18/01/2022

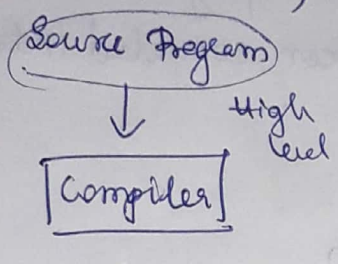
Unit-1

- Language Processors: -

Compiler and Interpreter

Compiler converts (whole program) and then execution (fast) the high level to low level language (which isn't direct)

Interpreter line by line execution of a source code. (slow)



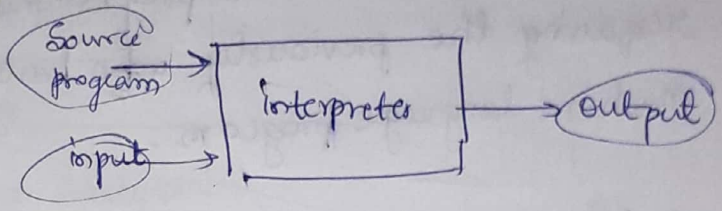


Fig Interpreter.

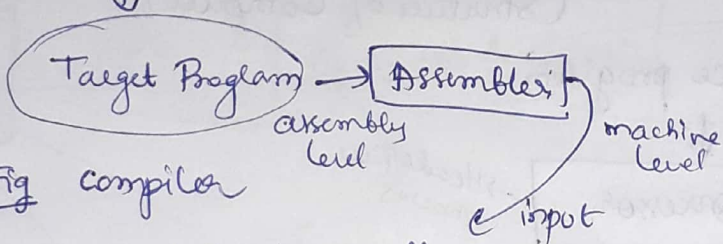


Fig compiler

That targt Prg is assembly level lang. → translated by assembler then o/p to machine level

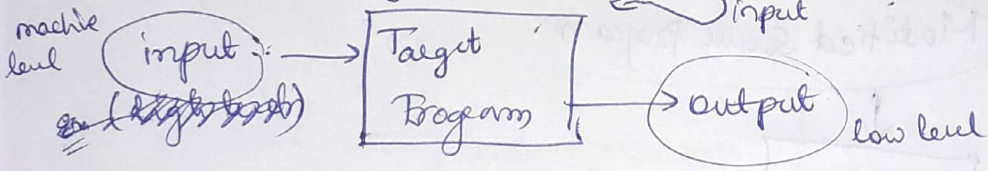


Fig Running the target Program.

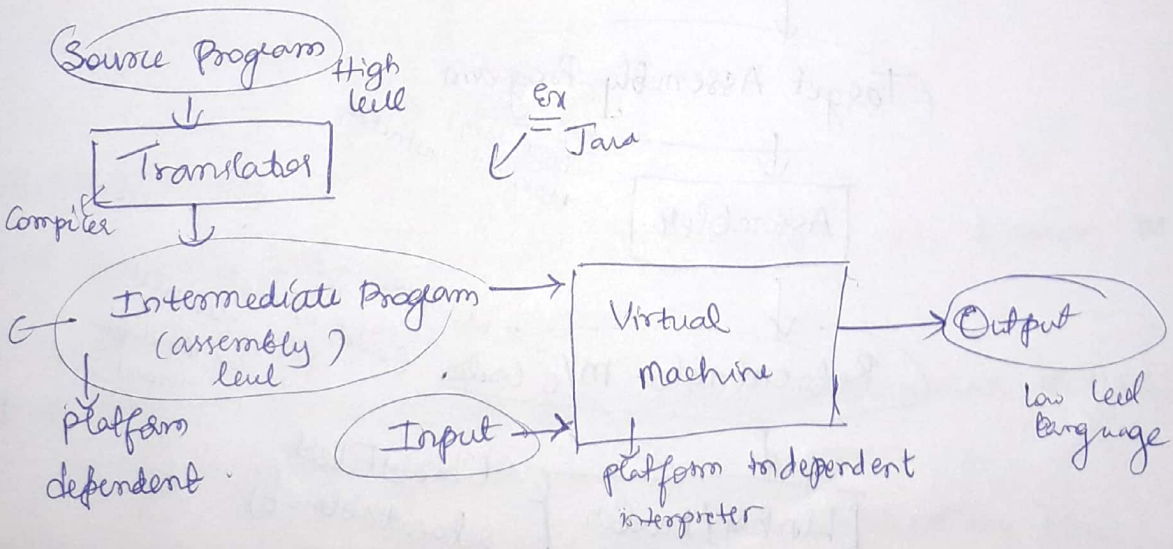


Fig A Hybrid compiler

→ Error - Mistake in coding then that is error

→ Bug - Tester finds the mistake, then that is bug

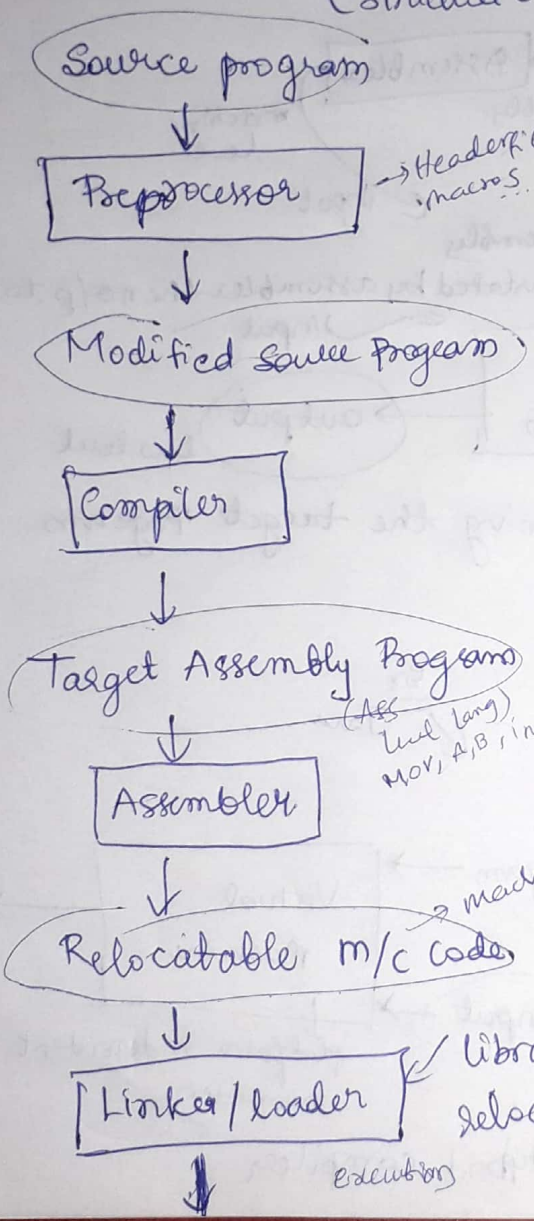
→ Defect - If these are accepted, then they are defects

→ Compiler - A compiler is a computer program that translates computer code written in one programming language into another programming language.

→ Interpreter - An interpreter is a prgeam that directly executes instructions written in a programming language without requiring the previously to have been compiled into a machine language program.

• A language Processing System

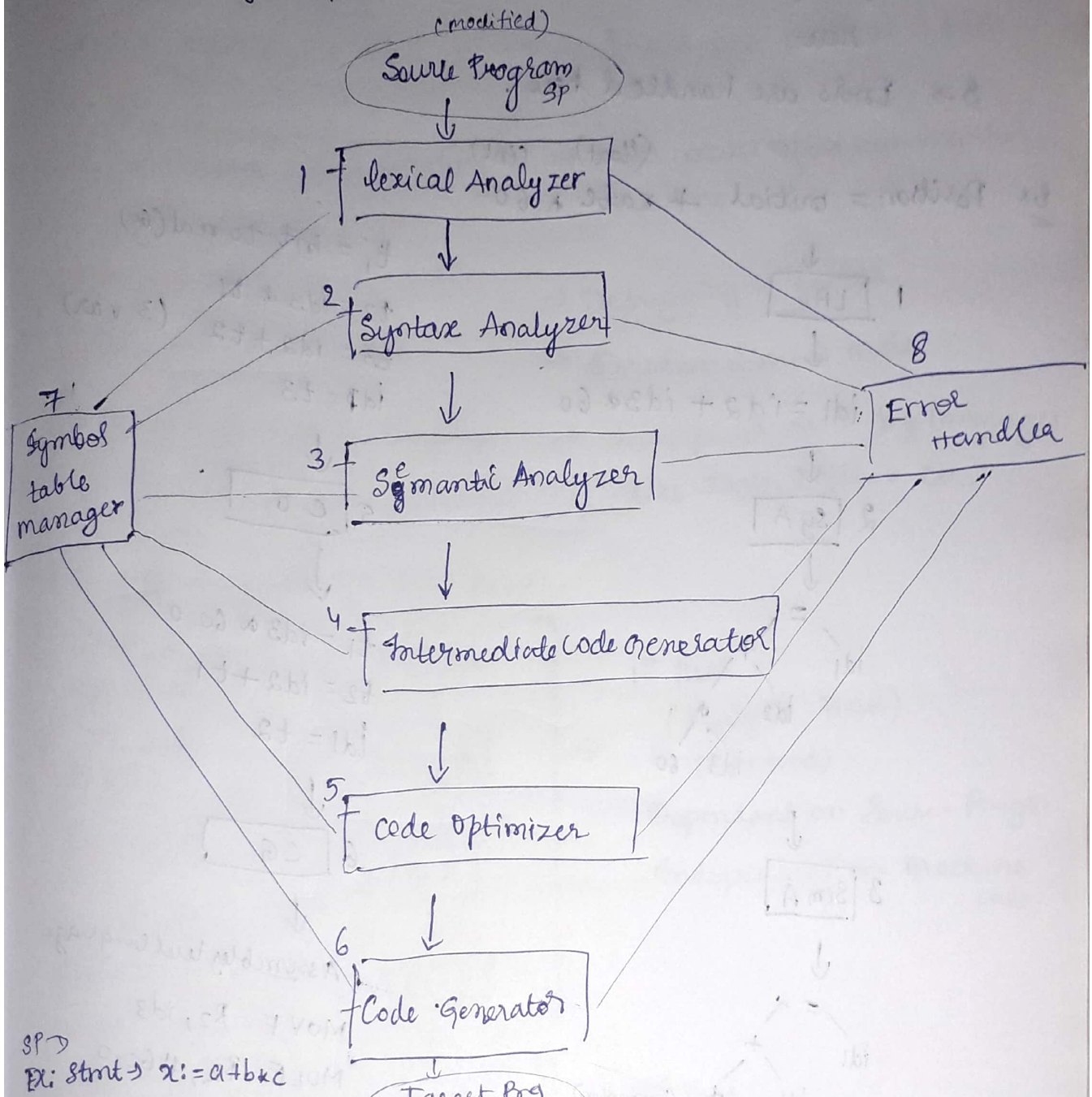
Skeleton (Structure of Compiler)



20/01/2022

Target m/c code
Machine level lang

- Phases of compiler



SP →
Ex: Stmt → x := a+b*c

1 → generates output has stream of tokens, Also known as scanner or token analyzer.

2 → whether the expression is according to grammar or not. Generates the syntax tree (Parse tree).

3 → Syntax tree meaningful or not is checked by this.

4 → generates a three address code. (using three variables solving. (Id - identifier)

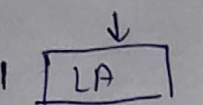
5 → generate the optimized code.

6 → Target program is generated

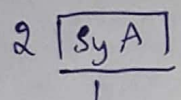
7 → If any variable, location, address that data is here.

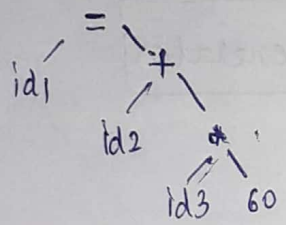
8 → Errors are handled here.

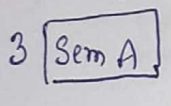
Ex Position = initial + rate * 60 (float) (int)

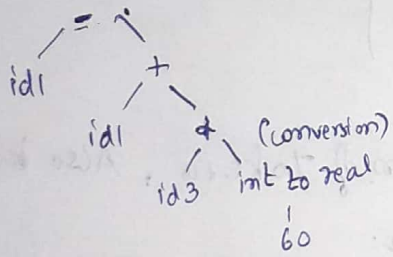


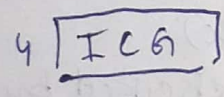
id1 = id2 + id3 * 60









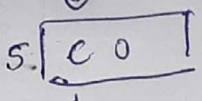


t_1 = int to real (60)

t_2 = id3 * t_1

t_3 = id2 + t_2 (3 var)

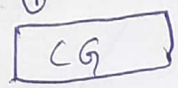
id1 = t_3



t_1 = id3 * 60.0

t_2 = id2 + t_1

id1 = t_2



Assembly level Language

MOV F R2, id3

MUL F R2, #60.0

~~ADD~~

MOV F R1, id2

ADD F R1, R2

MOV F id1, R1

↓

Float

- Difference b/w Compiler and Interpreter.

Compiler	Interpreter
→ Scans the whole program at a time and translate it into machine/target code	→ Scans the program line by line and translate into target code/program.
→ Shows all errors and warnings at a time.	→ Shows one error at a time.
→ Error occurs after scanning the whole program.	→ Error occurs after scanning the each line.
→ Debugging is slow.	→ Debugging is faster
→ Execution time is less.	→ Execution time is more
→ Compiler is used by language like C, C++ etc.	→ Interpreter used by languages like Java, Python etc.

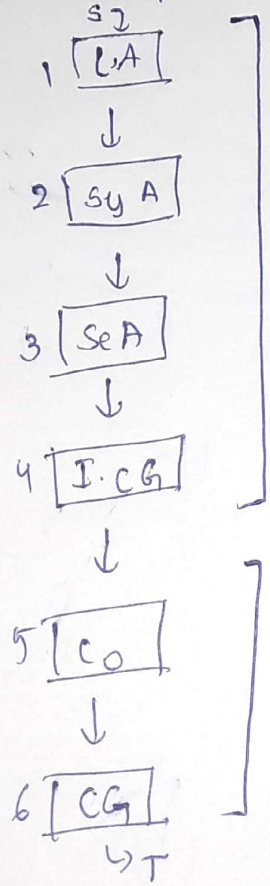
- Grouping of Phases into passes.

grouped of ~~by~~ phases to two passes.

Analysis and Synthesis phases grouping 4 each individual phases

Analysis → 1, 2, 3, 4.

Synthesis → 5, 6, 7, 8.



→ 1st Pass (Analysis Phase) (Frontend)
Dependent on Source Progr.
Independent on Machine code.

→ 2nd Pass (Synthesis Phase) (Backend)
Dependent on Machine Code
Independent of Source Progr.

∴ Source Prg → [Analysis Phase] → I.C.G. → [Synthesis Phase] → Target Prg

22/01/2022

As we already know the phases of a compiler from lexical analyzer to code generator.

- Symbol table Management
  - → For storing variable names, function names, classes etc.
  - → Used in Analysis and Synthesis
  - → Used to determine scope resolution.
  - → To give a name in a symbol table to check whether that name is existed or not.
  - → Access variables which is stored in symbol table.
  - → To give info to the already stored variables
  - → To delete a variable or graph of variables.

```
int var1;
int procA()
{
  int var2, var3;
  ---  --
  {
  int var4, var5;
  -------
  }
  int var6;
  ---  -
  {
    int var7, var8;
    ------
  }
}
```

```
int procB( )
{
  int var9, var10;
  - - - - - - - -
  {
    int var11, var12;
    - - - - - - - -
  }
  int var13;
}
```

- Symbol table (Global)

var1	var	int
ProcA	Proc	int
ProcB	Proc	int

Symbol table — ProcA

var2	var	int
var3	var	int
var6	var	int

Symbol table — ProcB

var9	var	int
var10	var	int
var13	var	int

var4	var	int
var5	var	int

inner scope 1

var7	var	int
var8	var	int

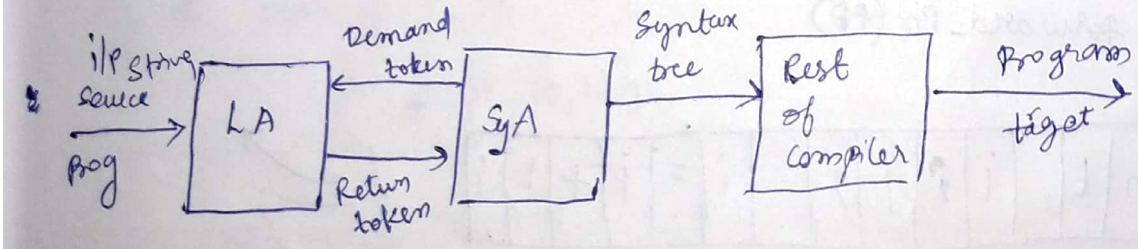
inner scope 2

var11	var	int
var12	var	int

inner scope 3

and need

- Role of lexical Analyzer



→ LA performs the following functions

i) Produces streams of tokens

ii) It eliminates blank spaces and comments

iii) It keeps track of lines no.s.

iv) It reports the errors encountered while generating the tokens.

Token	Pattern	Lexeme
class of category of i/p string. Ex- keyword, operator	set of rules that describe tokens	sequence of characters in source programe that are matched with the pattern of token.

Example

```
int max (int a, int b)
{
  if (a>b)
    return a;
  else
    return b;
}
```

Lexeme	Token
int	keyword
max	identifier
{	operator
int	keyword
⋮	⋮

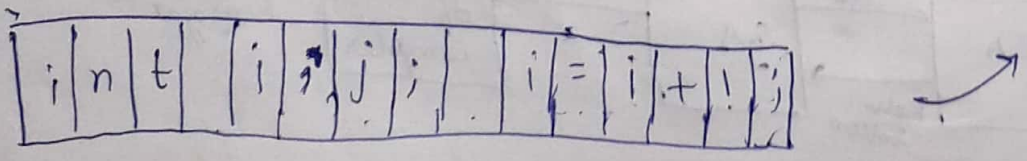
- Input Buffering

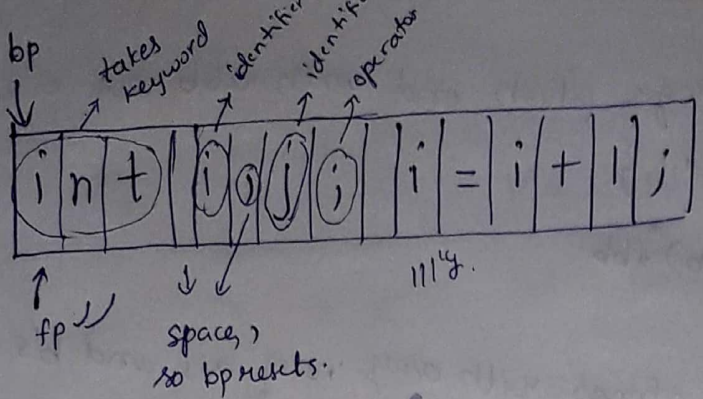
  ↓

  2 pointers

  i) begin_Ptr (bP)

  ii) forward_Ptr (fP)





→ Input buffering & methods -

i) One Buffer Scheme

ii) Two Buffer Scheme

→ One Buffer Scheme

Buffer 1 | i | n | t | | i | , | j |

Buffer 2 | ; | i | = | i | + | 1 | j | eof |

25/01/2021

• Regular Expression

→ R_1, R_2 are two Regular Expressions then

$R = R_1 + R_2$ where + represents Union operation

→ R_1, R_2 are two RE then

$R = R_1 R_2$ which represents concatenation

→ R_1, R_2 are two RE then

$R_1 = R_1^*$ which represents Kleene Closure

eg: Write a RE for language containing the strings of length 2 over $\Sigma = \{0, 1\}$

$L = \{00, 01, 10, 11\}$

$\therefore RE = (0+1)(0+1)$

Ex Containing the strings which end with abb one over

$\Sigma = \{a, b\}$.

$RE = (a+b)^* abb$

Ex Containing all the strings with any no of a's and b's.

$RE = (a+b)^*$

Ex containing all the strings starting with 1 and end with 0 over

$\Sigma = \{0, 1\}$.

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

Ex Consists of exactly 2 b's over $\Sigma = \{a, b\}$

$RE = a^* b a^* b a^*$

- Recognition of token

Token type	Token Value

```
if (a<10)
i = i+2
else
i = i-2
```

Token	Code	Value
if	1	—
else	2	—
while	3	—
for	4	—
identifier	5	ptr to symbol table
constant	6	ptr to symbol table
<	7	1
<=	7	2
>	7	3

			Location	Type	Value
>=	7	4			
!=	7	5	100	identifier	a
(	8	1	⋮		
)	8	2	105	constant	10
+	9	1	⋮		
-	9	2	107	identifier	i
=	10	-	⋮		
			110	constant	2

(code, value)

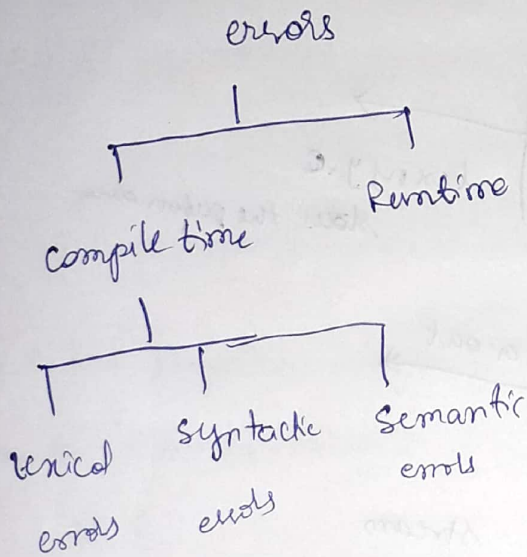
If(a<10) => 1, (8,1), (5,100), (7,1), (6,105), (8,2)

i=i+2 => (5,107), 10, (5,107); (9,1), (6,110)

else => 2

i=i-2 => (5,107), 10, (5,107), (9,2), (6,110)

- Lexical Phase Errors



- Lexical errors:

i) Spelling Mistakes

ii) Exceeding length of identifier or numeric constants.

iii) Appearance of illegal characters

i) Spelling mistakes

eg switch (choice)

{ — case1 : =
— case2 : =
}

The word 'sutich' spelling mistake can't be recognized.

In case1, there is no space, it takes it as an identifier.

iii)

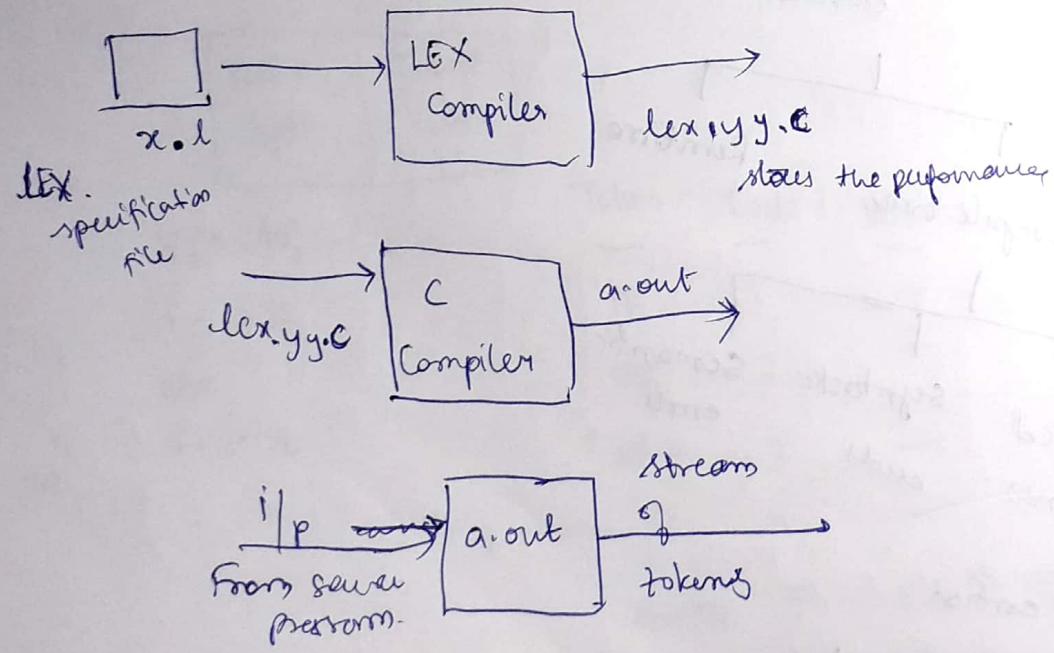
printf ("\nHello "); $ -> no use

ii) FOTRAN (only)

identifier to ✓ others identifier -8

- A language for specifying lexical Analyzer.

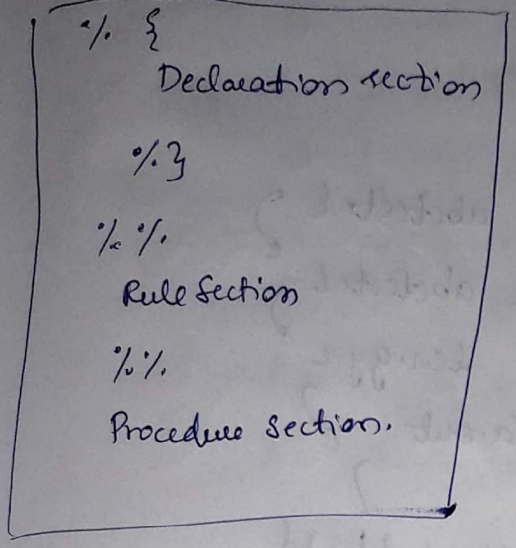
Tool used in LA is. LEX.



Generation of LA using LEX

22/01/2022

- 3 Sections -
  i) Declaration
  ii) Rule
  iii) Procedure.



- Sample Basic Program

```
%{
%}
%%
"Rama" ;
"Seetha" ;
"Geetha" ; printf("\n noun");
"Sings" ;
"dance" ;
"eats" ; printf("\n Verb");
%%
main()
{
yylex(); // Hi
}
int yywrap()
{
return 1;
}
```

this will be in lex.yy.c ←

- Write a lex program to convert the ~~today~~ substring abc to ABC from the given i/p string.

```
%{
#include<stdio.h>
#include<string.h>
int i;
%}
%%
[a-z A-Z]*{
for(i=0; i<=yyleng; i++)
```

```
{
if ((yytext[i] == 'a') && (yytext[i+1] == 'b') &&
                          (yytext[i+2] == 'c'))
{
 yytext[i] = 'A';
 yytext[i+1] = 'B';
 yytext [i+2] = 'C';
 }
}
printf ("%s", yytext);
}

%%
main()
  {
  yylex();
  }
  int yywrap()
    {
    return 1;
    }
```

```
$ vi abctest.l
$ lex abctest.l
$ cc lex.yy.c
$ ./a.out
```

i/p string: helloabcWorld

o/p: helloABCWorld

27/01/2022

Unit-2

- CFG - Context Free Grammar
  - is used to generate pattern of strings in a given finite language -
  - defined by 4 tuple

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

V - set of non-terminals
T = set of terminals
P - set of production rules
S - Start symbol

Ex Construct a CFG for the language having ~~only~~ any no. of a's over $\Sigma = \{a\}$

Sol $L = \{\epsilon, a, aa, aaa, \ldots\}$

$S \to aS$
$S \to \epsilon$ i/p → aaaaaa

$S \to aS$
$S \to aaS$ [∵ $S \to aS$]
$S \to aaaS$ [∵ $S \to aS$]
$S \to aaaaS$ [∵ $S \to aS$]
$S \to aaaaaS$ [∵ $S \to aS$]
$S \to aaaaaaS$ [$S \to aS$]
$S \to aaaaaa$ [∵ $S \to \epsilon$]

Ex Construct a CFG for the language $L = \{a^n b^{2n} / n \geq 1\}$

Sol $L = \{abb, aabbbb, aaabbbbbb, \ldots\}$

i/p aaabbbbbb

$S \rightarrow aSbb$
$S \rightarrow abb$

$S \rightarrow aSbb$
$S \rightarrow aaSbbbb$ [$\because S \rightarrow aSbb$]
$S \rightarrow aaaSbbbbbb$ [$\because S \rightarrow aSbb$]

- Derivation Tree and Parse Tree

↓

Sequence of Production Rules

→ 2 Types -

i) Left most Derivation (LMD)
ii) Right most Derivation (RMD)

Eg

i/p aaabbbbbb
$S \rightarrow aSbb$
$S \rightarrow aabb$

29/01/2022

Ex Derive the string aaabbabbba using CFG

$S \rightarrow aB/bA$
$A \rightarrow a/aS/bAA$
$B \rightarrow b/bS/aBB$

LMD

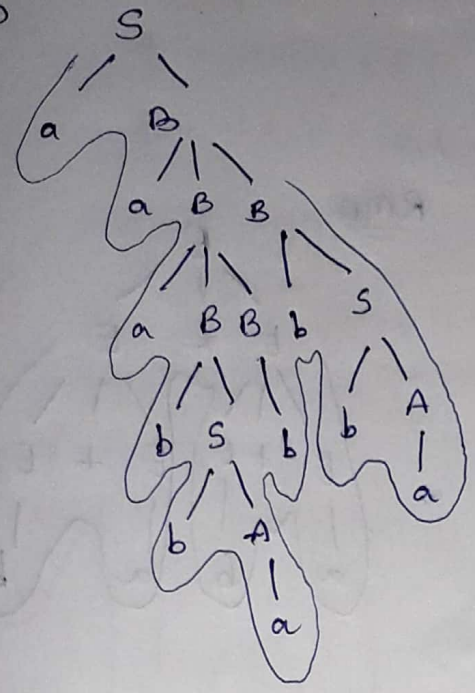
$S \rightarrow aB$
$S \rightarrow aBB$ ($\because B \rightarrow aBB$)
$S \rightarrow aaaBB$ ($\because B \rightarrow aBB$)
$S \rightarrow aaabSBB$ [$\because B \rightarrow bS$]
$S \rightarrow aaabbABB$ [$\because S \rightarrow bA$]
$S \rightarrow aaabbaBB$ [$\because A \rightarrow a$]
$S \rightarrow aaabbabB$ [$\because B \rightarrow b$]

$S \rightarrow aaabbabbS$ [$\because B \rightarrow bS$]

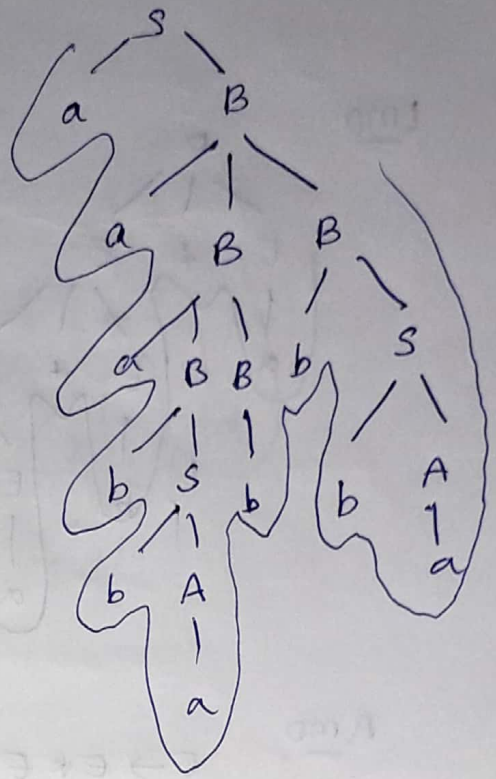
$S \rightarrow aaabbabbbA$ [$\because S \rightarrow bA$]

$S \rightarrow aaabbabbba$ [$\because A \rightarrow a$]

LMD



RMD -



RMD

$S \rightarrow aB$

$S \rightarrow aaBB$ [$\because B \rightarrow aBB$]

$S \rightarrow aaBbS$ [$\because B \rightarrow bS$]

$S \rightarrow aaBbbA$ [$\because S \rightarrow bA$]

$S \rightarrow aaBbbba$ [$\because A \rightarrow a$]

$S \rightarrow aaaBBbbba$ [$\because B \rightarrow aBB$]

$S \rightarrow aaaBbSbbba$ [$\because B \rightarrow bS$]

$S \rightarrow aaaBbbAbbba$ [$\because S \rightarrow bA$]

$S \rightarrow aaaBbbabbba$

$S \rightarrow aaaBbbbba$ [$\because B \rightarrow b$]

$S \rightarrow aaabSbbbba$ [$\because B \rightarrow bS$]

$S \rightarrow aaabbAbbbba$ [$\because S \rightarrow bA$]

$S \rightarrow aaabbabbbba$ [$\because A \rightarrow a$]

Ex $E \rightarrow E+E/E-E/E*E/E/E/a/b$

$a+b*a+b$

LMD $E \rightarrow E+E$

$E \rightarrow a+E$ [$\because E \rightarrow a$]

$E \rightarrow a+E*E$ [$\because E \rightarrow E*E$]

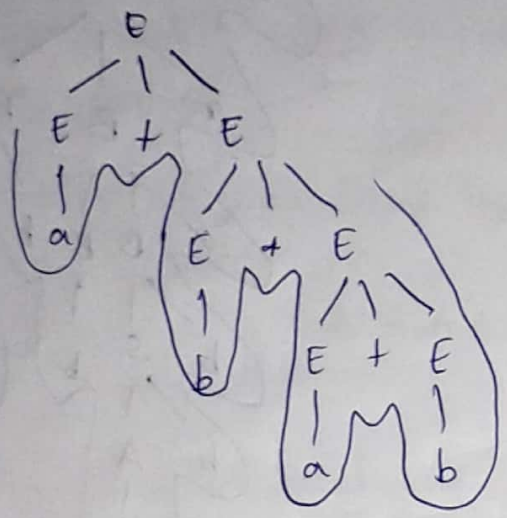
$E \rightarrow a+b*\underline{E}$ [$\because E \rightarrow b$]

$E \rightarrow a+b*E+E$ [$\because E \rightarrow E+E$]

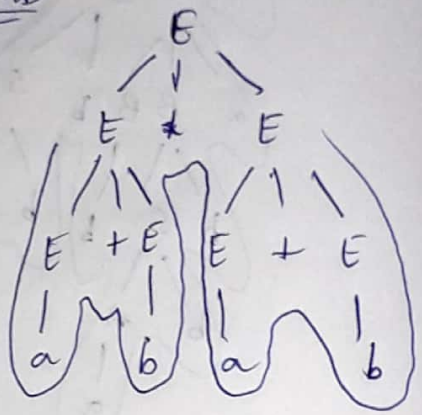
$E \rightarrow a+b*a+\underline{E}$ [$\because E \rightarrow a$]

$E \rightarrow a+b*a+b$ [$\because E \rightarrow b$]

LMD



RMD



RMD

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

$E \rightarrow E*E+\underline{E}$ ($\because E \rightarrow E+E$)

$E \rightarrow E*E+b$ [$\because E \rightarrow b$)

$E \rightarrow E*\underline{a}+b$ [$\because E \rightarrow a$)

$E \rightarrow E+\underline{E}*a+b$ [$\because E \rightarrow E+E$)

$E \rightarrow E+b*a+b$ [$\because E \rightarrow b$]

$E \rightarrow a+b*a+b$ [$\because E \rightarrow a$]

- Ambiguous Grammar
  - A grammar G is said to be ambiguous if it generates more than one parse tree

Ex $E \rightarrow E+E \mid E*E \mid (E) \mid id$

i/p id + id * id

LMD $E \to E+E$

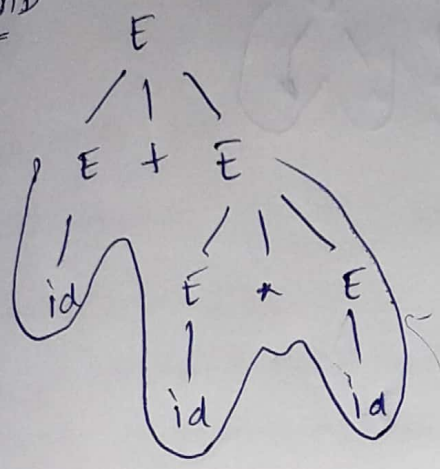
$E \to id+E$ ($\because E \to id$)

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

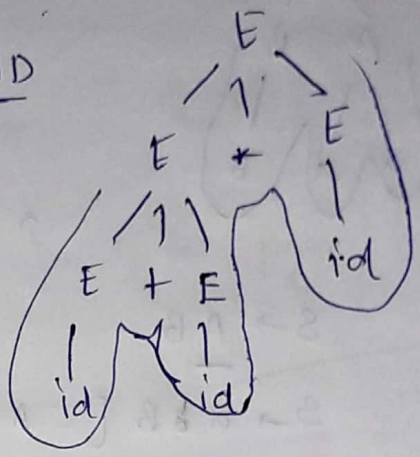
$E \to id+id*E$ ($\because E \to id$)

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

LMD



LMD



LMD $E \to E*E$

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

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

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

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

We have more than one parse tree, therfore this is an ambigous grammar.

Ex Prove the following grammar as ambiguous.

$S \to AB$

$B \to ab/b$

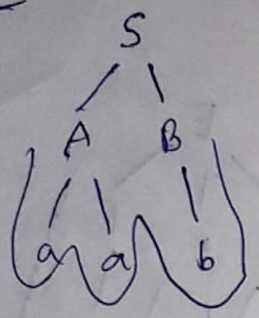
$A \to aa/a$ i/p aab

LMD. $S \rightarrow AB$

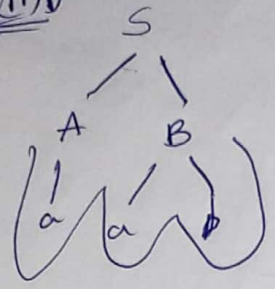
$S \rightarrow aaB$ [$\because A \rightarrow aa$]

$S \rightarrow aab$ [$\because B \rightarrow b$]

LMD



LMD



LMD $S \rightarrow AB$.

$S \rightarrow aaB$ [$\because A \rightarrow a$]

$S \rightarrow aab$ [$B \rightarrow ab$]

As more than one parse tree, it's an ambiguous grammar

- Capabilities of context free Grammar.
  - → CFG is useful to describe most of the programming language.
  - → If the grammar is properly designed then, an efficient parser can be constructed automatically.
  - → Using the features of associativity and precedence information suitable grammars for expressions can be constructed.
  - → CFG is capable of describing nested structures like balanced paranthesis, matching begin-end. and

corresponding if-then else and so on.

01/02/2021

- Syntactic Errors

→ It appears in the Syntax Analyzer page.

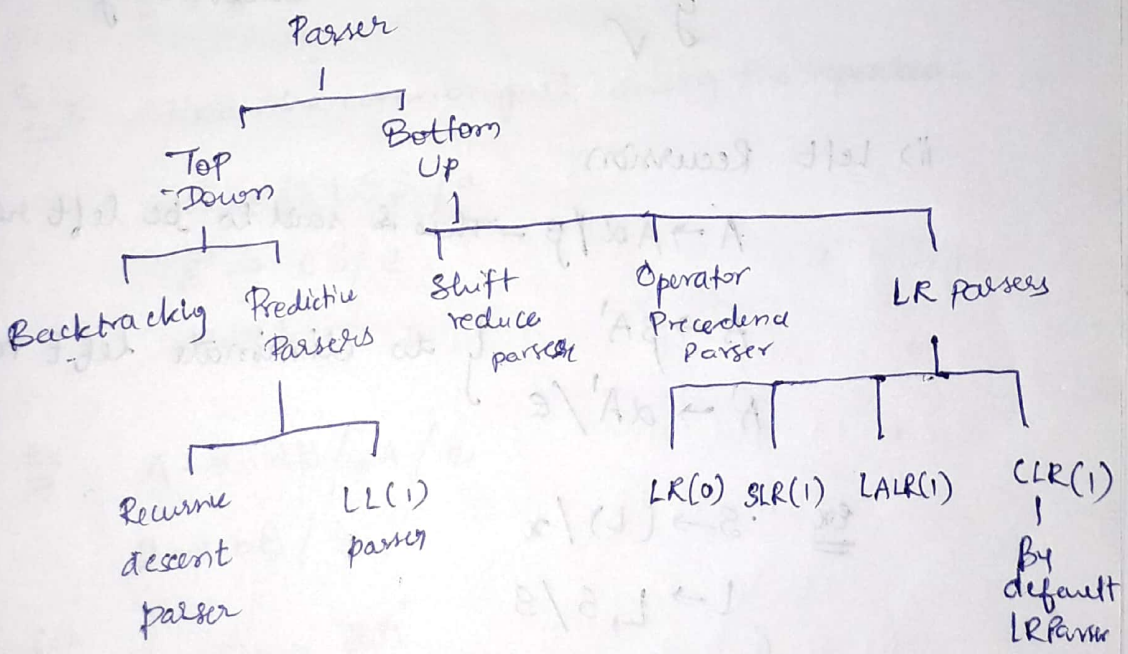
→ Syntax errors are
  - Error in structure — if(num = 2) { - }
  - Missing operators — int x = 2
  - Unbalanced paranthesis — x = (y+2

- Semantic Errors

→ It appears in the Semantic Analyzer page.

→ Semantic errors are
  - Incompatible type of operands — int a = "hi";
  - Undeclared variable — int x; if(x<2) { y = 2+z; }
  - Not Matching the actual argument with formal argument — int x = 8.5;

- Parser



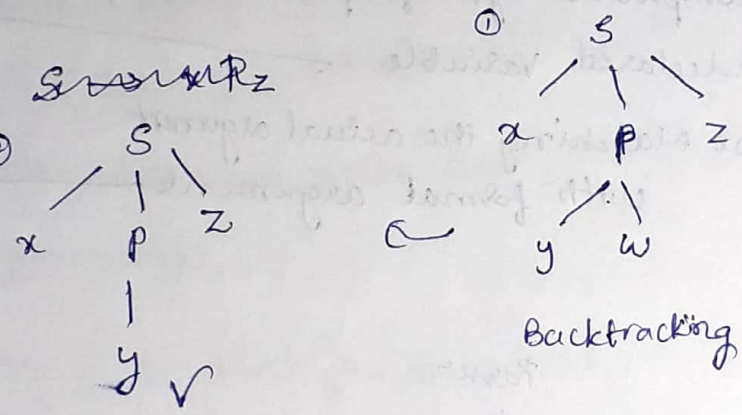
- Top Down Parsers
  - The parse tree is generated from top to bottom (root to leaves)
  - The derivation terminates when required input string terminates.
  - The main task in top down passing is to find the appropriate production rule in order to produce the correct input string.
- Backtracking
- Problems in Top-Down Passing

i) Backtracking

Ex $S \rightarrow xPz$ i/p xyz

$P \rightarrow yW/y$

Soln



ii) Left Recursion

$A \rightarrow A\alpha / \beta$ — This is said to be left recursion

$A \rightarrow \beta A'$
$A' \rightarrow \alpha A' / \epsilon$ } to eliminate left recursion

Ex $S \rightarrow (L) / x$

$L \rightarrow L, S / S$

$A \rightarrow A\alpha / \beta$ where

Soln $A = L, \alpha = ,S, \beta = S$

$\therefore L \rightarrow SL'$

$L' \rightarrow ,SL'/\epsilon$

Ex $A \rightarrow ABd/Aa/a$

$B \rightarrow Be/b$

Soln $A \rightarrow ABd/a$ $\quad A \rightarrow Aa/a$

$A \rightarrow aA'$ $\quad A \rightarrow aA'$

$A' \rightarrow BdA'/e$ $\quad A' \rightarrow aA'/\epsilon$

$\therefore A' \rightarrow BdA'/aA'/\epsilon$ and $A \rightarrow aA'$

$B \rightarrow Be/b$

$B \rightarrow bB'$

$B' \rightarrow eB'/\epsilon$

iii) Left factoring

Ex $S \rightarrow iEtS/iEtSeS/a$

$E \rightarrow b$

Soln Find the common part among the repeated.

$S \rightarrow iEtSS'/a$

$S' \rightarrow eS/\epsilon$

$E \rightarrow b$

Ex $A \rightarrow aAB/aA/a$

$B \rightarrow bB/b$

Soln $A \rightarrow aA'$

$A' \rightarrow AB/A/\epsilon$

$B \rightarrow bB'$

$B' \rightarrow B/\epsilon$

iv) Ambiguous

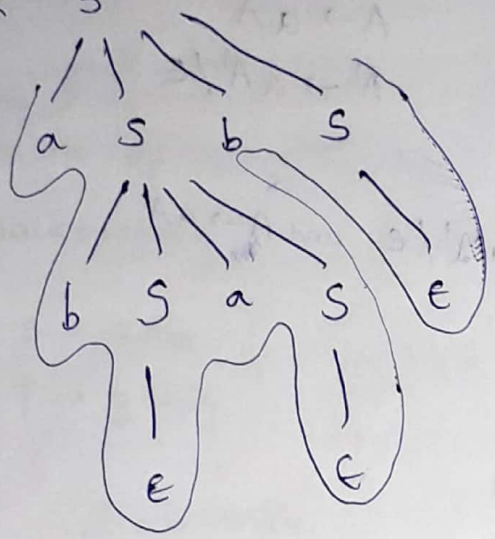
ex $S \to aSbS$

$S \to bSaS$ i/p abab

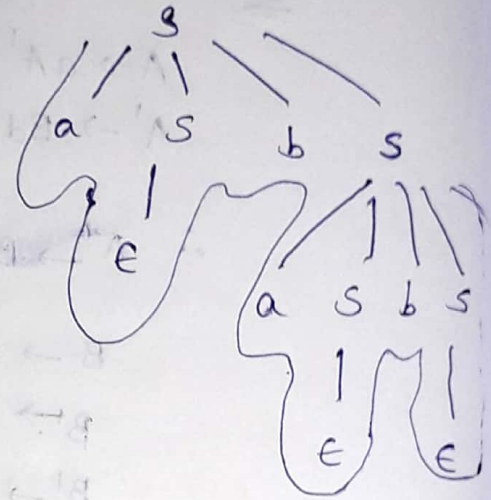
$S \to \epsilon$

LMD

Sol



LMD



Two leftmost derivations.
Hence this is ambiguous.

- Predictive Parsers (Non-Backtracking)

→ Recursive Descent Parser → A parser that uses collection of recursive procedures for parsing the given i/p string is called recursive descent parsers.

→ Basic steps to construct RDP

i) If the input symbol is non-terminal, then a call to the procedure corresponding to the non-terminal is made.

ii) If the i/p symbol is a terminal, then it is matched with the lookahead symbol from the i/p.

↓
getting next character (pointer)

iii) If the production rule has many alternatives, then all these alternatives have to be combined into a single body of procedure.

iv) The parser should be activated by a procedure corresponding to the start symbol.

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

Soln
E, E' are non-terminals → call a procedure } according to production rule
+, i, ϵ, are terminals → match them. }

∴
```
E()  // E
{
 if (l == 'i')   // l → lookahead
 {
  match('i'); // i
  E'();  // E'
 }
}
```

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

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

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

02/02/2022

- LL(1) Parsers-

L- Scanning the i/p from left to right

L- Left most derivation.

1- look ahead symbol (at a time only 1 character is scanned)

→ LL(1) parsor is an algorithm. Steps are-

1) → (1) FIRST } functions
(2) FOLLOW } must be followed

2) → construction of LR(1) parsing table by using FIRST and FOLLOW,

3) → Scan the i/p string by using LL(1) parsing table.

(first of Terminal is always a terminal

Ex
S → aABCD
A → b
B → c
C → d
D → e

FIRST(A) = FIRST(b) ⇒ {b}

FOLLOW(A) = After A ⇒ B ∴ FIRST(B) = {c}

Soln

FIRST

FIRST(S) = {a}
FIRST(A) = {b}
FIRST(B) = {c}
FIRST(C) = {d}
FIRST(D) = {e}

FOLLOW

FOLLOW(S) = {$}
FOLLOW(A) = {c}
FOLLOW(B) = {d}
FOLLOW(C) = {e}
FOLLOW(D) = {$}

As there is no next, go for follow(S)

Ex
S → ABC DE
A → a/ε
B → b/ε
C → c
D → d/ε
E → e/ε

Soln

FIRST

FIRST(S) = {a, b, c}
FIRST(A) = {a, ∈}
FIRST(B) = {b, ∈}
FIRST(C) = {c}
FIRST(D) = {d, ∈}
FIRST(E) = {e, ∈}

FOLLOW

FOLLOW(S) = {$}
FOLLOW(A) = {b, c}
FOLLOW(B) = {c}
FOLLOW(C) = {d, e}
FOLLOW(D) = {e, $}
FOLLOW(E) = {$}

ex

S → Bb/Cd
B → aB/ϵ
C → cC/ϵ

Soln

FIRST

FIRST(S) = {a, b, c, d}
FIRST(B) = {a, ϵ}
FIRST(C) = {c, ϵ}

FOLLOW

FOLLOW(S) = {$}
FOLLOW(B) = {b}
FOLLOW(C) = {d}

** ex

E → TE'
E' → +TE'/ϵ
T → FT'
T' → *FT'/ϵ
F → id / (E)

Soln

FIRST

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

FOLLOW

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

Table construction using first and follow

LL(1) Parsing table

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

To place the productions, see the first( ) then if 'ϵ' is there go for follow to know the position. The other empty spots were errors.

Scan the i/p string → id+id*id$ (end with $)

Stack	i/p	Action
$E	id+id*id$	—
$E'T	id+id*id$	$E \to TE'$
$E'T'F	id+id*id$	$T \to FT'$
$E'T'id	id+id*id$	$F \to id$
$E'T' → pop	+id*id$	—
$E'	+id*id$	$T' \to \epsilon$
$E'T+ → pop	+id*id$	$E' \to +TE'$
$E'T	id*id$	—
$E'T'F	id*id$	$T \to FT'$
$E'T'id	id*id$	$F \to id$
$E'T' → pop	*id$	—
$E'T'	*id$	$T' \to *FT'$
$E'T'F* → pop	id$	—
$E'T'F	id$	

$E'T'id	id$	F→id
$E'T'	$	—
$E'	$	T'→ϵ
$	$	E'→ϵ

∴ Accept.

3/03/2022

Ex Show that the following grammar is LL(1) or not?

S→ AaAb / BbBa

A→ϵ

B→ϵ

Sol

FIRST

FIRST(S) = {a, b}

FIRST(A) = {ϵ}

FIRST(B) = {ϵ}

Follow

FOLLOW(S) = {$}

FOLLOW(A) = {a, b}

FOLLOW(B) = {b, a}

Table construction using first and follow

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

Scan the i/p string i/p ba

Stack	i/p	Action
$S	ba$	—
$bAaA	✗ not matching	S→AaAb

$ aBbB	ba$	S → BbBa
$ aBb	ba$	B → ϵ
$ aB	a$	—
$ a	a$	B → ϵ
$	$	—

∴ Accept

Ex: S → iEtSS' / a
S' → eS / ϵ
E → b

Soln:

FIRST

FIRST(S) = {i, a}
FIRST(S') = {e, ϵ}
FIRST(E) = {b}

FOLLOW

FOLLOW(S) = {$, e}
FOLLOW(S') = {$, e}
FOLLOW(E) = {t}

Table construction using First and follow

	a	b	e	i	t	$
S	S → a			S → iEtSS'		
S'			S' → eS S' → ϵ			S' → ϵ
E		E → b				

As there is more than one production, the given grammar is not an LL(1).

Bottom-Up Parsing

→ The parse tree is constructed from Bottom to up i.e from leaves to the root.

→ In this process, the i/p symbols are placed at the leaf nodes after successful parsing.

i) Shift Reduce Parser (SR Parser)
ii) Operator Precedence Parser
iii) LR Parsers

Shift Reduce Parser

- It requires - input buffer
  - stack
- i/p buffer is used to store the i/p string
- stack is for storing and accessing the LHS and RHS of the rules.
- The parser performs the following basic operations
  - i) Shift
  - ii) Reduce
  - iii) Accept (ACCEPT)
  - iv) Error (ERROR)

* First shift only.
* RHS of non-terminal can be reduced with LHS of the production.

Ex Consider the grammar. Perform the SR Parser of given i/p string → $id_1 - id_2 * id_3$

$E \rightarrow E-E / E*E / id$

Soln

Stack	i/p Buffer	Action
$	$id_1 - id_2 * id_3 \$$	Shift
$ id_1	$-id_2 * id_3 \$$	Reduce by $E \rightarrow id$
$ E	$-id_2 * id_3 \$$	Shift
$ E -	$id_2 * id_3 \$$	Shift
$ E - id_2	$* id_3 \$$	Reduce by $E \rightarrow id$
$ E - E	$* id_3 \$$	Reduce by $E \rightarrow E-E$
$ E	$* id_3 \$$	Shift
$ E *	$id_3 \$$	Shift

$ E * id3	$	Reduce by E → [redacted]
$ E * E	$	Reduce by E → E*E
$ E	$	Accept

Ex S → TL ;
T → int / float
L → L, id / id

I/p string → int id, id ;

Soln

Stack	I/p Buffer	Action
$	int id, id ; $	Shift
$ int	id, id ; $	Reduce by T → int
$ T	id, id ; $	Shift
$ T id	, id ; $	Reduce by L → id
$ TL	, id ; $	~~Reduce by~~ Shift
$ TL ,	id ; $	Shift
$ TL , id	; $	Reduce by L → L, id
$ TL	; $	Shift
$ TL ;	$	Reduce by S → TL ;
$ S	$	Accept

Ex S → 0S0 / 1S1 / 2 i/p → 10201

Soln

Stack	I/p Buffer	Action
$	10201 $	Shift
$1	0201 $	shift

$10	201$	Shift
$102	01$	Reduce by S→2
$105	01$	shift
$1050	1$	Reduce by S→0S0
$1S	1$	shift
$1S1	$	Reduce by S→1S1
$S	$	Accept

09/02/2022

ex: S → (L) / a
L → L,S / S

i/p → (a,(a,a))

Sol:

Stack	i/p Buffer	Action
$	(a,(a,a))$	Shift
$(	a,(a,a))$	Shift
$(a	,(a,a))$	Reduce by S→a
$(S	,(a,a))$	~~Shift~~ Reduce by L→S
$(L	,(a,a))$	Shift
$(L,	(a,a))$	Shift
$(L,(	a,a))$	Shift
$(L,(a	,a))$	Reduce by S→a
$(L,(S	,a))$	Reduce by L→S
$(L,(L	,a))$	Shift
$(L,(L,	a))$	Shift

$ (L, (L, a	)) $	Reduce by S → a
$ (L, (L, S	)) $	~~Shift~~ Reduce by L → L, S
$ (L, (L	)) $	Shift
$ (L, (L)	) $	Reduce by S → (L)
$ (L, S	) $	~~Shift~~ Reduce by L → L, S
$ (L	) $	Shift
$ (L)	$	Reduce by S → (L)
$ S	$	Accept

- Operator Precedence Parser (OPP)
  - Operator Grammar - grammar used to generate mathematical operations is called Operator grammar.
  - Ex E → E+E ✓ (No two non-terminals are adjacent to each other) - OPP
    E → EAE X
  - Applicable for ambiguous grammar.
    Ex E → E+E/E*E / id.
  - First prepare operational relation table after checking if it is operator precedence grammar.

(old method)

f/g	id	*	+	$
id	-	·>	·>	·>
*	<·	·>	·>	·>
+	<·	<·	·>	·>
$	<·	<·	<·	-

left shift * > * shift
left shift + > + shift

- consider one string and check the same using the table.
- i/p - id + id * id $

when left < right - push()

~~right > left~~

left > right - pop() and substitute.

empty stack

→ | $ |

→ | $ | id |

→ | $ | id | + |

→ | $ | id | + | id |

→ | $ | id | + | id | * |

→ | $ | id | + | id | * | id |

→ | $ | id | + | id | * | id |

→ | $ | id | + | id | * | id |

→ | $ | id | + | id | * | id |

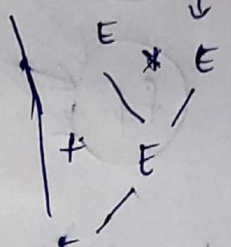
→ | $ | id | + | id | * | id |

↓ pointer moves.

id + id * id $

↓

E



↓

After Pop() again compare with the next top of the stack

if condition satisfies, push() or again pop().

→ Again the same stack ~~to~~ → empty.

- i/p id + id + id $

empty stack

→ | $ |

→ | $ | id |

→ | $ | id | + |

→ | $ | id | + | id |

→ | $ | id | + | id | * | id |

left+ > right+

→ | $ | id | * | id | + | id |

→ | $ | id | * | id | + | id |

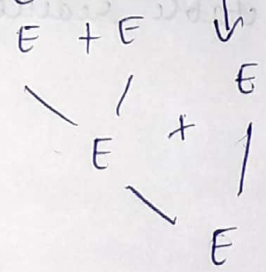
→ | $ | id | * | id | * | id |

Pointer → moves

↓

id + id + id $

↓ ↓ ↓

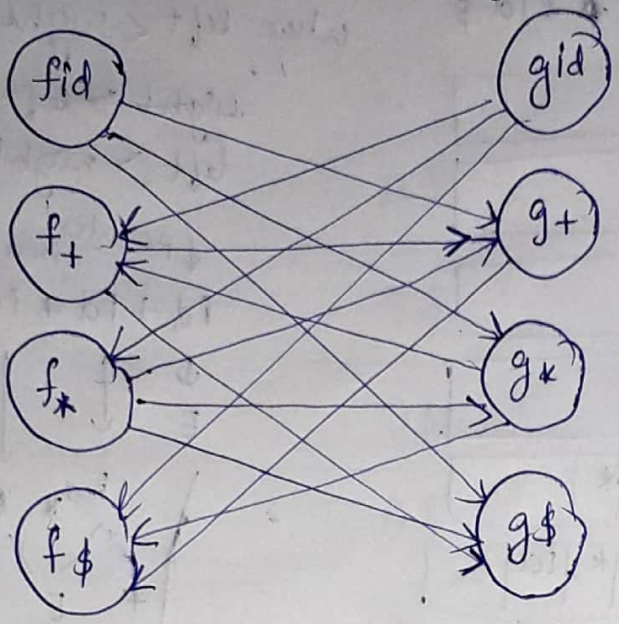


→ Again the stack → empty.

- Disadvantage - for n taminals → n^2 →

Therefore we use operator precedence grammar.

- We construct operator precedence graph. (from greater to lower) (→ ←)



- longest path from fid - g* - f+ - g+ - f$
- longest path from gid - f* - g* - f+ - g+ - f$
- Create operator function table from above paths

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

- table reduced to 2n values for n terminals.

09/2/23

Unit-3

- LR Parsers - Bottom-Up Parsers
  L- Scanning the i/p from left to right
  R- right most derivation in reverse.
- Divided into 4 -
  (i) LR(0) (ii) SLR(1) (iii) LALR(1) (iv) CLR(1)

  LR(0) items — LR(1) items.

Ex $S \to AA$
$A \to aA/b$

Sol we use 2 methods -
1) closure
2) goto

let
I_0 :
$S' \to .S$
$S \to .AA$
$A \to .aA/.b$
} → changed it to Augmented grammar.

Step-1 Canonical collection of LR(0) ~~tables~~ items.
'.' indicates LR(0) item. So for closure put a dot '.' ~~before~~ after → After '.' if non-terminal, then write productions related to it. else stop.

I_1 : goto (I_0, S) → (go to I_0 where after '.' S is there. then move '.' and check.)
$S' \to S.$ → final item

I_2 : goto (I_0, A)
$S \to A.A$
$A \to .aA/.b$
(after '.' we have another non-terminal, so apply, closure of that non-terminal)

I_3 : goto (I_0, a)
$A \to a.A$
$A \to .aA/.b$

I_4 : goto (I_0, b)
$A \to b.$ → final item

Once all the ones are traversed once continue doing until we get final item for all.

I_5 : goto (I_2, A)

$S \rightarrow AA.$ → final item

~~I_6~~ : goto (I_2, a)

this is same as I_3 (So it's not another)

goto (I_2, b)

Same as I_4

I_6 : goto (I_3, A)

$A \rightarrow aA.$ → final item

goto (I_3, a)

Same as I_3

goto (I_3, b)

Same as I_4

Always I_1, $

All are done, there are 7 items (0, 6).

Step-2 Construction of table. - Terminals - Shift (S) Reduce (R) Non-Terminals Just no's

Items	Action			Goto	
	a	b	$	A	S
0	S_3	S_4		2	1
1			Accept		
2	S_3	S_4		5	

3	S_3	S_4		6
4	r_3	r_3	r_3	
5	r_1	r_1	r_1	
6	r_2	r_2	r_2	

4, 5, 6 are final items. So to write, we used the productions

$S \to AA$ — (1) r_1

$A \to aA \mid b$

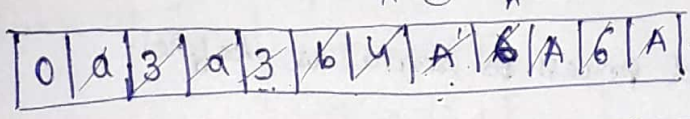
(2) r_2 (3) r_3

Depending on the final items at 4, 5, 6, we write.

Step 3 Use the i/p string give. Parse it.

i/p ⇒ a a b b $

Goto methodology



10/02/2022

Actual methodology Goto only when reduction.

Stack	i/p Buffer	Action	Goto	Parsing Action
$0	aabb$	$[0,a]=S_3$	-	Shift
$0a3	abb$	$[3,a]=S_3$	-	Shift
$0a3a3	bb$	$[3,b]=S_4$	-	Shift
$0a3a3b4	b$	$[4,b]=r_3$	$[3,A]=6$	Reduce by $A \to b$
$0a3a3A6	b$	$[6,b]=r_2$	$[3,A]=6$	Reduce by $A \to aA$
$0a3A6	b$	$[6,b]=r_2$	$[0,A]=2$	Reduce by $A \to aA$
$0A2	b$	$[2,b]=S_4$	-	Shift
$0A2b4	$	$[4,\$]=r_3$	$[2,A]=5$	Reduce by $A \to b$

until shift comes don't move the pointer.

The top of stack no. of the pointer variable.

Ex. I⓪④

when $r_{1,2,3}$ comes, goto the productions and count the no. of items to the right and multiply x 2. that many no.s must be popped out. And write the respective substitution and calculate it with no. present behind it.

$0A2A5	$	[5,$] = r_1	[0,S] = 1	Reduce by S → AA
$0S1	$	[1,$] = Accept	–	Accept

SLR(1) Parser

Eg: Same as with step 1 and step 2 till 3 are same. from 4 table reduction changes.

$S \to AA$
$A \to aA / b$

FOLLOW(S) = {$}
FOLLOW(A) = {a, b, $}

I_4: A → b.
I_5: S → AA.
I_6: A → aA.

write reductions only in the concerned non terminal FOLLOW.

	a	b	$	A	S
4	r_3	r_3	r_3		
5			r_1		
6	r_2	r_2	r_2		

i/p string and process is same as before.

Eg: Construct a SLR(1) parsing table for following grammar.

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

Soln: Construction of canonical form of LR(0) items closure and goto.

I_0:
$E' \to .E$
$E \to .E+T / .T$
$T \to .T*F / .F$
$F \to .(E) / .id$

I_1: goto(I_0, E)
$E' \to E.$
$E \to E.+T$

I_2: goto (I_0, T)

$E \to T.$

$T \to T. * F$

I_3: goto (I_0, F)

$T \to F.$

→ Final item

I_4: goto $(I_0, ()$

$F \to (.E)$

$E \to .E + T / .T$

$T \to .T * F / .F$

$F \to .(E) / .id$

I_5: goto (I_0, id)

$F \to id.$

→ final item

I_6: goto $(I_1, +)$

$E \to E+.T$

$T \to .T * F / .F.$

$F \to .(E) / .id$

~~I_7~~ goto (I_4, E)

~~$E \to E.+T$~~

$E \to T.$

same as I_2

I_7: goto $(I_2, *)$

$T \to T*.F$

$F \to .(E) / .id$

I_8: goto (I_4, E)

$F \to (E.)$

~~$E \to E.+T$~~

goto (I_6, F)

Same as I_3

goto (I_4, F)

Same as I_3

goto (I_4, id)

same as I_5

goto $(I_6, ()$

same as I_4.

goto $(I_4, ()$

Same as I_4

I_9: goto (I_6, T)

$E \to E+T.$

$T \to T. * F.$

goto (I_6, id)

same as I_5

I_{10}: goto (I_7, F)

$T \to T*F.$

→ Final item

goto $(I_7, ()$

Same as I_4

goto (I_7, id)

Same as I_5

I_{11}: goto $(I_8,))$

$F \to (E).$ → final item

goto $(I_8, +)$

same as I_6

goto $(I_9, *)$

Same as I_7

~~goto~~ FOLLOW $(E) = \{\$, +,)\}$

FOLLOW $(T) = \{\$, +,), *\}$

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

Step-2 Construction of SLR(1) parsing table.

	Action						Goto		
	id	+	*	(	)	$	E	T	F
0	s_5			s_4			1	2	3
1		s_6				Accept			
2		r_2	s_7		r_2	r_2			
3		r_4	r_4		r_4	r_4			
4	s_5			s_4			8	2	3
5		r_6	r_6		r_6	r_6			
6	s_5			s_4				9	3
7	s_5			s_4					10
8		s_6			s_{11}				
9		r_1	s_7		r_1	r_1			
10		r_3	r_3		r_3	r_3			
11		r_5	r_5		r_5	r_5			

$E \rightarrow E+T$ ① / T ②

$T \rightarrow T*F$ ③ / F ④

$F \rightarrow (E)$ / id ⑥ ⑤

consider $r_1, r_2, r_3, r_4, r_5, r_6$ and place them in the table.

Step 3 Passing the i/p string using SLR(1) parsing table.

i/p string ⇒ id * id + id $

gate methodology

id * id + id $
↑ ↑ ↑ ↑ ↑ ↑

0	id	5	F	3	T	2	*	7	id	5	F	10	T	2	E	1	+	6	id	5	F	3	T	9	E	1	-

- Accept

10/02/2022

Ex $E \to E+T/T$ i/p a*b+a

$T \to TF/F$

$F \to F*/a/b$

Sol Step 1 - Construction of canonical form of LR(0) items.

I_0: $E' \to .E$
$E \to .E+T/.T$
$T \to .TF/.F$
$F \to .F*/.a/.b$

I_1 : goto(I_0, E)
$E' \to E.$
$E \to E.+T$

I_2 : goto(I_0, T)
$E \to T.$
$T \to T.F$
$F \to .F*/.a/.b$

I_3 : goto(I_0, F)
$T \to F.$
$F \to F.*$

I_4 : goto(I_0, a)
$F \to a.$ → final item

I_5 : goto(I_0, b)
$F \to b.$
→ final item

I_6 : goto$(I_1, +)$
$E \to E+.T$
$T \to .TF/.F$
$F \to .F*/.a/.b$

I_7 : goto(I_2, F)
$T \to TF.$
$F \to F.*$

goto(I_2, a)
Same as I_4

goto(I_2, b)
Same as I_5

I_8 : goto$(I_3, *)$
$F \to F*.$ → final item

I_9 : goto(I_6, T)
$E \to E+T.$
$T \to T.F$
$F \to .F*/.a/.b$

goto (I_6, F) — same as I_3

goto (I_6, a) — same as I_4

goto (I_6, b) — same as I_5

goto $(I_7, *)$ — same as I_8

goto (I_9, F) — same as I_7

goto (I_9, a) — same as I_4

goto (I_9, b) — same as I_5

Follow(E) = {$, +}

Follow(T) = {$, +, a, b}

Follow(F) = {$, +, a, b, *}

Step-2 construction of SLR(1) parsing table

Items	Action					Goto		
	+	*	a	b	$	E	T	F
0			s_4	s_5		1	2	3
1	s_6				Accept			
2	r_2		s_4	s_5	r_2			7
3	r_4	s_8	r_4	r_4	r_4			
4	r_6	r_6	r_6	r_6	r_6			
5	r_7	r_7	r_7	r_7	r_7			
6			s_4	s_5			9	3
7	r_3	s_8	r_3	r_3	r_3			

8	r_5	r_5	r_5	r_5	r_5				
9	r_1		S4	S5	r_1				7

① ②

$E \rightarrow E+T / T$

$T \rightarrow TF / F$

③ ④

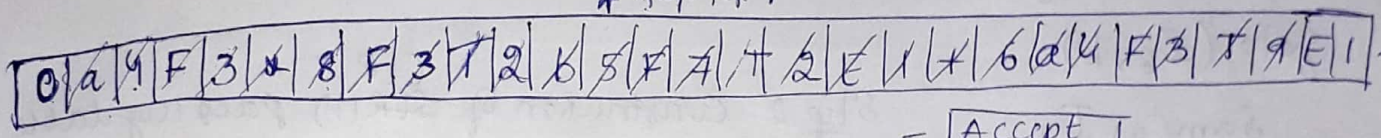
$F \rightarrow F* / a / b$

⑤ ⑥ ⑦

$r_1, r_2, r_3, r_4, r_5, r_6, r_7$

Step-3 Parsing i/p string using SLR(1) parsing table.

a * b + a $



- Accept

- CLR(1) Parser

ex $S \rightarrow AA$

$A \rightarrow aA / b$

Step-1 canonical form of LR(1) items.

lookahead symbol

↓ → at these symbols, we place reduction symbol in parsing table.

Sol I_0: $S' \rightarrow .S, \$$

$S \rightarrow .AA, \$$

$A \rightarrow .aA, a/b$

$/.b, a/b$

I_1: goto (I_0, S)

$S' \rightarrow S., \$$

- final item

I_2: goto (I_0, A)

$S \rightarrow A.A, \$$

$A \rightarrow .aA, \$$

$/.b, \$$

I_3: goto (I_0, a)

$A \rightarrow a.A, a/b$

$A \rightarrow .aA, a/b$

$/.b, a/b$

I_4: goto (I_0, b)

$A \rightarrow b., a/b$

- final item

I_5: goto (I_2, A)

$S \to AA.$, \$

→ final item

I_6 : goto (I_2, a)

$A \to a.A$, \$

$A \to .aA$, \$

$/.b$, \$

I_7: goto (I_2, b)

$A \to b.$, \$

- final item

I_8: goto (I_3, A)

$A \to aA.$, $a|b$

- final item

goto (I_3, a)

Same as I_3

goto (I_3, b)

Same as I_4

I_9: goto (I_6, A)

$A \to aA.$, \$

- final item

goto (I_6, a)

Same as I_6

goto (I_6, b)

Same as I_7

Step-2 Construction of CLR(1) parsing table

Items	Action			Goto	
	a	b	\$	S	A
0	S_3	S_4		1	2
1			Accept		
2	S_6	S_7			5
3	S_3	S_4			8
4	r_3	r_3			
5			r_1		
6	S_6	S_7			9
7			r_3		
8	r_2	r_2			
9			r_2		

$S \to AA$ (1)

$A \to aA$ (2) $| b$ (3)

r_1, r_2, r_3

i/p- b b $

0	b	4	A	2	b	7	A	5	5	1	Accept

Step-3 same as before

Ex

$S \to CC$

$C \to cC / d$

Soln Step-1 canonical formation of LR(1) items.

I_0: $S' \to .S, \$$

$S \to .CC, \$$

$C \to .cC, c/d$

$/.d, c/d$

I_1: goto(I_0, S)

$S' \to S., \$$

- final item

I_2: goto(I_0, C)

$S \to C.C, \$$

$C \to .cC, \$$

$/.d, \$$

I_3: goto(I_0, c)

$C \to c.C, c/d$

$C \to .cC, c/d$

$/.d, c/d$

I_4: goto(I_0, d)

$C \to d., c/d$

- final item

I_5: goto(I_2, C)

$S \to CC., \$$

- final item

I_6: goto(I_2, c)

$C \to c.C, \$$

$C \to .cC, \$$

$/.d, \$$

I_7: goto(I_2, d)

$C \to d., \$$

- final item

Same method as before

- LALR(1) - Look Ahead LR parser

Ex

$S \to AA$ (1)

$A \to aA$ (2) $| b$ (3)

Same items as before but now, we combine five states.

$I_0, I_1, I_2, (I_3, I_6) I_{36}, (I_4, I_7) I_{47}, I_5, (I_8, I_9) I_{89}$

∴ The final states are →

$I_0, I_1, I_2, I_{36}, I_{47}, I_5, I_{89}$.

Step-2 construction of LALR(1) parsing table.

	a	b	$	S	A
0	S_{36}	S_{47}		1	2
1			Accept		
2	S_{36}	S_{47}			5
36	S_{36}	S_{47}			89
47	r_3	r_3	r_3		
5			r_1		
89	r_2	r_2	r_2		

Step-3 Input string

Ex $S \to Aa / bAc / Bc / bBa$

$A \to d$

$B \to d$

LR(1)/CLR(1) LR parser but not LALR(1)

Sol Step 1 Construction of canonical form of LR(1) items.

I_0: $S' \to .S, \$$

$S \to .Aa, \$$

$/. bAc, \$$

$/. Bc, \$$

$/. bBa, \$$

$A \to .d, a$

$B \to .d, c$

I_1 : goto(I_0, S)

$S' \to S.,\$$ - final item

I_2 : goto(I_0, A)

$S \to A.a, \$$

I_3 : goto(I_0, b)

$S \to b.Ac, \$$

$S \to b.Ba, \$$

$A \to .d, c$

$B \to .d, a$

I_4 : goto(I_0, B)

$S \to B.c, \$$

I_5 : goto(I_0, d)

$A \to d., a$

$B \to d., c$

final item

I_6 : goto(I_2, a)

$S \to Aa., \$$ - final item

I_7 : goto(I_3, A)

$S \to bA.c, \$$

I_8 : goto(I_3, B)

$S \to bB.a, \$$

I_9 : goto(I_3, d)

$A \to d., c$

$B \to d., a$

- final item

I_{10} : goto(I_4, c)

$S \to Bc., \$$

- final item

I_{11} : goto(I_7, c)

$S \to bAc., \$$

final item.

① ② ③ ④

$S \to Aa \mid bAc \mid Bc \mid bBa$

$A \to d$ - ⑤

$B \to d$ - ⑥

I_{12} : goto(I_8, a)

$S \to bBa., \$$

- final item

Action

	a	b	c	d	$	S	A	B
0		S_3		S_5		1	2	4
1					Accept			
2	S_6							
3				S_9			7	8
4			S_{10}					
5 9	r_5, r_6		r_6, r_5					
6					r_1			

7			S_{11}					
8	S_{12}							
10				r_3				
11				r_2				
12				r_4				

Reduce / Reduce conflict in 59 cell. So LALR(1) is not possible. but it is LR(1), i.e CLR(1) is possible.

Step 3 i/p string - bd a $
↑ ↑ ↑ ↑

0	b	3	d	a	B	8	a	b	S	1	Accept

17/02/22

- Comparison of LR parsers.

SLR	LALR	CLR
i) Smallest in size	i) LALR and SLR have the same size.	i) Largest in size.
ii) It is an earliest method based on FOLLOW function.	ii) This method is applicable to wider clause than SLR.	ii) This method is most powerful than SLR and LALR.
iii) This method exposes less syntatic features than that of LR Parser	iii) Most of the syntatic features of a language expressed in LALR.	iii) This method exposes less syntatic features than that of SLR and LALR.
iv) Error detection is not immediate in SLR	iv) Error detection is not immediate in LALR.	iv) Immediate error detection is done by LR parsers.

v) requires less time and space complexity.

v) time and space complexity is more is LALR but efficient methods exist for constructing the LALR parsers directly.

v) time and space complexity is more for CLR parser.

- Comparison of top-down Parser and Bottom up parser

Top-Down Parser	Bottom-Up Parser
i) Parse tree can be built from root to the leaves	i) Parse tree can be built from leaves to root.
ii) Simple to implement	ii) Complex to implement
iii) less efficient parsing techniques. Various problems that occur in top-down parsing are backtracking, left recursion.	iii) handles the ambiguous grammar. conflicts occur in parsing table.
iv) applicable to small class of languages.	iv) applicable to broad class of languages.
v) Various parsing techniques are - recursive descent parser, LL(1) parser	v) Various parsing techniques are - shift reduce parser, operator precedence parser and LR parser.

- Handling Ambiguous grammar.

Ex SLR ① ② ③ ④

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

(i) Construction of canonical construction of LR(0) items

I_0: $E' \to .E$
$E \to .E+E$
$.E*E$
$.(E)$
$.id$

I_1 : goto (I_0, E)
$E' \to E.$
$E \to E.+E$
$E \to E.*E$

I_2 : goto $(I_0, ()$
$E \to (.E)$
$E \to .E+E$
$.E*E$
$.(E)$
$.id$

I_3 : goto(I_0, id)

$E \to id \cdot$ → final item

I_4 : goto (I_1, +)

$E \to E + \cdot E$

$E \to \cdot E + E$

$\cdot E * E$

$\cdot (E)$

$\cdot id$

I_5 : goto (I_1, *)

$E \to E * \cdot E$

$E \to \cdot E + E$

$\cdot E * E$

$\cdot (E)$

$\cdot id$

I_6 : goto (I_2, E)

$E \to (E \cdot)$

$E \to E \cdot + E$

$E \to E \cdot * E$

goto (I_2, ()

Same as I_2

goto (I_2, id)

same as I_3

I_7 : goto (I_4, E)

$E \to E + E \cdot$

~~→ final item~~

$E \to E \cdot + E$

$E \to E \cdot * E$

goto (I_4, ()

same as I_2

goto (I_4, id)

same as I_3

I_8 : goto (I_5, E)

$E \to E * E \cdot$

$E \to E \cdot + E$

$E \to E \cdot * E$

goto (I_5, ()

same as I_2

goto (I_5, id)

same as I_3.

I_9 : goto (I_6,))

$E \to (E) \cdot$

→ final item

~~I_{10}~~ : goto (I_6, +)

Same as I_4

goto (I_6, *)

same as I_5

goto (I_7, +)

Same as I_4

goto (I_7, *)

Same as I_5

goto (I_8, +)

same as I_4

goto (I_8, *)

Same as I_5

Step-2 Construction of parsing table ~~using~~ ~~the~~ ~~table~~ SLR(1)

FOLLOW(E) = {$, +, *,)}

Items	Action						Goto
	id	+	*	(	)	$	E
0	s_3			s_2			1
1		s_4	s_5			Accept	
2	s_3			s_2			6
3		r_4	r_4		r_4	r_4	
4	s_3			s_2			7
5	s_3			s_2			8
6		s_4	s_5		s_9		
7		s_4/r_1	s_5/r_1		r_1	r_1	
8		s_4/r_2	s_5/r_2		r_2	r_2	
9		r_3	r_3		r_3	r_3	

Shift / reduce conflict.

Step-3 Parsing the i/p string by using SLR(1) parsing table.

i/p ⇒ id + id * id $ → s_5, r_1 (*), + precedence ∴ go for s_5 [7, *] - s_5

0	id	3	E	1	+	4	id	3	E	7	*	5	id	3	E	8	E	7	E	1	Accept

∴ The row changes without conflict for [7, *] - to s_5.

7		s_4/r_1	s_5		r_1	r_1

i/p ⇒ id * id + id $ → s_4, r_2 +, (*) - precedence [8, +] → r_2

0	id	3	E	1	*	5	id	3	E	8	E	1	+	4	id	3	E	7	E	1	Accept

8		r_2	s_5/r_2		r_2	r_2

i/p : id + id + id $

s_4, r_1 $+, +$ if same go for reduce

$(7, +] \to r_1$

0	id	3	E	1	+	4	id	3	E	7	E	1	+	4	id	3	E	7	E	1	Accept

7	-	r_1	s_5	.	r_1	r_1

i/p : id * id * id $

s_5, r_2 $*, *$

$[8, *] \to r_2$

0	id	3	E	1	*	5	id	3	E	8	E	1	*	5	id	3	E	8	E	1	Accept

8		r_2	r_2		r_2	r_2

Hence ambiguous grammar can be handled.

26/2/22

Ex: $S \to Aa / bAc / dc / bda$ ① ② ③ ④

$A \to d$ ⑤

Sol: Step-1 construction of canonical form of LR(1) items:

I_0: $S' \to .S, \$$

$S \to .Aa, \$$

$S \to .bAc, \$$

$S \to .dc, \$$

$S \to .bda, \$$

$A \to .d, a$

I_1: goto (I_0, S)

$S' \to S., \$$ - final item

I_2: goto (I_0, A)

$S \to A.a, \$$

I_3: goto (I_0, b)

$S \to b.Ac, \$$ $S \to b.da, \$$

$A \to .d, \$$ a

I_4: goto (I_0, d)

$S \to d.c, \$$ $A \to d., \$$ a final item

I_5: goto (I_2, a)

$S \to Aa., \$$ → final item

I_6 : goto (I_3, A)

$S \to bA \cdot c, \$$

I_7 : goto (I_3, d)

$S \to bd \cdot a, \$$

$A \to d \cdot$, ~~c~~ - final item

I_8 : goto (I_4, c)

$S \to dc \cdot, \$$ final item

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

~~$A \to d \cdot a$~~ ~~final item~~

I_9 : goto (I_6, c)

$S \to bAc \cdot, \$$

→ final item

I_{10} : goto (I_7, a)

$S \to bda \cdot, \$$ → final item

	~~I_2 : goto (I_0~~ a	b	c	d	$	S	A
0		S_3		S_4		1	2
1							
2	~~S_5~~			S_7			6
3							
4	r_5				r_1		
5							
6							
7	r_5						
8							
9							
10							
#							

25/2/22

S. Unit-IV

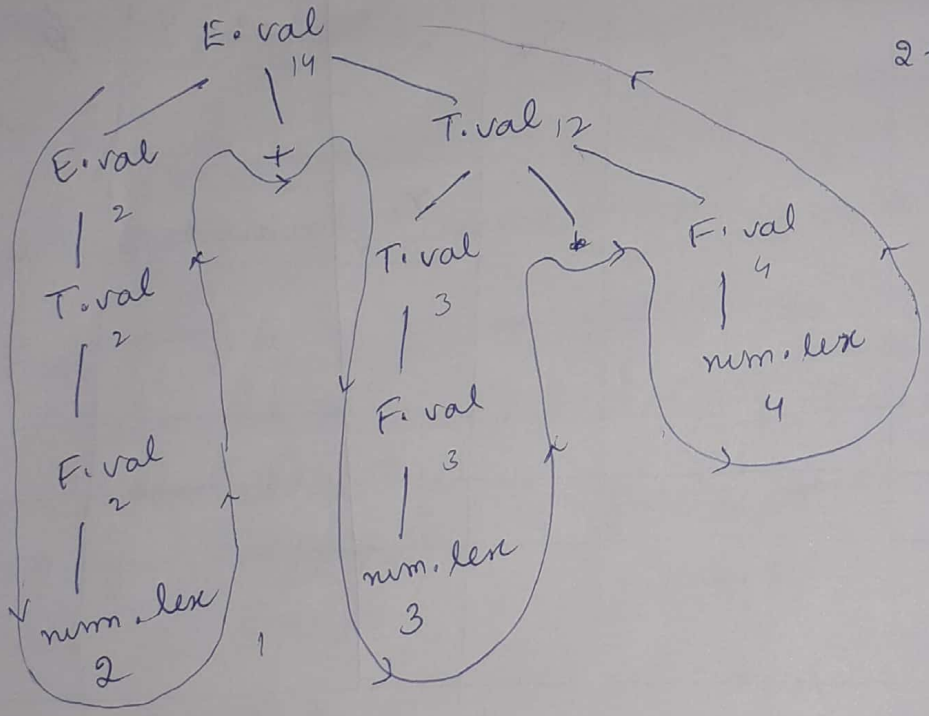
Syntax Directed Translation

- Grammar + Semantic Rules = SDT
- SDT for evaluating an expression

Ex: 2+3*4 = 14 using grammar

E → E+T	E.val → E.val + T.val	Semantic rule for all production
E → T	E.val → T.val	
T → T*F/F	T.val → T.val * F.val	
F → num	T.val → F.val	
	F.val → num.lex	

Sol: Passing from bottom up approach:



Hence the expression 2+3*4 = 14 using SDT.

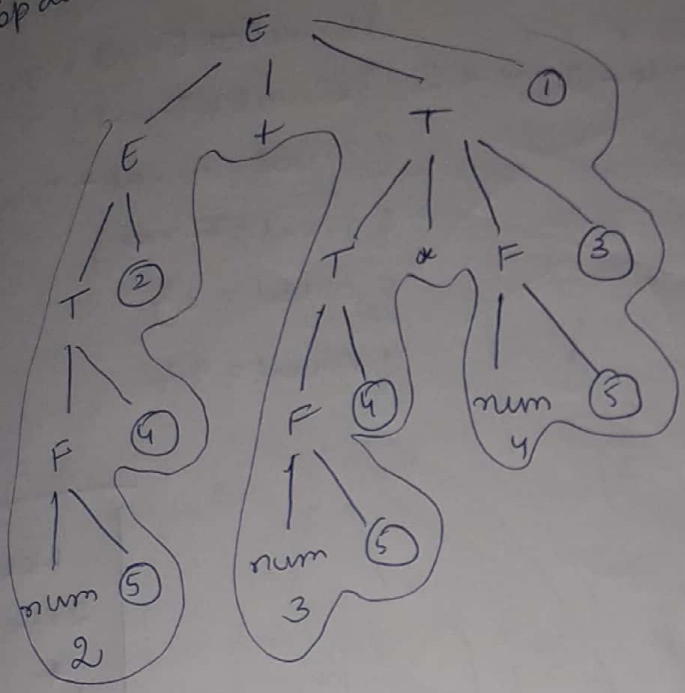
- Converting infix to postfix notation.

infix → 2+3*4

E → E+T {printf("+");} -①
/T {} -②
T → T*F {printf("*");} -③
/F {} -④
F → num {printf("num.lex");} -⑤

Top down parsing

Solu

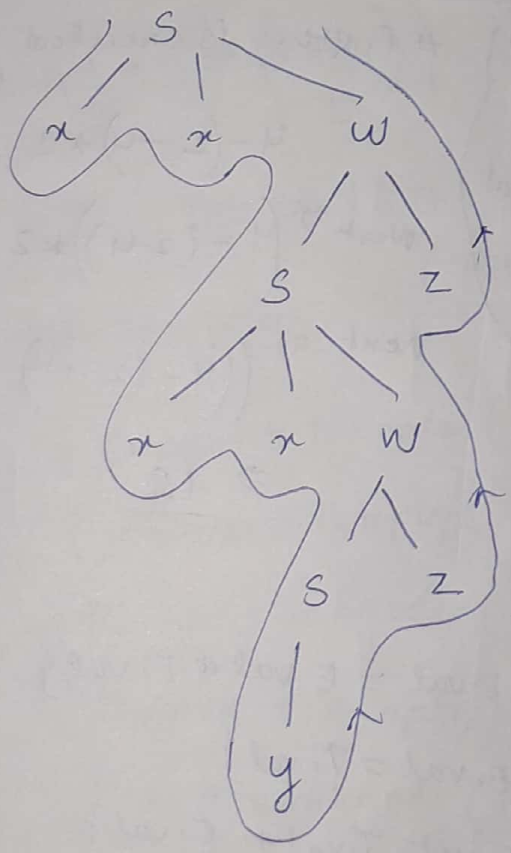


Postfix expression
→
234*+

Ex

S → xxW { printf(1); }
/y { printf(2); }
W → S z { printf(3); }

i/p => xxxxyzz



Reduce ↑ calculate and print.

o/p

23139

3/3/22

Ex

$E \to E * T$ / T

$T \to F - T$ / F

$F \to 2$ / 4

i/p ⇒ 4 − 2 − 4 * 2

{E.val ⇒ E.val * T.val}

{E.val = T.val}

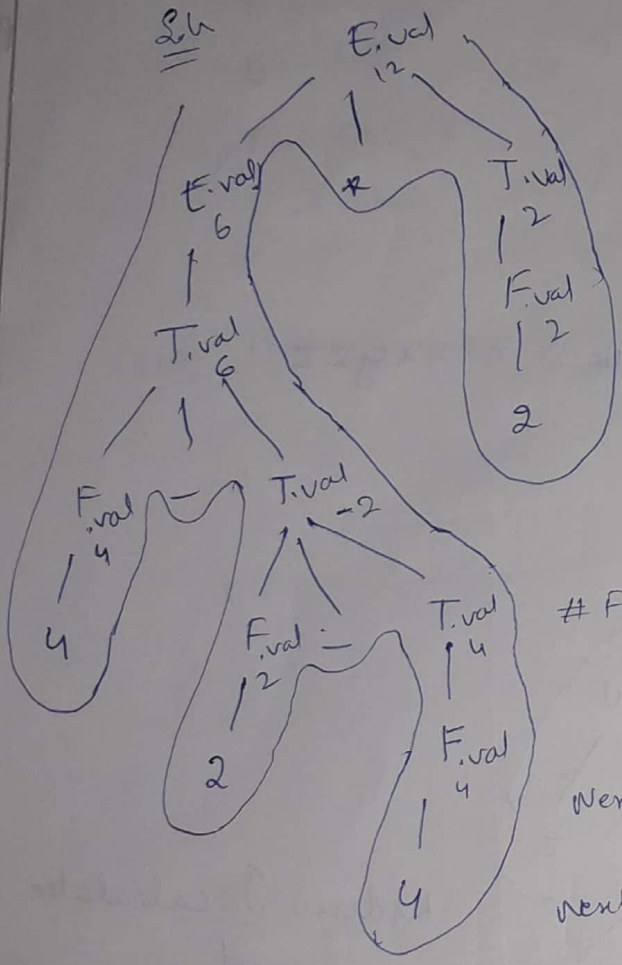
{T.val = F.val − T.val}

{T.val = F.val}

{F.val = 2}

{F.val = 4}

Sol



First this is executed

∴ $4 - (2 - 4) * 2$

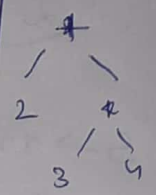
Next ⇒ $(4 - (2-4)) * 2$

next ⇒ $((4 - (2-4)) * 2)$

⇒ 12

Note

2 + 3 * 4



Abstract Syntax tree.

If we use grammar ⇒ then it is parse tree

Ex

$E \to E \# T$ / T

$T \to T \& F$ / F

$F \to num$

{E.val = E.val * T.val}

{E.val = T.val}

{T.val = T.val + F.val}

{T.val = F.val}

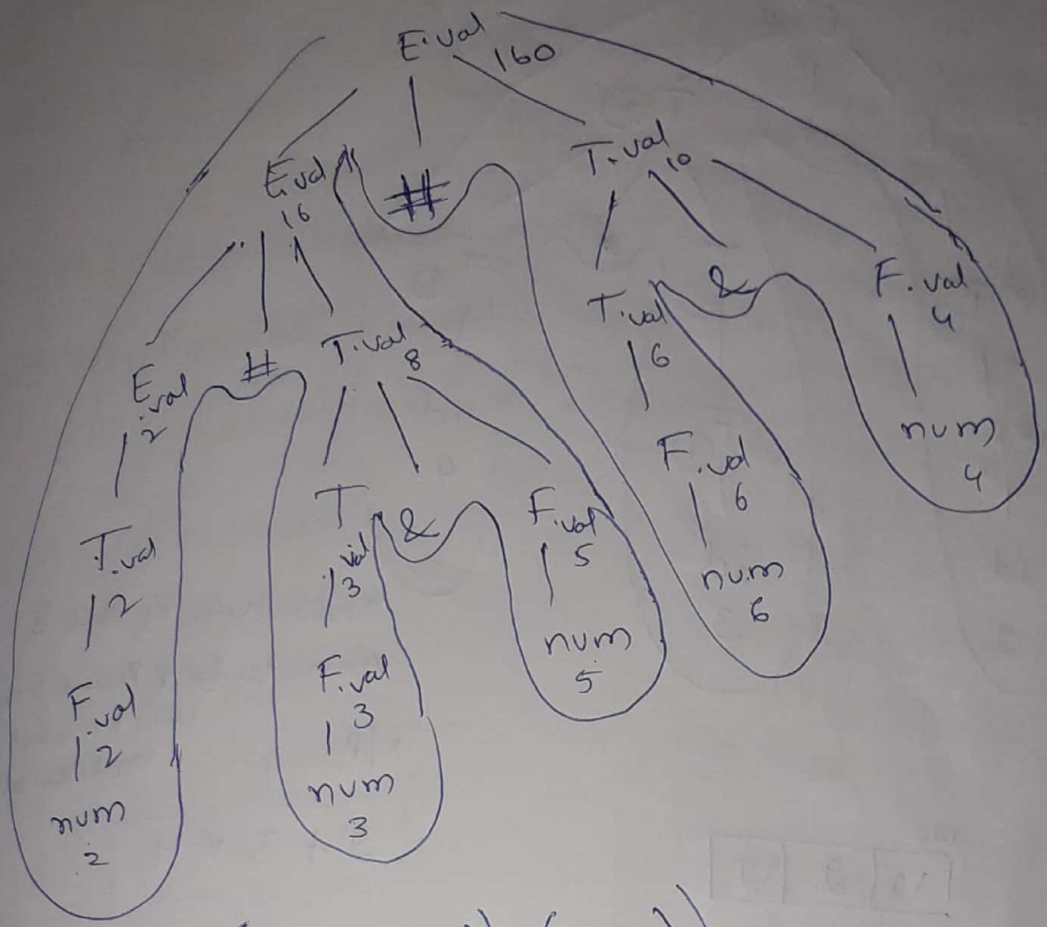
{F.val = num.lexval}

i/p.

2 # 3 & 5 # 6 & 4

Sol: # ⇒ *
& ⇒ +

∴ the i/p string is
2 * 3 + 5 * 6 + 4



$$((2*(3+5))*(6+4))$$

∴ Ans ⇒ 160.

(Till now parse tree)

• SDT to build a syntax tree.

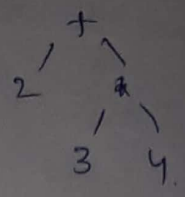
ex:

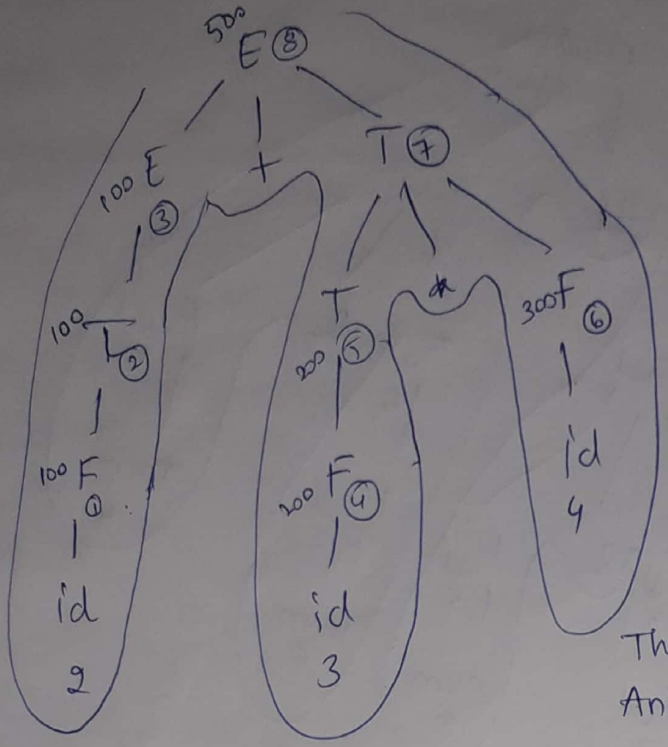
E → E + T	{ E.nptr = mknode (E.nptr, '+', T.nptr); }
/ T	{ E.nptr = T.nptr; }
T → T * F	{ T.nptr = mknode (T.nptr, '*', F.nptr); }
/ F	{ T.nptr = F.nptr; }
F → id	{ F.nptr = mknode (null, idname, null); }

i/p ⇒ 2 + 3 * 4

=> We already know the syntax tree then usually

=> First the parse tree





This type of tree is Annotated Parse tree as it makes nods unlike others

i/p
2 + 3 * 4

① 100 | 10 | 2 | 10 |

② '=' T and F same

③ '=' T and E same

④ 200 | 10 | 3 | 10 |

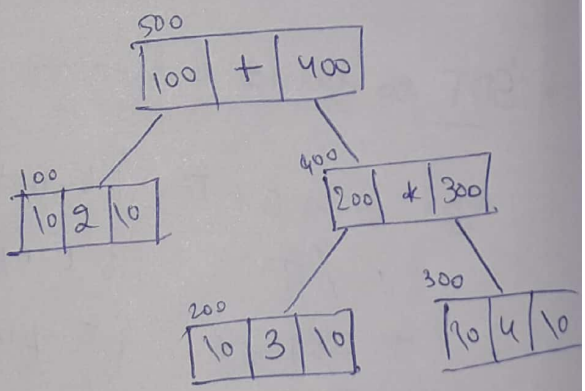
⑤ '=' T and F same.

⑥ 300 | 10 | 4 | 10 |

⑦ 400 | 200 | * | 300 |

⑧ 500 | 100 | + | 400 |

Thus resulting in syntax tree



• Conversion of binary number to decimal number.

N → L
L → LB
  | B
B → 0
  | 1

N → number
L → list of bits
B → bit. 0/1

Ex 1010 (2)

1×2+0
⇒ 2×2+1
⇒ 5×2+0 = 10

Semantic Rules for integer part

{N.dval = L.dval}

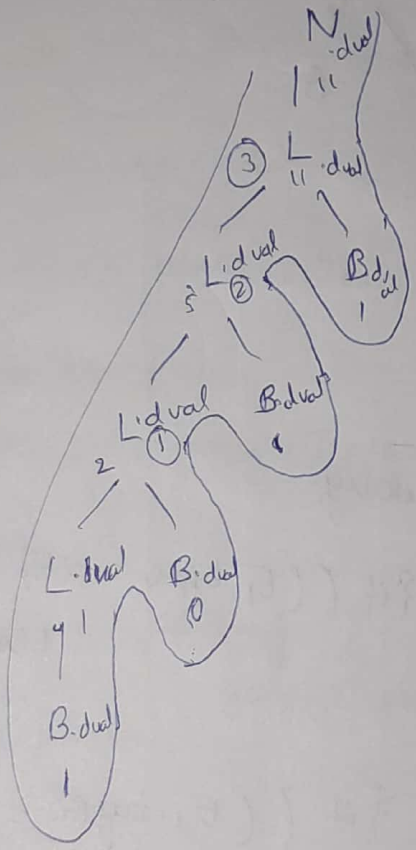
{L.dval = L.dval * 2 + B.dval}

{L.dval = B.dval}

{B.dval = 0}

{B.dval = 1}

i/p - 1011



① 1*2+0
= 2+0 = 2

② 2*2+1
= 4+1 = 5

③ 5*2+1
= 10+1 = 11

∴ o/p ⇒ 11

If decimal

11.01
3 ↙ ↘ 01 → decimal ⇒ $\frac{1}{2^{\text{no. of bits}}} = \frac{1}{2^2} = \frac{1}{4}$
= 0.25

∴ 11.01 ⇒ 3.25

Eg: $N \to LL$ { N.dval = L.dval + ~~L2.dval~~ / L_2.dval ; c }

$L \to LB$ { L.c = L.c + B.c, L.dval = L.dval + B.dval }

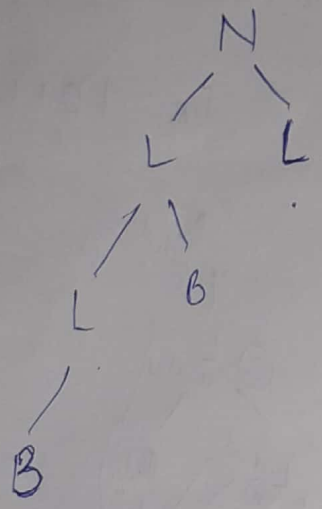
/B { L.c = B.c, L.dval = B.dval }

$B \to 0$ { B.c = 1, B.dval = 0 }

/1 { B.c = 1, B.dval = 1 }

(c: count; dval: what value)

i/p ⇒ 11.01



- SDT for type checking.

Eg: $E \to E+E$ { if ((E1.type == E2.type) && (E1.type = int)) then E.type = int else error; }

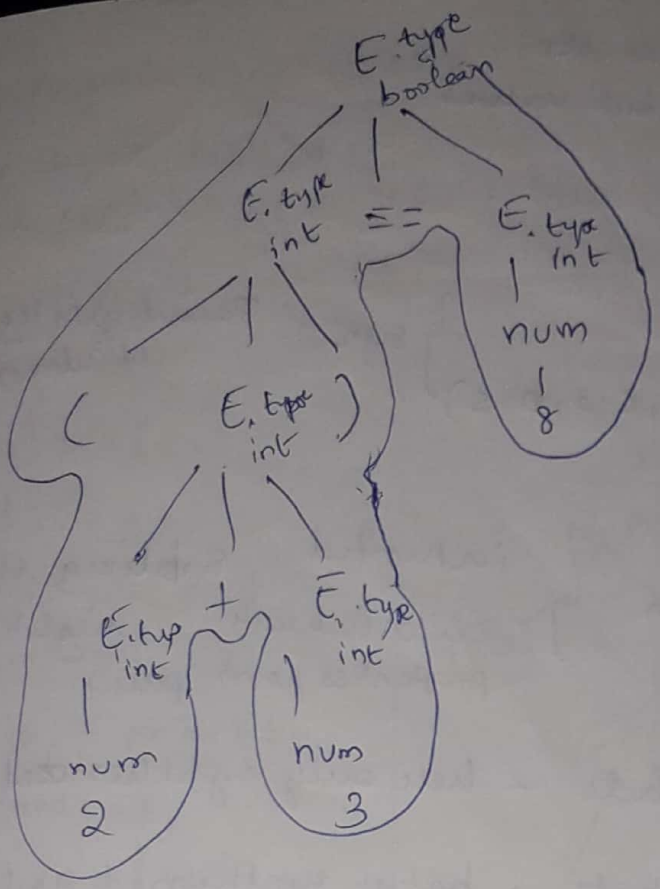
/ $E_1 == E_2$ { if ((E1.type == E2.type) && (Ep.type = int/boolean)) then E.type = boolean else error; }

/ (E_1) { E.type = E1.type; }

i/p ⇒ (2+3) == 8

/num { E.type = int; }

/TRUE { E.type = bool; } /FALSE { E.type = bool; }



4/3/2022

- SDT to generate the Three address code

i/p
x = a + b * c

$S \to id = E$ { gen(id.name = E.place); }

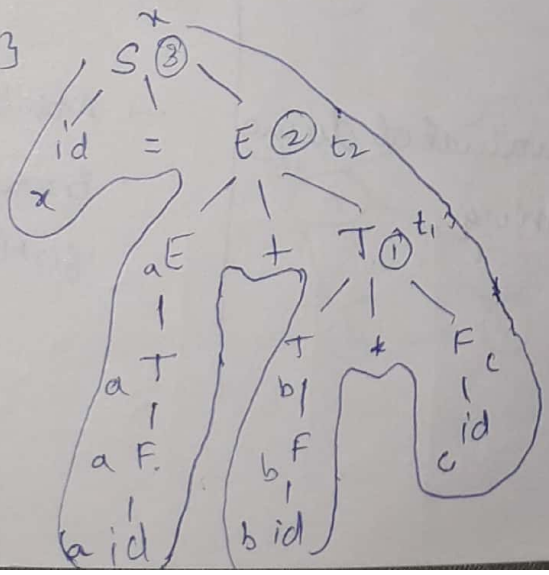
$E \to E + T$ { E.place = new temp();
gen (E.place = E.place + T.place); }

/ T { E.place = T.place; }

$T \to T * F$ { T.place = new temp(); }
gen (T.place = T.place * F.place); }

/ F { T.place = F.place; }

$F \to id$ { F.place = id.name }



① t_1 temp var
$t_1 = b * c$

② t_2 temp var
$t_2 = a + t_1$

③ $x = t_2$

- Attribute => Ex: E-place etc..
  => nothing but values
  2 types
  Synthesized Inherited

Ex: A → BCD
A.S → F(B.S, C.S, D.S) } syn. - Parent gets the value from children

B.n = A.n
C.n = B.in
D.in = C.in
} inherited sibling should be child inherits right hand side. properties from parent.

- SDT
  - S-attribute - here only synthesized attribute
  - L attribute → both synthesized and inherited

S attri SDT

→ It uses only synthesized attribute

Ex: ↑

→ Semantics actions are placed at the right hand of the production

→ Attributes are evaluated during bottom up passing.

L attri SDT

→ It uses both inherited and synthesized attribute. Each inherited attri is restricted to inherit either from parent or left siblings only.

Ex: ↑

→ Semantic actions are placed anywhere.

→ Attributes are evaluated by traversing the parse tree depth first, left to right.

Directed Acyclic Graph (DAG)

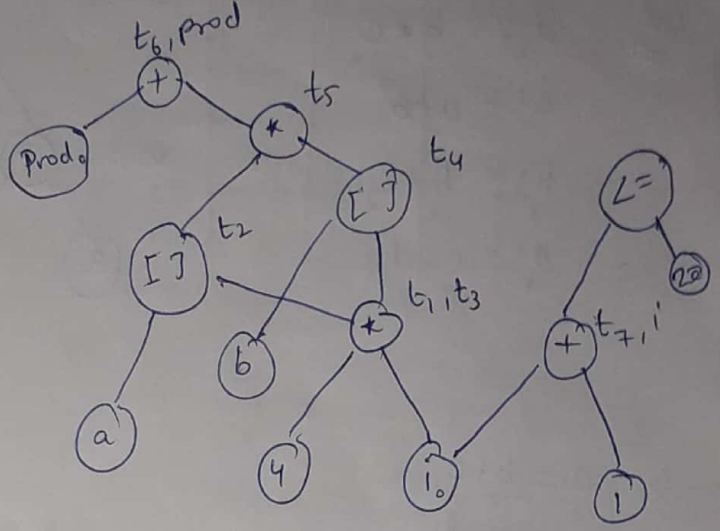
- → used basic blocks
- → block contains set of instructions

Block

$t_1 := 4 * i$
$t_2 := a[t_1]$
$t_3 := 4 * i$
$t_4 := b[t_3]$
$t_5 := t_2 * t_4$
$t_6 := prod + t_5$
$prod := t_6$
$t_7 := i + 1$
$i := t_7$ if $i <= 20$ goto (1)

(1) ↓ step 1

DAG for this Block.

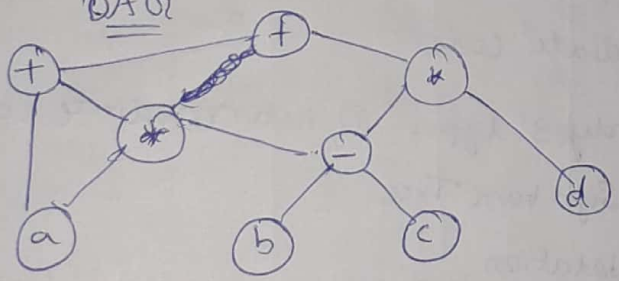


a + b + c

In syntax tree, we use 2 '+'. Here no need we can reuse the same

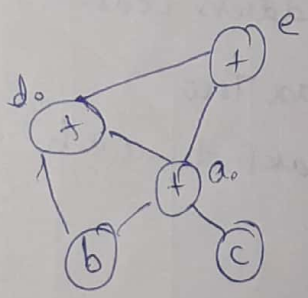
Ex — $(a + a * (b - c)) + ((b - c) * d)$

DAG



Ex =

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

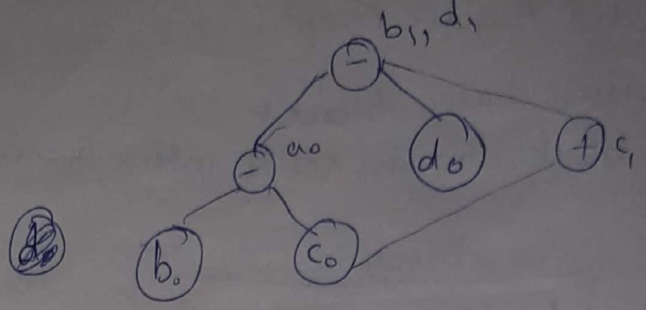


eg $a_0 := b_0 - c_0$

$b_1 := a_0 - d_0$

$c_1 := b_1 + c_0$

$d_1 := a_0 - d_0$

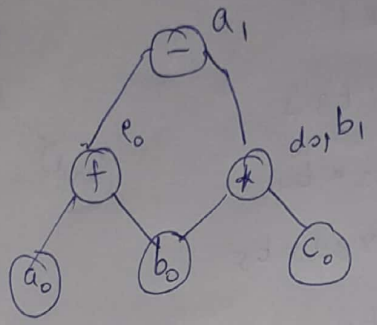


eg $d_0 := b_0 * c_0$

$e_0 := a_0 + b_0$

$b_1 := b_0 * c_0$

$a_1 := e_0 - d_0$

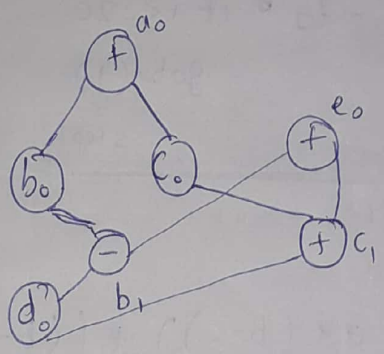


eg $a_0 := b_0 + c_0$

$b_1 := b_0 - d_0$

$c_1 := c_0 + d_0$

$e_0 := b_1 + c_1$



9/3/22

- Forms of intermediate codes.

- There are mainly 3 types of intermediate code representation.

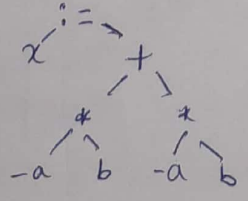
  i) Abstract Syntax Tree

  ii) Polish Notation

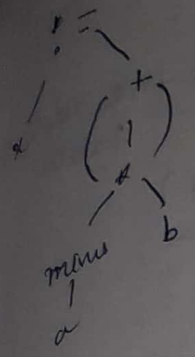
  iii) Three address code.

i). Abstract Syntax Tree

$x := -a * b + -a * b$



DAG



ii) Polish Notation (Prefix)

ex (a+b)(c+d)

* +ab+cd

iii) Three address code

ex $a := b+c+d$

$t_1 := b+c$

$t_2 := t_1+d$

$a := t_2$

Implementation of three address code - 3 ways -

i) Quadruple

ii) Triple

iii) indirect triple.

i) Quadruple - 4 columns

ex $x := -a * b + -a * b$

Num	op (operator) ①	Arg1 (Arguments) ②	Arg2 ③	Result ④
(0)	uminus	a	-	t_1
(1)	*	t_1	b	t_2
(2)	uminus	a	-	t_3
(3)	*	t_3	b	t_4
(4)	+	t_2	t_4	t_5
(5)	:=	t_5	-	x

$t_1 := -a$

$t_2 := t_1 * b$

$t_3 := -a$

$t_4 := t_3 * b$

$t_5 := t_2 + t_4$

$x := t_5$

uminus -> unary minus

ii) Triple -> No result column / use num (0), (1) ....

Num	op	Arg1	Arg2
(0)	uminus	a	-
(1)	*	(0)	b
(2)	uminus	a	-
(3)	*	(2)	b
(4)	+	(1)	(3)
(5)	:=	x	(4)

iii) Indirect Triple -

Write the Triple table + along with that write the address table.

Num	op	Arg1	Arg2
(0)	uminus	a	-
(1)	*	1000	b
(2)	uminus	a	-
(3)	*	1200	b
(4)	+	1100	1300
(5)	:=	x	1400

Num	Addr (start)
(0)	1000
(1)	1100
(2)	1200
(3)	1300
(4)	1400
(5)	1500

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

i) Quadruple

Num	Op	Arg1	Arg2	Result	
(0)	uminus	c	-	t_1	$t_1 := -c$
(1)	*	b	t_1	t_2	$t_2 := b * t_1$
(2)	uminus	c	-	t_3	$t_3 := -c$
(3)	*	b	t_3	t_4	$t_4 := b * t_3$
(4)	+	t_1	t_4	t_5	$t_5 := t_1 + t_4$
(5)	:=	t_5	-	a	$a := t_5$

ii) Triple

Num	Op	Arg1	Arg2
(0)	uminus	c	-
(1)	*	b	(0)
(2)	uminus	c	-
(3)	*	b	(2)
(4)	+	(1)	(3)
5	:=	a	(4)

(ii) Indirect Triple

Num	Op	Arg1	Arg2
(0)	uminus	c	-
(1)	*	b	1000
(2)	uminus	c	-
(3)	*	b	1200
(4)	+	1100	1300
(5)	:=	a	1400

Num	Addr(start)
(0)	6000
(1)	1100
(2)	1200
(3)	1300
(4)	1400
(5)	1500

11/3/22

- Contents of Symbol Table. (Already an example is discussed in Unit 1)
  - i) variablesname
  - ii) constants
  - iii) Datatypes
  - iv) Compiler generated temperories.
  - v) function names
  - vi) parameter names
  - vii) Scope information

i) ⇒ when a variable is identified, it is stored in symbol table by its name.

ii) ⇒ These constants can be accessed by the compiler with the help of pointers.

iii) ⇒ The datatypes of associated variables is stored in the symbol table.

iv) ⇒ During this process, the temperories get generated which are stored in the symbol table.

v) ⇒ function names, ~~the names~~ can be stored in the symbol table.

vi) ⇒ ~~paran~~ the information such as call by value or call by reference is stored in the symbol table.

vii) ⇒ Scope of a variable, where it can be used, such type of information can be stored in the symbol table.

- Data Structures in the Symbol table :-
  - i) List Data Structure in the symbol table
  - ii) Self organising list
  - iii) Binary trees
  - iv) Hash tables

i) Linear list:

Name 1	info 1
Name 2	info 2
Name 3	info 3
⋮	⋮
Name n	info n

→ It is the simplest kind of mechanism to implement the symbol table.

→ In this method, an array is used to store the names and associated information.

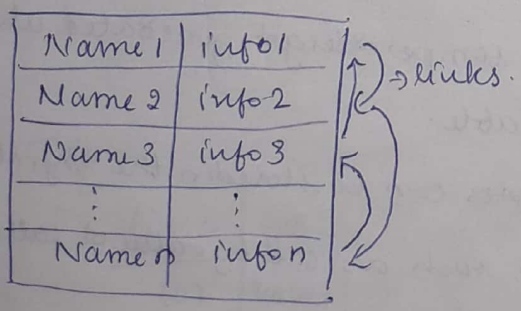
→ New names can be added in the order as they arrive.

→ The pointer available is maintained at the end of all stored records.

ii) → A link field is added to each record.

→ We search the records in the order pointed by the link of link field

→ A pointer 'first' is maintained to point to first record of the symbol.



→ The reference to these names can be
Name 2, Name n, Name 3, Name 1

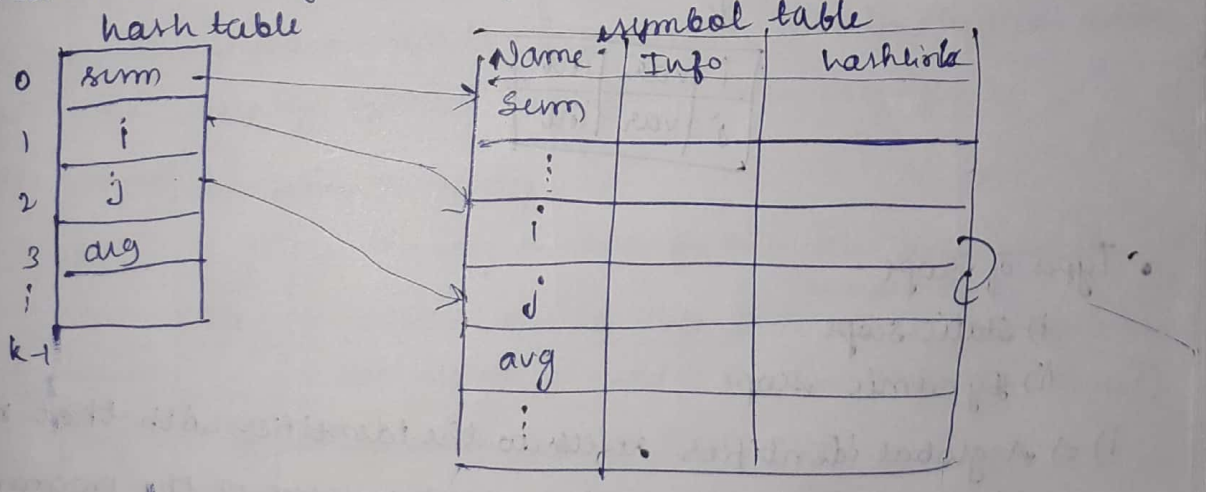
→ The most frequently referred names will be moved to the front of the list.

iii) → The node structure will be as follows:

left child	symbol (sym)	info	Right child

→ The left child field stores the address of the previous symbol.
→ The right child field stores the address of the next symbol.
→ The symbol field is used to store the name of the symbol.
→ The information field is used to give the information about the symbol.
→ The binary tree structure is basically a binary search tree in which the value of left node is allways less than the value of parent node. Similarly, the value of right node is always greater than the values of parent node.

iv) → It is an important technique used to search the records of the symbol table.
→ In hashing scheme, two tables are maintained: Hash table and Symbol table.
→ Hash table consists of k-entries from 0 to k-1.
→ These entries are basically pointers to symbol table pointing to the names of the same table.
→ To determine whether the name is in symbol table, we use hash function h, such that h(name) will result any integer between 0 to ~~k-th~~ k-1.
→ we can search any name by position = h(name).



24/3/22

- Representing Scope Information

- Scope rules in a block-structured ({ }) language is as follows -

i) A name declared within a block B is valid only within B.

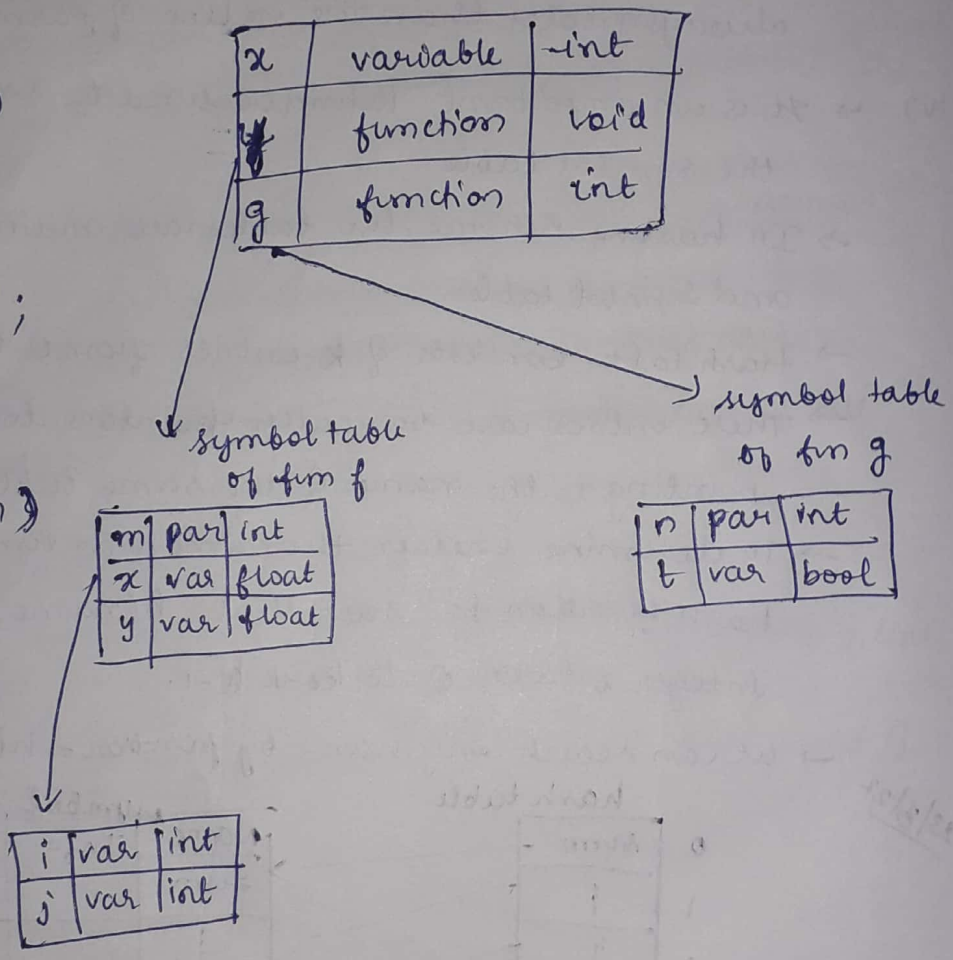
ii) If Block B1 is nested within B2 then any name that is valid for B2 is also valid for B1 unless the identifier for that name is redeclared in B1.

iii) Tables are organised in stack and each table contains the list of name and their associated attributes.

iv) whenever a new block is entered onto the stack, the new table holds the name i.e declared as local to this block.

Ex

```
int x;
void F (int m)
{
  float z, y;
    {
      int i, j;
    }
}
int g (int n)
{
  bool t;
}
```



- Types of scope -

i) Static scope

ii) Dynamic scope

i) => A global identifier refers to the identifier with that name i.e declared in the closest enclosing scope of the program ~~con~~ text

=> It uses the static relationship between blocks in the program.

ii) => A global identifier refers to the identifier associated with the most recent activation record.

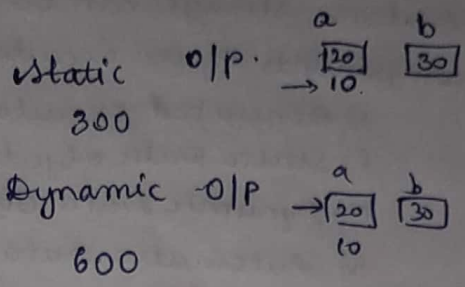
=> It uses actual sequence of calls that are executed in the dynamic scope.

Ex

```
int a=10;
fun1 (int b)
{
printf("%d", a*b);
}
fun2 ()
{ int a=20;
 fun1 (30);
}
main()
{
 fun2();
}
```

Static O/P: 300

Dynamic O/P: 600



5/12/22

• Runtime Environment

⇒ Storage Organisation

- The executing target program runs in its own logical address space in which each program value as a location.
- The management and organisation of this logical address space is shared b/w the compiler, operating system and target machine.
- The operating system maps the logical addr into the physical addr which are usually spread throughout the memory.

→ Subdivision of Runtime Memory

code	→ memory location for this code determined at compile time
static data	→ location of static data is determined at compile time
stack	→ data objects allowed at Runtime (Activation Record)
free memory	→ remaining space of empty area
Heap	→ The dynamically allocated data objects at runtime e.g. malloc.

→ Runtime storage comes into blocks where byte is used to show the smallest unit of addressable memory.
→ Using the four (4) bytes, a machine word can form.
→ Runtime storage can be subdivided to hold the different components of an executing program.
  i) Genrated executable code
  ii) Static Data objects
  iii) Dynamic Data Objects
  iv) Automatic Data objects.

⇒ Stack Allocation of Space.
  1 → Activation Trees
  2 → Activation Record
  3 → Calling sequences
  4 → Variable length data on the stack.

- Procedure calls and returns can usually managed by a run-time stack called the control stack.
- Each live activation has an activation record on the control stack.
- The root of all activation records will be at the bottom of the control stack.
- The last activation record will be found at the top of the stack.

1) ⇒ Represents the sequence and relation b/w caller and calling function / procedure.

2) ⇒ Record has the following fields. (7)

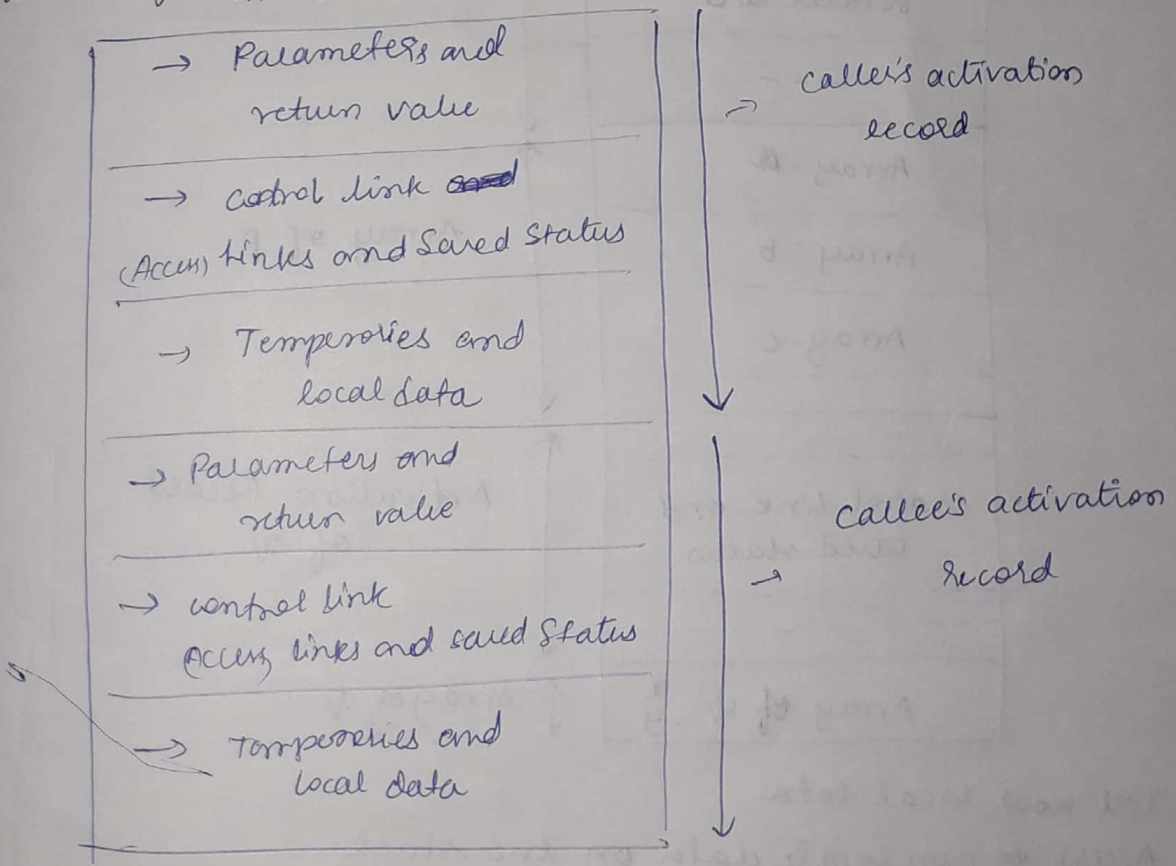
i)	Actual Parametrs
ii)	Returned Values
iii)	Control link
iv)	Access link
v)	Saved machine Status
vi)	Local Data
vii)	Temporaries

i) ⇒ used by the calling procedures, but usually they will get saved & stored on the CPU registers

ii) ⇒ space for return value of the called function, it is also not mandatory the returned value can get stored on the CPU registers.

ii) ⇒ Pointing to the activation record of the caller.
iv) ⇒ It may be needed to locate the data needed by the called procedure but found elsewhere i.e in another activation record.
v) ⇒ Saved machine status with information about the state of the machine just before the call to the procedure.
vi) ⇒ belonging to the procedure whose activation record it is.
vii) ⇒ such as those arising from the evaluation of the expression, those temporaries cannot be held in registers.

3) ⇒ Procedure calls are implemented by what are known as calling sequences which consists of code that allocates an activation record on the stack and enters information into its fields. A return sequence is similar code to restore the state of the machine so the calling procedure can continue its execution after the call.
- The code in a calling sequence is often divided b/w the calling procedure and the procedure it calls.
- There is no exact division of runtime task b/w the caller and the callee. The source language, target machine and operating systems will impose the requirements.



31/3/22

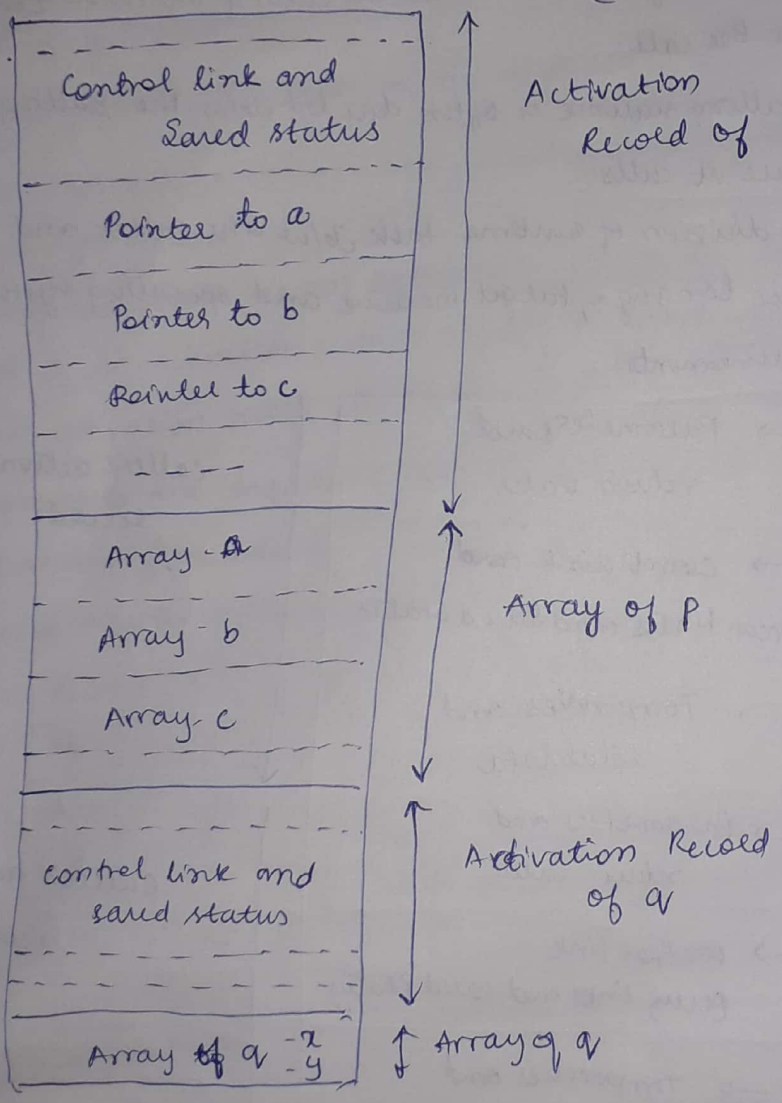
4) Variable length data on the stack:

The runtime memory management system must deal frequently with the allocation systems of objects and the sizes of which are not known at compile time but which are local to a procedure and thus may be allocated in the stack.

→ The scheme works for objects of any type if they are local to the procedure called and have a size that depends on parameters of the call.

→ A common strategy for allocating the variable length array is shown below:-

```
p(int a, int b)
{
  q();
}
q(int x, int y)
{
}
```



Till now local data

- Access to non local data on the stack:
  - mechanism for finding data used within the procedure but does not belong to P.

```
P() {
  q() { }
}
```

- Data access without nested procedures.
  i) Variables are defined either within a single function or outside any function
  ii) A global variable of scope consisting all functions.
  iii) Global variables are allocated static storage.
  iv) Other name i.e local to the activation at the top of the stack can be accessed through top_sp pointer on the stack.

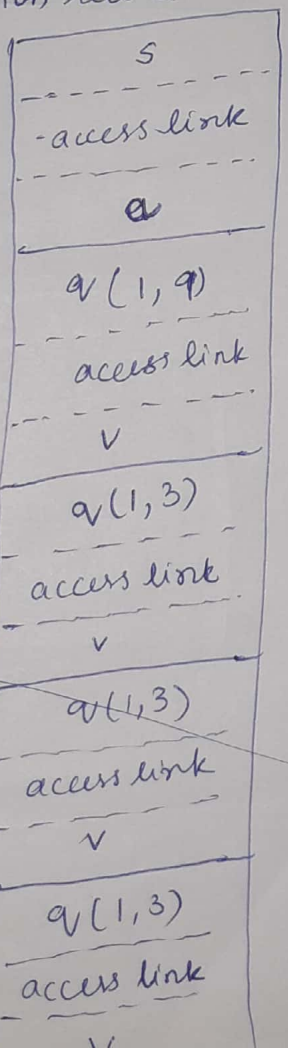
- Nesting Depth
  i) For procedures without nested within other procedure the nesting depth is 1.

```
1 - P() {
      q() { }
    }
```

```
q() {
  }
```

  ii) the nesting depth for the nested procedure is i+1.

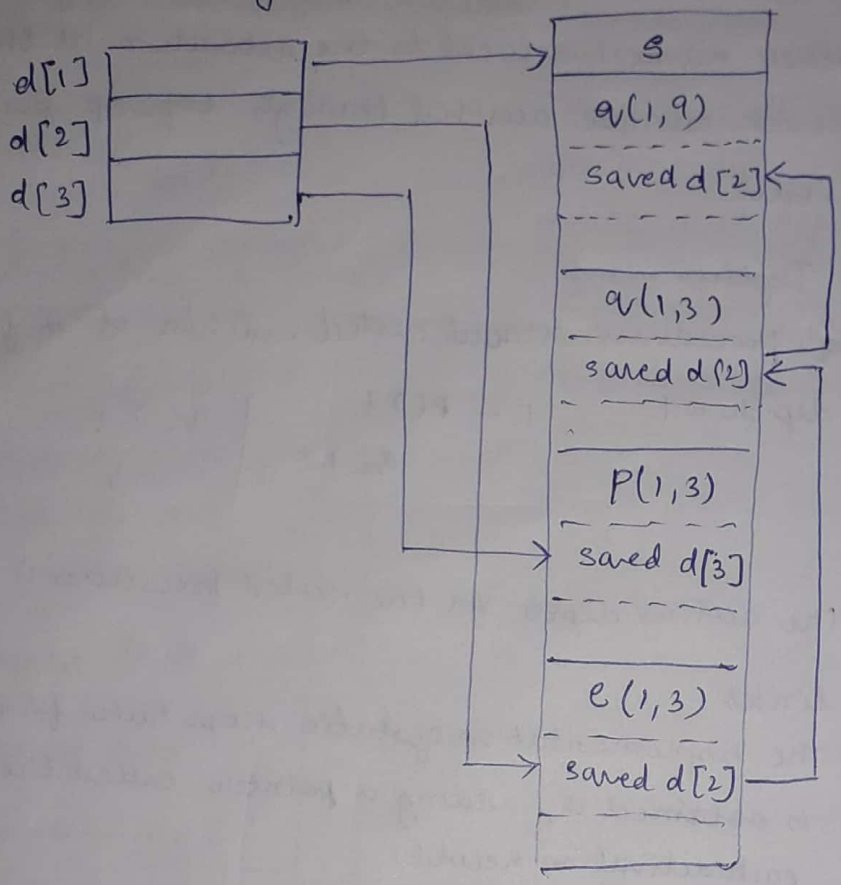
- Access links
  i) The implementation of static scope rule for nested functions is obtained by adding a pointer called the access link to each activation record.



- Display

i) If the nesting depth gets large then we have to follow long chain of links to reach the data.

ii) An auxilary array d consisting of one pointer for each nesting depth. ~~will be~~



Unit-5

Code Optimization

Optimization can be classified -
- machine independent
- machine Dependent

Machine independent Optimization
- are program transformations that improve the target code without taking into consideration any properties of target machine.

Machine Dependent Optimization
- this is based on register-allocation and utilization of special machine instruction sequence.

Code optimization techniques - (Principal Sources of Optimization)
1→ Common sub expression elimination
2→ Constant Folding
3→ Copy Propagation
4→ Dead code Elimination
5→ Code Motion
6→ Induction variable Elimination and Reduction in strength

Common sub expression elimination
- If it was previously computed, then values of variables have not changed.

Ex→

basic block.
a := b+c
b := a-d
c := b+c
d := a-d

⇒

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

// here it is not same as a as b value changes in 2nd step. ∴ only d part is changed

2) Constant Folding

- If the value of an expression is constant then use the constant instead of expression.

ex $PI = \frac{22}{7} \Rightarrow PI = 3.14$ // Instead of repeated division use 3.14.

3) Copy Propagation

- $f = g$

Use g for f after $f = g$.

```
x := a          x = a
y := x * b  ⇒   y = a * b
z := x * c      z = a * c
```

4) Dead code Elimination

- A variable is alive if its value can be used subsequently, otherwise it is dead.

main code ⇒

```
x = a           y = a * b
y = a * b  ⇒    z = a * c
z = a * c
```

x is dead.

5) Code Motion

- moves the code outside the loop.

```
while (i<10)          x := y+z;
{                ⇒    while(i<10)
x := y+z;             {
i := i+1;             i := i+1;
}                     }
```

6) Induction Variable Elimination and Reduction in Strength

```
i=1;                  t=4;
while (i<10) {   ⇒    while (t<40) {
t := i*4;             t := t+4;
i := i+1; }           }
```

Loop Optimization

- most execution time of a program is spent on loops.
- decreasing the no. of instructions in an inner loops improves the running time of a program.

Loop Optimization Techniques -

i) Code motion
ii) Induction variable Elimination and Reduction in Strengths.
iii) Loop unrolling
iv) Loop Jamming

i) ⇒ Code motion
- moves the code outside the loop.

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

⇒

```
t = limit/2;
while (i<100)
{
 a = t + i ;
 i++;
}
```

ii) ⇒ Induction Variable Elimination and Reduction in Strength

```
i=1
while (i<10)
{
 y = i*4;
 i = i+1;
}
```

⇒

```
t = 4;
while ( t<40)
{
 y = t;
 t = t+4;
}
```

iii) Loop Unrolling
- Duplicates the body of the loop multiple times inorder to decrease the no. of times the loop condition is tested.

```
for (i=0; i<100; i++)
{
 display ();
}
```

⇒

```
for(i=0 ; i<50; i++)
 {
  display();
 display();
}
```

Loop Jamming
- It combines the bodies of 2 adjacent loops that would iterate the same no of times.

```
for(i=0;i<100;i++)
{ a[i]=1;
for(i=0;i<100;i++)
  b[i]=2;
```

⇒

```
for(i=0;i<100;i++)
{
  a[i]=1;
  b[i]=2;
}
```

9/4/22

- Redundant Sub Expression Elimination
  - an occurrence of an expression E, is called redundant sub expression if E was previously computed and the values of the variables in E ~~and t~~ have not changed since the previous computation

Ex

```
a=x+y+z          t=x+y
r=p+q      ⇒     a=t+z
b=x+y+r          r=p+q
                 b=t+r
```

"x+y" is a redundant subexpression since "x+y" is already computed in "a=x+y+z", and values of x, y have not changed after the first computation and it is same in "b=x+y+r".

- It is a compiler optimization that searches for instances for identical expressions and analyzer whether it is worthwhile replacing them with a single variable holding the computed value.

Ex

B5

```
t6 = 4*i
x = a[t6]
t7 = 4*i
t8 = 4*j
t9 = a[t8]
a[t7] = t9
t10 = 4*j
a[t10] = x
goto B2
```

Before elimination

⇒

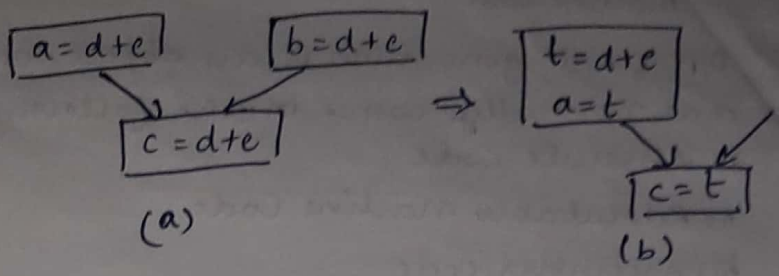
B5

```
t6 = 4*i
x = a[t6]
t8 = 4*j
t9 = a[t8]
a[t6] = t9
a[t8] = x
goto B2
```

After elimination

- Copy Propagation

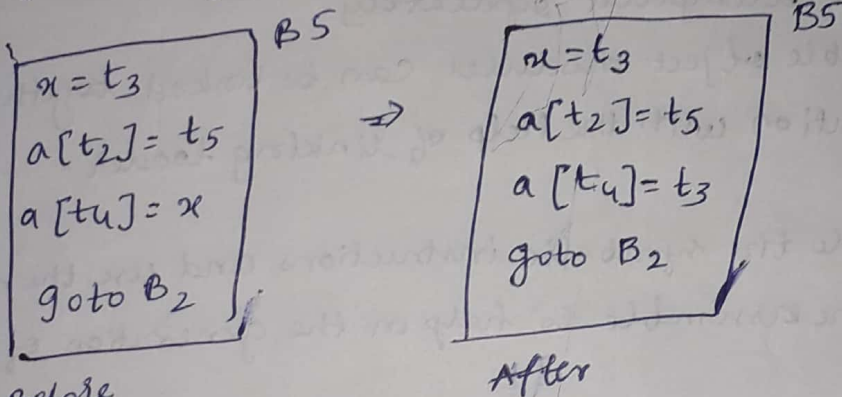
In order to eliminate the common subexpression from the statement "c=d+e", in the Fig(a), we must use a new variable 't' to hold the value of "d+e". The value of variable 't' instead of that of the expression "d+e" is assigned to "c" in the figure (b).



(a) (b)

Since the control may reach "c=d+e", either after the assignment to 'a' (or) assignment to 'b' and it would be incorrect to replace "c=d+e" by either "c=a" or "c=b".

Ex:



Before After

- Dead Code Elimination

- It removes unneeded instructions from the program.
- Dead code is a section in the source code of a program which is executed but whose results are never used in any other computation.
- It prevents from wastage of computation, time and memory.

```
z = x*y          z = x*y
a = x      ⇒     w = x*y+6
w = x*y + 6
```

- Object Code / Machine Code Forms
  - The O/P of code generation is an object code / machine code.
  - This code normally comes in the following forms -
    i) Absolute code
    ii) Relocatable Machine Code
    iii) Assembler code

=> Absolute Code
- Is a machine code that contains reference to actual addresses within the programs address space.
- The generated code can be placed directly in the memory and execution starts immediately.

=> Relocatable Machine code
- Producing a relocatable m/c language program as O/P allows sub programs to be compiled separately.
- A set of relocatable object modules can be linked together and loaded for execution with the help of linking loader.

=> Assembler Code
- we can generate the symbolic instructions and use the macro-facilities of the assemble to help in the generation of the code.

11/4/22

- Issues in Design of a Code Generator
  1) I/P to the code generator
  2) Target Program
  3) memory management
  4) Instruction selection
  5) Register Allocation
  6) Choice of Evaluation order

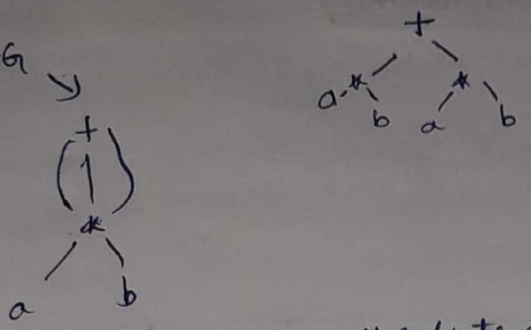
1) => I/P should be in the form of -
   i) linear representation
   ii) Three addr representation
   iii) Graphical representation.

i) => postfix notation Ex a*b => ab*

ii) ⇒ again — quadruples
— triples
— indirect triples

iii) ⇒ Syntax tree → ex ≡ a*b+a*b

⇒ DAG



2) ⇒ o/p to the code generator — Absolute code
— Relocatable Machine code
— Assembler code

3) ⇒ mapping names in the source program to runtime memory.

4) ⇒ ex a:=b+c

```
MOV  b, R0
ADD  c, R0
MOV  R0, a
```

R0 [b] ⇒ R0 [b+c]

a [R0]

ex x:=y+z
w:=x+s

```
MOV  y, R0
ADD  z, R0
MOV  R0, x
MOV  x, R0
ADD  s, R0
MOV  R0, w
```

to use ↩ { MOV R0, x } { MOV x, R0 }

After instruction selection

```
MOV  y, R0
ADD  z, R0
ADD  s, R0
MOV  R0, w
```

ex x:=x+1

```
MOV  x, R0
ADD  #1, R0
MOV  R0, x
```

} instead directly INC x (default increment)

5) ⇒ — Register Allocation ⇒ For all variables are stored in register or not.
— Register Assignment ⇒ Each variable has its register assigned or not.

ex t:=x+y
t:=t*z
t:=t/w

```
MOV  x, R0
ADD  y, R0
MUL  z, R0
DIV  w, R0
MOV  R0, t
```

6) ⇒ Some computation order prefers the less registers.!

25/4/22 • A machine model | Target m/c model

→ The target m/c model is a sequence of 3-adde statement.

Opcode	source	Destination

↓
MOV
ADD
MUL
SUB

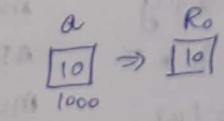
• Addressing modes

Addressing mode	Form	Address	Added cost
i) Absolute	Memory	M	1
ii) Register	Register	R	0
iii) Indexed	C(R) contents of Register R	C+contents(R)	1
iv) Indirect register	*R	Contents (R)	0
v) Indirect indexed	* C(R)	contents (C+ contents(R))	1
vi) Immediate/ Literal	#C	NA	1

i) Absolute

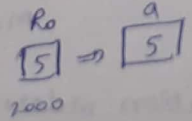
Instructions

1) mov a, R_0 contents (a) => R_0



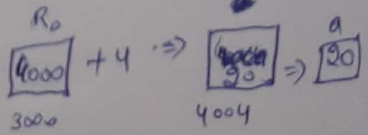
ii) Register

1) mov R_0, a contents(R_0) => a



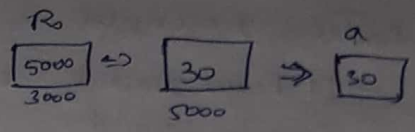
iii) Indexed

1) mov 4(R_0), a contents(4+ contents (R_0)) => a



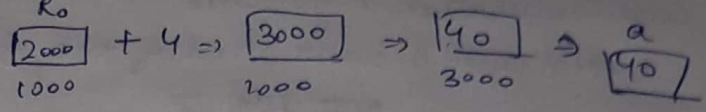
iv) Indirect Register

i) MOV *R0, a contents(R0) ⇒ a



v) Indirect Indexed

i) MOV *4(R0), a contents(contents(~~contents~~(4+contents(R0)))) ⇒ a



vi) Immediate

i) MOV #4, R0

- How to find the instruction cost?

(1+ cost of source and destination addressing mode)

Ex: MOV R0, R1

Instruction cost = 1+0+0

= 1

Ex: MOV R0, M

cost = 1+0+1

= 2

Ex: ADD #1, R0

cost = 1+1+0

= 2

Ex: MOV b, a = 1+1+1 = 3

~~ADD~~ ADD c, a = 1+1+1 = ~~3~~

6

Ex: SUB 2(R0), *8(R1)

cost = 1+1+1 = 3

Ex: Let a := b+c

MOV b, R0 ⇒ 1+1+0 = 2

ADD c, R0 ⇒ 1+1+0 = 2

MOV R0, a ⇒ 1+0+1 = 2

6

MOV *R1, *R0 ⇒ 1+0+0 = 1

ADD *R2, *R0 ⇒ 1+0+0 = 1

2

- A Simple Code Generator

→ It generates a target code ~~with~~ for a sequence of 3 addr ~~stmt~~.

→ For each operator in the stmt, there is a corresponding target language operator.

→ A ~~single~~ simple code generator contains register and address descriptors.

i) Register Descriptor -

→ It keeps track of what is currently in each register

→ Initially all registers are empty.

ii) Address Descriptor-

→ It keeps track of the location, where the current value of the name can be found.

→ That location may a register, stack location or memory address.

- Code Generation Algorithm

→ For each 3-addr stmt of the form,

x := y op z

i) Invoke a function getreg() to determine the location L, where the result of y op z should be stored.

getreg() → may be an empty register / occupied register / memory.

ii) consult address descriptor for y to determine the y', the (y → may be register / memory) current location of y. If y is not already in L, then generate MOV y', L.

Mov y, R_0

iii) Generate the instruction op z', L : Update the address descriptor of x to indicate that x is in L. If L is a register, update its descriptor to indicate that it contains the value of x.

ADD z, R_0

iv) If y and z have no next uses and not live on exit, update the descriptors to remove the y and z.

MOV R_0, x.

Ex: d = (a-b) + (a-c) + (a-c)

$t_1 = a - b$

$t_2 = a - c$

$t_3 = t_1 + t_2$

$d = t_3 + t_2$

Statements	Code generator	Register Descriptor	Address Descriptor
$t_1 = a - b$	MOV a, R_0 SUB b, R_0	Registers are empty and R_0 contains t_1	t_1 in R_0.
$t_2 = a - c$	MOV a, R_1 SUB c, R_1	R_0 contains t_1 R_1 contains t_2	t_1 in R_0. t_2 in R_1.
$t_3 = t_1 + t_2$	ADD R_1, R_0	R_0 contains t_3 R_1 contains t_2	t_2 in R_1 t_3 in R_0
$d = t_3 + t_2$	ADD R_1, R_0 MOV R_0, d	R_0 contains d	d in R_0 and memory

23/4/22

- Peephole optimization

→ Peephole - small moving window on the target program.

→ Peephole optimization - replaces short sequence of target instructions by a shorter or faster sequence.

→ Techniques -

i) Redundant Instruction Elimination
ii) Unreachable code elimination
iii) Flow of control optimization
iv) Algebraic simplification.
v) Use of machine idioms

i) Redundant Instruction Elimination

Eg
```
MOV R0, x
MOV x, R0   X
```

ii) Unreachable Code Elimination

Eg
```
x=0
if(x==1){
  a=b;
}
```

iii) Flow of control optimization

Eg goto L1 (can write directly goto L2) !!

```
    ....
L1: goto L2
    ....
L2: a = b;
```

iv) Algebraic Simplification

Eg: $x = x + 0$

$x \wedge 2 = x * x$

$x * x = x + x$

$x * 2 = x << 1$

$x * 2 = x >> 1$

v) Use of machine Idioms

$i = i + 1$ INC i

$i = i - 1$ DEC i

28/4/22

- Register allocation and assignment
  - → Instructions with register operands are faster than memory operands
  - → Efficient utilization of registers is important in generating the good code.
  - → Various strategies for register allocation and assignments are
    - i) global register allocation
    - ii) usage count
    - iii) Register assignment for outer loops
    - iv) Register allocation by graph coloring

i) => Global register allocation

- Keep frequently used value in a fixed register
- Assign some fixed no. of registers to hold the most active values in each inner loop.

ii) => Usage count $\sum_{\text{blocks } B \text{ in } L}$ use$(x, B) + 2 *$ live(x, B)

5 - ~~PE-III~~ , PE-II
1st 111rd

7 - CD , PE-1, PE-III
1st 2nd 3rd

9 - OE-II , DWDM
1st 5th hr

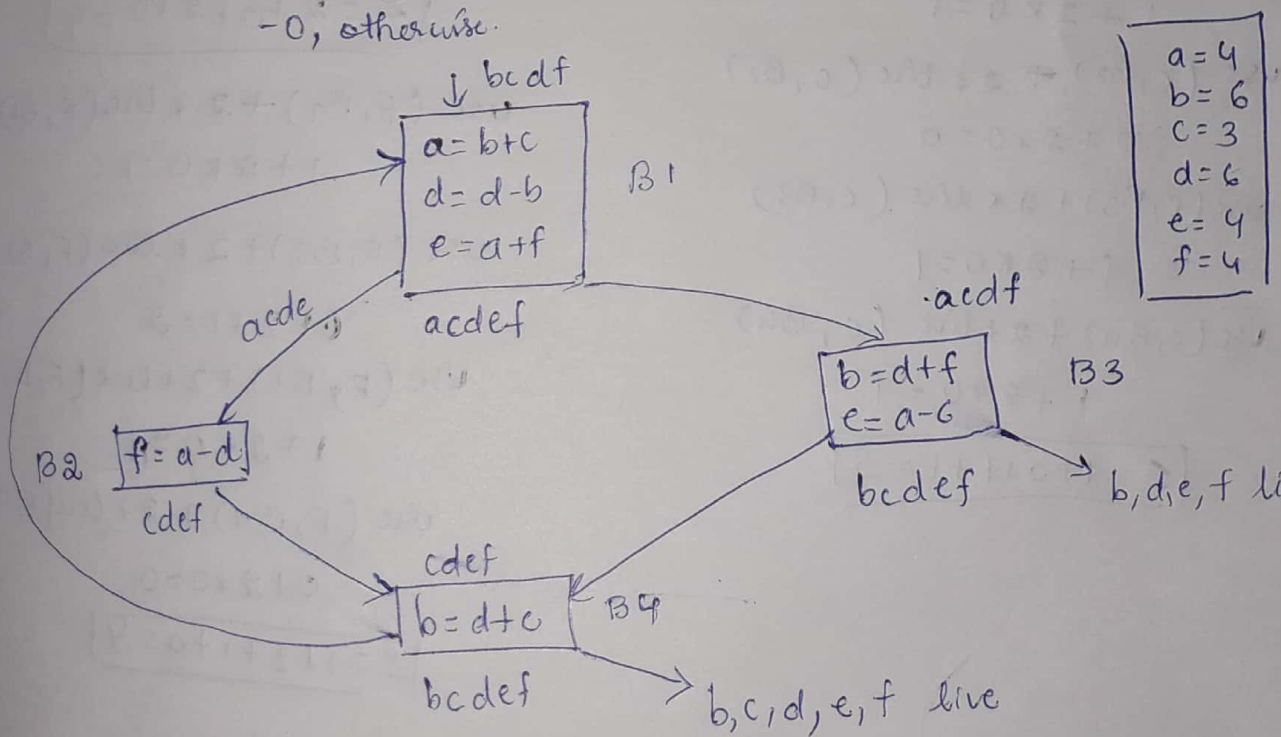
Assignment Qus

1) Construct a CLR 1 Parsing table for the given grammar.
2) Comparison of Topdown Parser and Bottom up parser
3) Explain the three Adress Code in Detail - Quadruples, triples, indirect triples.
4) Explain the Data Structures for symbol tables.
5) Explain the Principle sources of Optimization
6) Explain the issues in the code generation.

---

use (x, B) - No. of times x is used and not preceded by an assignment of x in same block

live (x, B) - 1, if x is live on exit and x is assigned a value in B.
- 0, otherwise.



use(a, B_1) + 2 * live(a, B_1)
0 + 2 * 1 = 2

use(a, B_2) + 2 * live (a, B_2)

using → right hand side
live: exist and assigned.

$1+2*0=1$

use$(a, B_3)+2*$ live(a, B_3)

$1+2*0=1$

use$(a, B_4)+2*$ live(a, B_4)

$0+2*0=0$

$\Sigma = 2+1+1+0=4$

use$(b, B_1)+2*$ live(b, B_1)

$= 2+2*0=2$

use$(b, B_2)+2*$ live(b, B_2)

$= 0+2*0=0$

use$(b, B_3)+2*$ live(b, B_3)

$0+2*1=2$

use$(b, B_4)+2*$ live(b, B_4)

$0+2*1=2$

$\Sigma = 2+0+2+2=6$

use$(c, B_1)+2*$ live(c, B_1)

$1+2*0=1$

use$(c, B_2)+2*$ live(c, B_2)

$0+2*0=0$

use$(c, B_3)+2*$ live(c, B_3)

$1+2*0=1$

use$(c, B_4)+2*$ live(c, B_4)

$1+2*0=1$

$\Sigma = 1+0+1+1=3$

use$(D, B_1)+2*$ live(D, B_1)

~~$2+2*0=2$~~ $1+2*1=3$

use$(D, B_2)+2*$ live(D, B_2)

$1+2*0=1$

use$(D, B_3)+2*$ live(D, B_3)

$1+2*0=1$

use$(D, B_4)+2*$ live(D, B_4)

$1+2*0=1$

$\Sigma = 3+1+1+1=6$

use$(E, B_1)+2*$ live(E, B_1)

$0+2*1=2$

use$(E, B_2)+2*$ live(E, B_2)

$0+2*0=0$

use$(E, B_3)+2*$ live(E, B_3)

$0+2*1=2$

use$(E, B_4)+2*$ live(E, B_4)

$0+2*0=0$

$\Sigma = 2+0+2+0=4$

use$(F, B_1)+2*$ live(F, B_1)

$1+2*0=1$

use$(F, B_2)+2*$ live(F, B_2)

$0+2*1=2$

use$(F, B_3)+2*$ live(F, B_3)

$1+2*0=1$

use$(F, B_4)+2*$ live(F, B_4)

$0+2*0=0$

$\Sigma = 1+2+1+0=4$

(ii) ⇒ Register assignment for outer loops: If an outer loop L_1 contains an inner loop L_2 the names allocated for register L_2 need not to be allocated in $L_1 - L_2$.

- If x is allocated to register L2 then Load x on entrance of L2 · the and store x on exit of L2.

iv) Register allocation by graph coloring: when an register is needed for computation but all registers are in use, the content s of one one register must be stored into memory location. There are two passes are used -

i) Target machine instructions are selected.

ii) Register interference graph is constructed and noded are symbolic registers and edge connects two nodes.