure (goto (I_4, ( ))~~

$= I_4$

~~I_9 : closure (goto (I_4, id))~~

$= I_5$

I_9 : closure (goto (I_6 , T))

$E \to E + T \cdot$

$T \to T \cdot * F$

~~I_{10} closure (goto (I_6, F))~~

$\Rightarrow I_3$

~~I_{10} closure (goto (I_6, ( ))~~

$\Rightarrow I_4$

I_{10} ~~closure (goto~~ $(I_6, id))$

$\Rightarrow I_5$

I_{10} closure $(goto(I_7, F))$

$T \to T * F \cdot$

I_{11} closure ~~(got~~ $(I_7, ())$

$\Rightarrow I_4$

I_{11} ~~closure (goto~~ $(I_7, id))$

$\Rightarrow \cancel{I_5}$

I_{11} closure $\cdot(goto(I_8,)))$

$F \to (E) \cdot$

I_{12} closure $(goto(I_8, +))$

$\Rightarrow I_6$

UNIT – III

⇒ Syntax directed translation :-

↓ stream of token

$\boxed{\text{Syntax Analyzer}}$

↓

Syntax tree

A syntax directed definition (SDD) is a Context-free grammar together with attributes & rules

↙

Attributes associated with grammar symbols & rules associated with productions.

$$\left.\begin{array}{l} E \to E+E \\ E \to E*E \\ E \to (E) \\ E \to id \end{array}\right\} \text{Ambigious grammar}$$

∴ unambiguous grammar is

$E \to E+T / T$

$T \to T*F / F$

$F \to (E) / id$

, D is used to declare the datatype of variable

Datatype ⟶ list of variables

L → list of variables

$$\boxed{\begin{array}{l} D \to TL; \\ T \to int / char \\ L \to L, id / id \end{array}}$$

grammar for declaring variables of particular datatype

↙

This grammar donot explains how the expression is evaluated

↙

so we need to add some additional information in form of symantic rules.

↓

we have 2 methods to associate symantic rules to the Content free grammar

(1) Syntax directed definition →

(2) Translation schema

CFG + semantic rule

evaluation of arithmetic expression :
(using syntax directed definition)

$E \to E+T$

$E \to T$

$T \to T*F$

$T \to F$

$F \to (E)$

$F \to digit.$

- we associate the attributes with the terminals & non-terminals using the (·) operator
- attributes are used to hold some information. So we associate ·val with non terminals for value.
- And attribute needs to be associated based on the type.
  eg: ·type (datatype)
  ·string (string)
- If terminal is predefined we donot associate any attribute
  eg: (+, -, *, /, (,) ... etc)

- $\underline{E \to E+T}$

  $E.val = E_1.val + T.val$

- $\underline{E \to T}$

  $E.val = T.val$

- $\underline{T \to T_1 * F}$

  $T.val = T_1.val * F.val$

- $\underline{T \to F}$

  $T.val = F.val$

- $\underline{F \to (E)}$

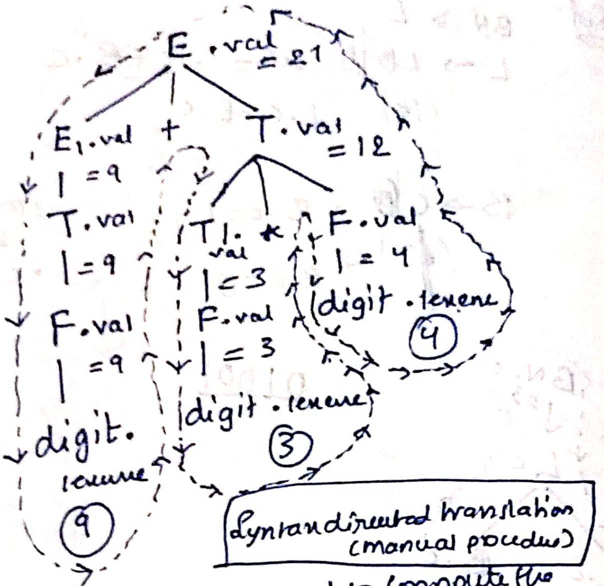
  $F.val = E.val$

- $\underline{F \to digit}$

  F.val = digit.lexeme

9+3*4

digit + digit * digit



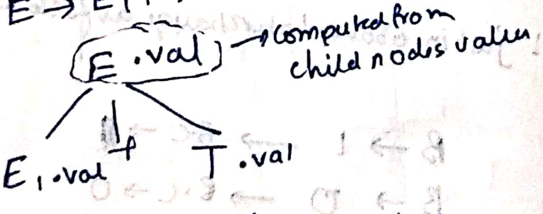
* At each node we need to compute the attribute values
* we are going to use postorder traversal

left → right → root

→ There are two types of attributes

① synthesized attribute :- In a parse tree attribute value at a node computed from attribute values of child nodes

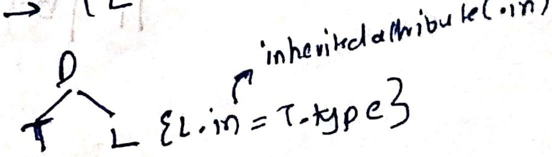
Eg :- $E \to E_1 + T$ (semantic rule)



② Inherited attribute : In a parse tree at a node attribute value computed from attribute value of parent / or siblings

(nodes at same level called sibling)

$D \to TL$



S-attributed definition :

* A syntax directed definition synthesized attributes (exclusively) is called S-attributed definition.

$x + \underbrace{y * 3}_{T_1}$

$x + T_1$

$\boxed{T_2 = x + T_1}$ ✓

Different types of 3-address code statements :-

① Assignment ~~operation~~ Statement : A = B op C

② Assignment Instruction : A = op B

unary minus

~ (negation)

Type cast

int float

a = 6

a = (int) 6

③ Copy statement :

A = B

④ unconditional Jump statement :-

goto label ; / Jmp

★A

⑤ Conditional Jump statement

If A (relational operator) B.

goto L

⑥ Procedure call statement :- ↗ Procedure name

param x call (P, n) → no. of parameters Passing in call statement

↳ each parameter is represented by param eg: Sum(x, y, 3)

This procedure call is converted into the three address code

Param x
param y
param 3
call sum, 3

sum(x, y, 3) is converted as like this

(7) Indexed Assignment statement

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

(8) Address Assignment pointer Assignment

x = &y;

x = *y

*x = y

Record structure: To store the Three address code

(1) Quadraple record structure

op	arg1	arg2	result

eg:-1 x = y + 3

+	y	3	x

eg 2:-2

unary minus	y	~~z~~	x

eg: 3 x = y (If no op then we take '=' operator)

:= (assign)	y	~~X~~	x

(2) Triple record structure

(3) Indirect Triple record structure

Disadvantage structure:

: Representing the following in Quadrapule, ~~Triple, Indirect triple~~

~~a = b * -c + b * -c~~

~~a = T1 + T2~~

(1) $a = b * -c + b * -c$

$T_1 = -c$

$T_2 = b * T_1$

$T_3 = -c$

$T_4 = b * T_3$

$T_5 = T_2 + T_4$

$a = T_5$

one record → and memory is assigned to each record

r#	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	assign (:=)	T_5	--	a

In actual implementation the pointer to symbol table is written instead of names.

disadvantage of Quadrapule Record structure :

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

↓

lexical analyze

each identifier stored in symbol table (while translating into three address code)

In the quadrapule format the symbols are pointers to symbol tables & some are not (temporary variables eg: T_1, T_2)

And these temporary variables also need to store inside the symbol table

So it occupies huge memory

To avoid this we use Triple record structure

Instead of storing in temporary variable & also memory location we use Triple Rs

It occupies more memory because of storing temporary variables inside symbol table

Triple Record structure for $a = b * -c + b * -c$

→ mem loc

r	op	Arg1	Arg2
(0)	unary m	c	--
(1)	*	b	(0)
(2)	unary m	c	--
(3)	*	b	~~T3~~ (2)
(4)	+	(1)	(3)
(5)	assign	a	(4)

Disadvantage of Triple R.s during compilation

- During runtime the assigned memory locations for records may not be free
- So we face runtime overhead to change the memory location during runtime & otherwise we get undesired results

In order to overcome this issue we go for the Indirect triple record structure.

It is a list of pointers to the triple record structure

Indirect Triple Record Structure

Ptr	ptr (list execut)	Statement
(19)	(13)	(0)
(20)	(14)	(1)
(21)	(15)	(2)
(22)	(16)	(3)
(23)	(17)	(4)
(24)	(18)	(5)

Address stored inside → Pointer will not change.

The pointer address may change but the address pointed by pointer will not change

disadvantage of Indirect triple record structure

- It includes one more indirection level which is a hectic.

UNIT – IV

Typechecking

- 4 types of errors are identified by the semantic analyzer
  1. Type checks → for an operator compatible operands are applied or not
  2. Flow control checks
  3. uniqueness checks
  4. Name related checks

In order to leave a language construct we need to have a flow of control

eg: Switch construct

If in case break; is not used no error is shown, all cases are executed --
but within C#, .NET it shows an error

In order to leave switch construct we need a flow of control to exit out of switch & transfer control to another statement

eg: B1
begin
B2
begin
end B2
end B1

So each block should be ended by its name

even the nested blocks are supported by the Ada language

A variable can be declared of one datatype, but not more than one datatype

eg: int x; ✗
float x;

ambiguous

eg: In switch each case (') should be unique

within the Ada programming language we use name for procedure

Procedure x
begin
end x;

checked whether These are defined or not.

⇒ In order to perform type checking functionality a tool is used called Typechecker.

- It computes the datatype of language construct (It is a syntactical statement which is formed by using one or more tokens) and verifies meaning of those language constructs according to the language rules.

eg: a % b. is a language construct.
(float)

↓

but within the 'c' language we cannot perform the modulus operation on the float variables.

↓

but this is valid within the java programming language.

- Type system is the main functionality that is implemented inside the type checker

→ Type system keeps track of datatype info of language constructs and also assigns datatype to language construct. Also verifies whether it is according to the language rules or not.

↓

In order to assign datatype to the language construct Type system uses an expression called type expression (depends on the language)

There are two types of type expressions
- basic type expression
- type construct type expression.

↓

within the basic type expressions
1. – boolean
   – character
   – int
   – real
2. type_error: used to signal an error within the programming statement.
3. void: In absence of the datatype.

↓

within the type construct type expression

"Type construction is formed by applying an operator to the other type expression"

1. Array (within pascal)

Var A : array[1..50] of int

⇒ [In c]

int A[50];

Type expression for array variables

array(1..50, int)

2. Cartesian product

$T_1 \times T_2$

-- Consider the formal parameters passed within the function definition

function (int a, float b, char c)
{
}

These type expression is

int × float × char

③ within the c programming language we use the structure

```
Struct emp
{
    char A[10];
    int age;
};
```

so we use a type expression for structure

record((A × array (0..10, char)) × (age × int))

This type expression uses cartesian product alongwith datatype and also field.

④ within the c programming language we use the pointers

int *p;

Then the type expression for the pointer is

Pointer(int) | Pointer(T);

int *P[10];

Then the type expression is

array (1..10, Pointer(int))

⑤ functions

we use mathematical notation to represent the type expression for the functions

D → R (domain mapped to range)

eg:- float sum(int a, int b)

{

}

The type expression is

int × int → float.

write type expressions for following types

① An array of pointers to real, where the array index ranges from 1-100.

② A two dimensional array of integers (an array of arrays) whose rows are indexed from 0-9 and whose columns are indexed from -10 to 10

③ functions whose domains are functions from integers to pointers to integers and whose ranges are records consisting of an integer and character.

eg:-

$P \rightarrow D ; E$

declarations; expressions

This program only consists of declarations followed by expression

$D \rightarrow D ; D$

$D \rightarrow id : T$

$T \rightarrow char \mid integer \mid \uparrow T \mid array [num] \text{ of } T$

Pointers in pascal are represented using ↑ followed by variable in the pascal language.

$E \rightarrow literal \mid number \mid id \mid E \bmod E \mid E[E] \mid E\uparrow$

$P \rightarrow D ; E$ represents the program structure

- So we write the semantic rules for the above grammar

$D \rightarrow id : T$ addtype(id.entry, T.type)

id stored within the symbol table.

$T \rightarrow char$	{T.type = char}
$T \rightarrow integer$	{T.type = integer}
$T \rightarrow \uparrow T$	{T.type = pointer(T.type)}
$T \rightarrow array [num] \text{ of } T$	{T.type = array(1..num.val, ...}

$E \rightarrow char$ ---> literal

∴ $E \rightarrow literal$ {E.type = char}

$E \rightarrow num$ {E.type = integer}

$E \rightarrow id$ {E.type = lookup(identity)}

$E \rightarrow E_1 \bmod E_2$

{E.type = if E_1.type = integer and E_2.type = integer then integer else type-error}

$E \rightarrow E_1[E_2]$ {E.type := if E_2.type = integer and E_1.type = array(s,t) then t_1 else type-error}

Type conversion can be done in 2 ways:

(i) implicit type conversions

(ii) explicit type conversions

⇒ equivalance of type expressions

- Type expressions are built from basic types or formed by applying type constructors to other type expressions.

Two type expressions are said to be **structurally equivalent** if they are same basic types or formed by applying type constructors to other expressions.

They are also structurally equivalent if they are identical.

```
function: sequiv (s,t) : boolean
begin
    if s & t are same basic types then
        return true;
    else if s = array(s1,s2) & t = array(t1,t2)
        return sequiv(s2,t2)
    else if s → s1 x s2 & t = t1 x t2 then
        return sequiv(s1,t1) & sequiv(s2,t2)
    else if s = pointer(s1) & t = pointer(t1) then
        return seqiv(s,t).
    else if s = s1 → s2 & t = t1 → t2 then
        return sequiv(s1,t1) & sequiv(s2,t2)
    else
        return false
end
```

Two variables are said to be **name equivalent** if they are associated with same type expressions.

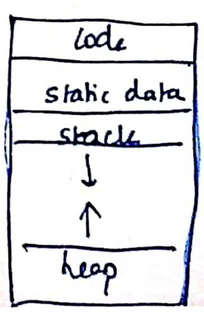
Runtime environment

- Compiler obtains a block of storage from OS to run the target code.

They contain

1. generated target code
2. data objects known at compile time
3. Stack used to control execution of procedures

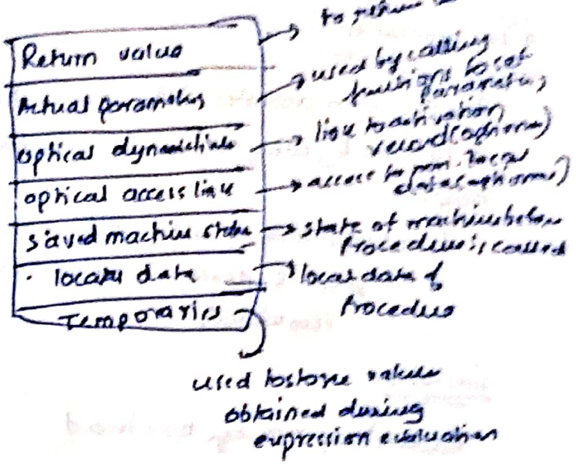
Run time memory is divided into



stack & heap size change dynamically

Activation record / frame :

Information needed for execution of a procedure is stored in a block of storage called as Activation record / frame.



Storage allocation strategies

① **Static Allocation**

- is done at compile time
- all data objects should be known at compile time
- no dynamic changes allowed

Limitations:

- recursive functions not allowed
- no dynamic data structures.

② **Stack Allocation**

manages runtime storage as a stack (control stack)

- Activation records are pushed & popped when procedure begins & ends respectively.
- local values are bound to fresh storage when activation record is pushed to stack
- & are deleted when activation record is popped

Allocation tree :

A tree which represents sequence of procedure calls is called Allocation tree.

eg:

```
main()
{ f1();
}
f1()
{
  f2()
  f(3()
}
f2()
{
}     f3(){ }
```

Limitations:

- local values are not retained
- memory addressing can be done using pointers

Heap allocation :-

This allocation allocates & deallocates memory at runtime by using the memory area called heap.

It allocates memory when activation record starts & deallocates when activation record ends

Limitations:

~~Frag~~ Heap manager overhead

Parameter passing: communication

3 ways of parameter passing

1. **Call by value :**

actual parameters are evaluated & values of actual parameters are passed to called procedure

Two steps

- Formal parameters are considered as local names to the called procedures so the storage of formals is in activation record of procedure
- caller evaluates actual parameters and their r-values are stored in the storage of formals.

```
eg:- main()            swap(int a, int b)
     {                 {  temp = x;
       swap(a,b)       }  x = y;
     }                    y = temp;
```

2. **Call by reference :** l-values of an expression are passed as actual parameter to the called procedure.

```
swap(&a,&b)        swap(int *a, int *b)
                   {
                     temp = *x;
                     *x = *y;
                     *y = temp;
                   }
```

3. **copy restore !**

- This is a hybrid mechanism of call by value & call by reference.
- Also called as copy-in & copy-out

1. Before the control transfers to the called procedure, actual parameter and their r-values are stored in the storage of the formal parameter.
2. when the control returns the current r-value of formal values of formal parameter are copied into the l-value of actual parameters.

```
program copy (input, output)
  var a:integer
  procedure display (x:integer)
    begin
      a:=2;
      a:=0;
    end
  a:=1
  deploy(a);
  writeln('a=',a);
end
```

Symbol table : is a datastructure which consists of records of each identifier with fields for the value of the identifier

1. variable names
2. constants
3. compiler generated temporaries
4. function names
5. scope information

symbol table identifier format

Name	Information

Loop optimization :

(i) Code motion (or) loop Invariant (or) Frequency reduction.

- reducing the number of instructions under loop or reducing the executing time of loops will speed up the program

Code motion :- an expression value that is not changing within the loop, then moving that expression to beginning before loop

consider

```
limit = 100
while (i <= limit - 2)
{ = =
  i++;
}
```

Also called loop invariant.

And also called as frequency reduction

This technique skips the evaluation of the loop

After applying Code motion

```
t = limit - 2
while (i <= t)
{ i++;
}
```

(ii) Induction variable elimination

Induction variable : A variable which have the following form

$$\boxed{I = I \pm C}$$

eg: consider $c = 1$ and $I = 1$

∴ I value increments linearly
i.e 1, 2, 3, 4, 5, 6, 7

eg:

```
(3) T1 = 4*i
     |
     |
    i = i+1
    If i <= 20 goto (3)
```

i	T_1
1	4
2	8
3	12
4	16

In here T_1 is also linearly incrementing within the loop.

" If two variables like above are linearly incrementing, so we replace these two variables by a single variable

∴ After applying Induction variable elimination

```
    T1 = 0;
(3) T1 = 4 + T1
     |
     :
    If T1 < 80 goto (3)
```

→ we eliminate inside the

" And the variables Can be in linear Increment or linear decrement in order to perform the Induction variable elimination ".

(iii) Strength reduction :-

- Consider within the loop there is an expression like

$a \uparrow 2 = a * a$

$a * 2 = a + a$"

If the induction variable elimination is Performed it also covers the Strength reduction optimization in loops

" so in place of performing multiplication, we replace with addition and we replace division operator by substraction"

" replacing a high level operator with low level Operator is called Strength reduction.

(iv) Loop unrolling :-

- "replicating the body of the loop in order to reduce required number of test conditions is called loop unrolling "

consider

```
I = 1;
while (I <= 100)
{
    a[I] = 0;
    I++;
}
```

" replacing a variable with another variable is called copy propagation

" upto h
go

" dec
un

eg

wh

(V) L

"upto how many lines the variable is going to be alive is live variable analysis"

" dead code elimination is elimination of unused variables and statements "

eg:
```
I = 1,3
while (I<=100)
{ a[I]=0;
  I++;
  a[I]=0;
  I++;
}
```

The no. of times the conditions executed is reduced to 50.

This only works when number of iterations are constant.

(V) loop jamming:

"merging the two bodies of the loop, if the number of test conditions, number of indices are same".

eg:
```
for (i=0; i<10; i++)
{ for(j=0; j<10; j++)
  {
     a[I,J]=0;
  }
}
for (i=0; i<10; i++)
{  a[i,i]=1;
}  //making diagonals = 1 //
```

After applying loop jamming

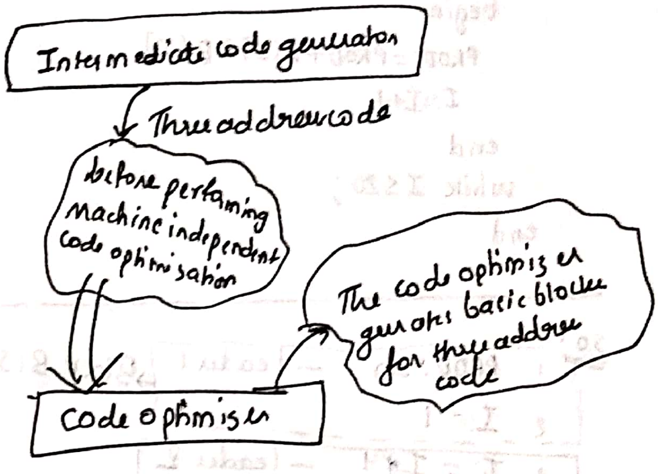
```
for(i=0; i<10; i++)
{ for (j=0; j<10 ;j++)
  { a[i,j] = 0;
    a[i,i]=1;
  }
}
```

Principle sources of optimisation ***

(or)

machine independent code optimization

Basic Blocks: "It is a sequence of consecutive statements that in "control enters at the beginning, once entered all the statements executed sequentially without halt (or) possible branching".



Algorithm Constructing basic blocks:-

Input: Three address code

Output: List of basic blocks with each tree address statement exactly in one block.

Procedure:

① we determine leaders, the first statements of basic blocks following are the rules

→ [The first statement is a leader]

→ [Any statement targeted by conditional or unconditional Jump statement is a leader]

→ [Any statement that follows a conditional jump statement is a leader]

② For each leader we construct a basic block which consist of first statement as leader and all statements upto but not including next leader.

→ [If there is no next leader, than it is the end of program]

Flowgraph: The relationship between basic blocks represented by a directed graph is called as a flow graph.

- Flowgraph consists of nodes and edges
- each node is a basic block
- There is a distinguished node called initial block whoose leader is a first statement of three address statements.

→ There is an edge from block B1 to block B2

(i) In Block B1 last statement is a Conditional jump statement, targeting to the first statement of block B2, then we place an edge from Block B1 to Block B2

(ii) Block B2 follows Block B1 in the flow of execution and then we place a edge

(iii) If there is an edge Block B1 to Block B2 B1 is a predecessor of B2 And the Block B2 is successor of Block B1