Miss Raazia Sosan

Compiler Construction

1-11-23

Chapter # 01: Introduction

Project: 15 Marks

Compiler converts high level language code into low-level, assembley language code.

Textbooks: Compiler Principles, Techniques & Tools.
by Alfred V. Aho, Ravi Sethi & Jeffry D. Ulman.
Publisher: Addison Wesley Longman, 1986.

Language Processor:

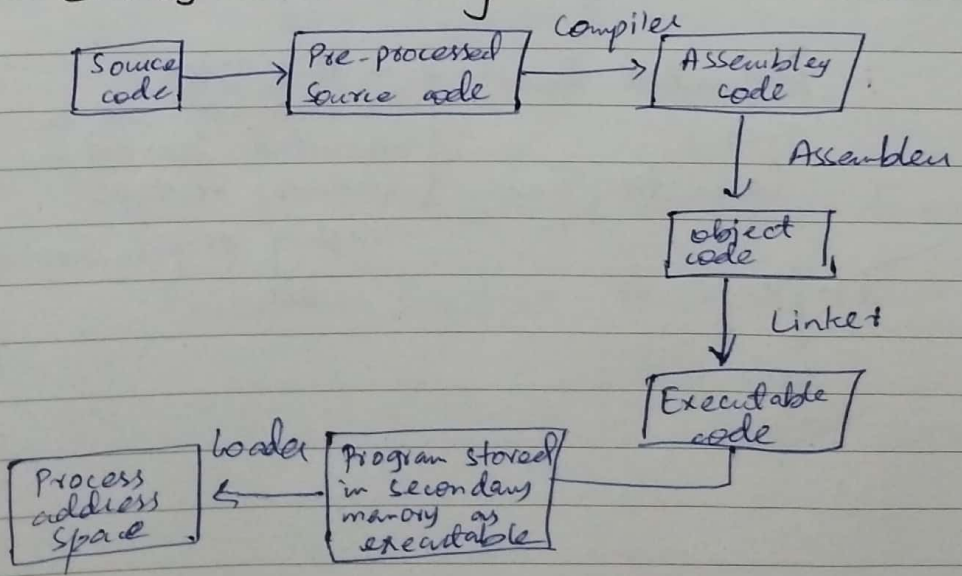
↳ Compiler: compiler makes an executable file which can be run anytime if no changes ~~have~~ are made.

↳ Interpreter: At run time it executes code line by line.

Hybrid compiler: works with both compiler & interpreter.
Such as Java.
program compiles into byte code & then it will interpret by JVM.

C Building Process

C-Program Building Process



Program

```
#include <stdio.h>
int main() {
    printf("Hello World");
}
```

→ pre-processor directive.
include → a directive
< ... > → default location
" ... " → user defined.

Structure of a Compiler

Compiler
- Analysis — Frontend
- Synthesis — Backend

Translation (phases).

position = initial + rate * 60.

↓

Lexical Analyzer.

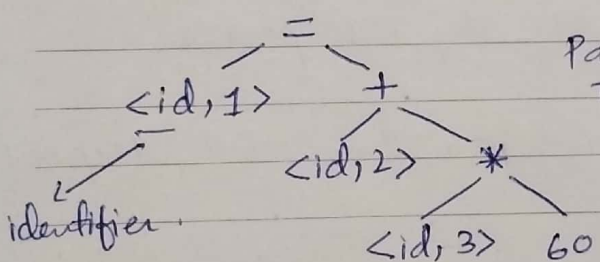
<id,1> <=> <id,2> <+> <id,3> <*> <60>.

} tokens divided into groups

↓

Syntax Analyzer.

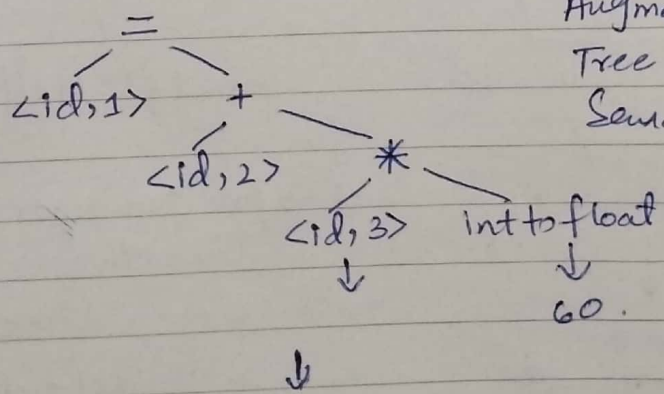
} context free grammar will check the order of tokens.
grammatically syntax will be checked



Parse Tree

↓

Semantic Analyzer



Augmented Parse Tree, Semantic Tree.

} in this step, the resources will be allocated based on memory or type casting or polymorphism etc.

↓

Intermediate Code Generator.

↓

t1 = inttofloat(60).
t2 = id3 * t1
t3 = id2 * t2.
id1 = t3.

↓

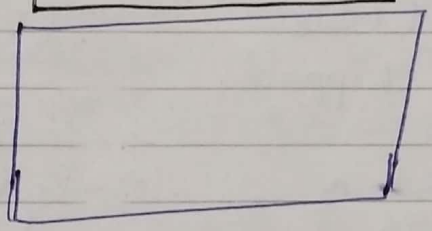
Code Optimizer

↓

t1 = id3 * 60.0.
id1 = id2 + t1.

↓

Code Generator.



Assembly code.

* Modified Source code.
  ↳ This will include all the code of header file

6-11-23

CHAPTER #2 : LEXICAL ANALYSER.

Two Phases:

— Analysis Phase:
- Breaks up modified source code into constituent piece. (chunks)
- Produces an ~~intermediate~~ internal representation known as intermediate code.

— Synthesis Phase:

COMPILER - FRONT END.

Parser → Syntax Analyzer.

The Role Of The Lexical Analyser.

Lexical analyzer will group / classify the modified source code words according to rules → regular expression and generates a token which will be passed to parser

Example: int x = 7 ;

⇒ dataType identifier eqOp numericConstant st Term.
these are tokens.

- lexemes: grouped characters.
- identifiers are added to symbol table.
- comment and whitespaces are stripped

★ Programming constructs.
→ looping structures
→ othe rules.
→ rules for variable
→ selection stmt
→ if else.

Lexical Analyzer - Process.

Lexical Analyzer are divided into a cascade of two processes.

1 → Scanning.
2 → Tokenization.

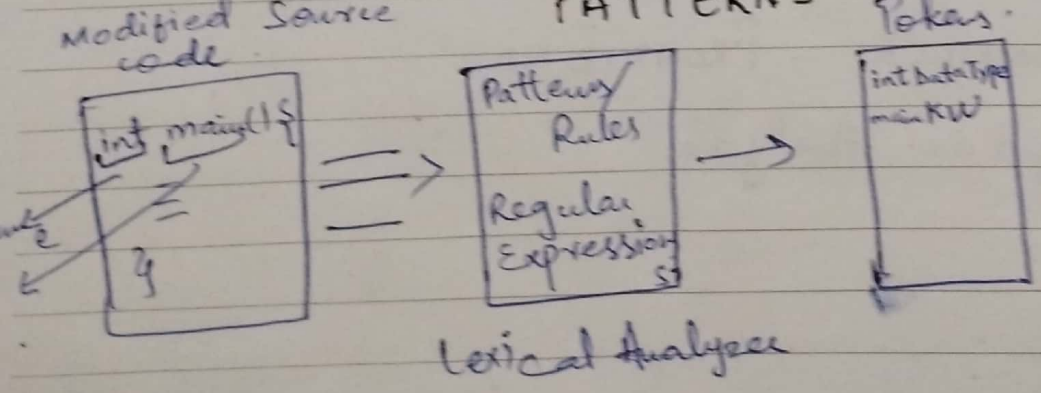
Tokens.

→ A pair with token name & its attribute (optional)

< token name , token attribute >

→ token name is in boldface by convention.

*.pdf



8-11-23

Example 3.1

printf (" Total = %d \n " , score) ;

ST: statement terminator ; OB: opening bracket id: identifier

Tokens: <printf_RW> <OB> <string_const> <comma> <id>, <ST> <CB>

One token for each keyword. make token name same as keyword
One token for all identifiers.

Example 3.2.

Write token names & associated attribute values. for the following.

E = M * C ^ 2

<id,1> <eqOP> <id,2> <mulOP> <id,3) <PowOP> <numConst, 2).

Input Buffering.

Buffer is nearest to compiler which can read. code file's ~~data~~, which compiler give to it.

LB: lexeme Begin
FWD: Forward

H e l l o !
LB ↑ FWD ↑ Fwd Fwd ...

LB => lexemeBegin
FWD => Forward
↳ pointers to the input.

Two Buffers, working simultaneously.

Sentinels.
↳ ending characters.

LEXICAL ANALYZER - PROCESS.

Thomson Construction
RE → NFA.

Subset Construction
NFA → DFA.

DFA will always be same.
↳ it will be coded.

13-11-23

Table => part of PLS.

Sample Lexeme	Rule	Token
if	if	if_KW
else	else	else_KW
NUM	_[A-Z]*	id

DSL Domain Specific Language.
GPL General Purpose Language.
PLS Programming Language Specification document.

Regular Expression. Characters in R.E.

$L(a) = \{a\}$
$L(\epsilon) = \{\epsilon\}$ empty string
$L(\phi) = \{\}$ null, contains no string at all.

Types of R.E.

1) choice $\longrightarrow L(a|b) = \{a, b\}$.
2) concat
3) repitition.

(3)(1) Choice => $L(a|b) = L(a) \cup L(b)$
$L(a|b) = \{a\} \cup \{b\}$
$L(a|b) = \{a, b\}$

(2)(2) Concat => $L(ab) = L(a)\,L(b)$
$L(ab) = \{ab\}$

(1)(3) Repitition => $L(a^*) = L(\epsilon) \cup L(a) \cup L(a)L(a) \cup \ldots$
$L(a^*) = \{\epsilon, a, aa, aaa, \ldots\}$.

$L(a|bb^*) = \{a, b, bb, bbb, \ldots\}$.

① *
② concat
③ | choice

$L(a|bb^*) = ?$

① $L(b^*) = L(\epsilon) \cup L(b) \cup L(b)L(b) \cup \ldots$

$L(b^*) = \{\epsilon, b, bb, \ldots\} \rightarrow A$

② $L(b\boxed{A}) = L(b) \cdot L(\boxed{A})$

$= \{b\boxed{A}\}$

$L(b\boxed{A}) = \{b, bb, bbb, \ldots\} \rightarrow B$

③ $L(a|\boxed{B}) = L(a) | L(\boxed{B})$

$= \{a, \boxed{B}\}$.

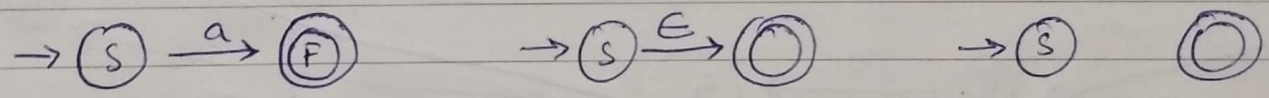
$L(a|\boxed{B}) = \{a, b, bb, bbb, \ldots\}$.

THOMPSON CONSTRUCTION

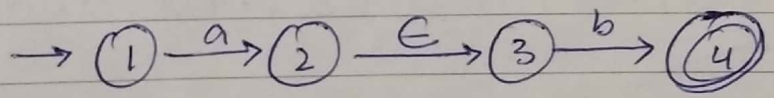
15-11-23

R.E → NFA.

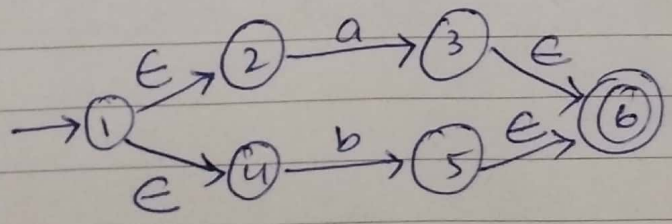
① $L(a) = \{a\}$ ② $L(\epsilon) = \{\epsilon\}$ ③ $L(\phi) = \{\}$.



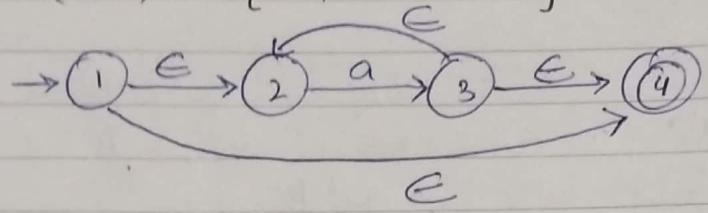
④ $L(ab) = \{a \bullet b\}$ Hint: Serial.



⑤ $L(a|b) = \{a, b\}$ Hint: Parallel.

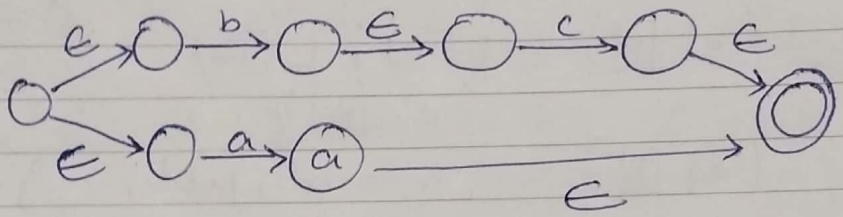


⑥ $L(a^*) = \{\epsilon, a, aa, \cdots\}$

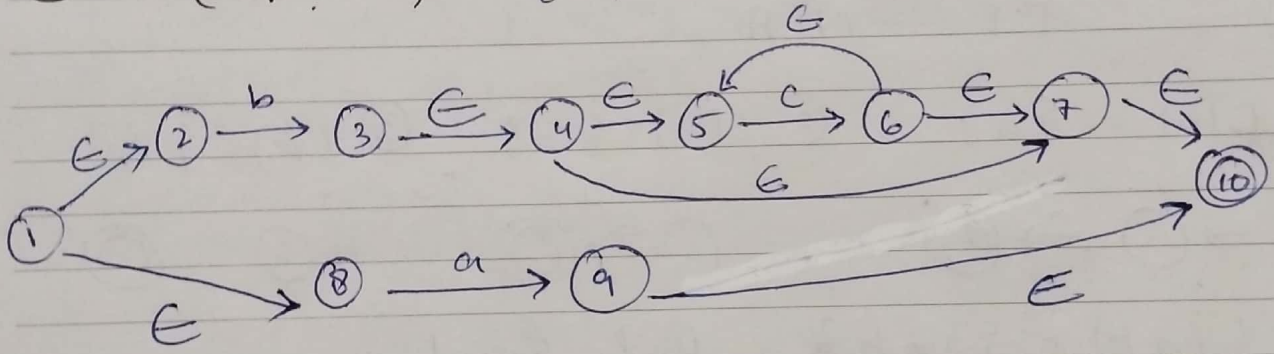


⑦ $L(a|bc) = \{a, bc\}$

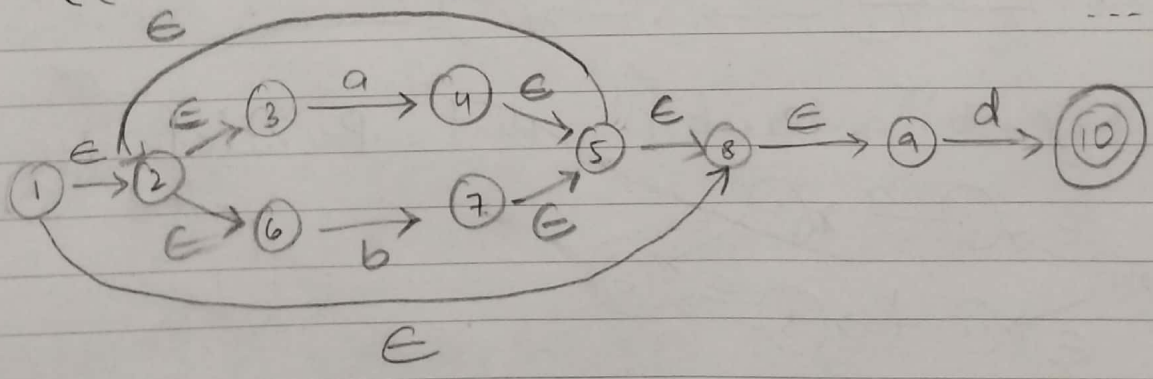
① Repitition
② Concat
③ OR.



⑧ $L(a|bc^*) = \{a, b, bc, bcc, \cdots\}$.



⑨ $L((a|b)^* d) = \{d, ad, aad, bd, bbd, abd, bad, \ldots\}$.

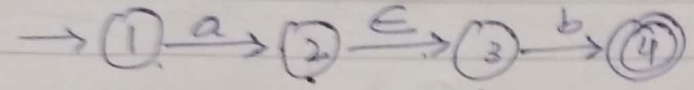


Subset Construction

22-11-23

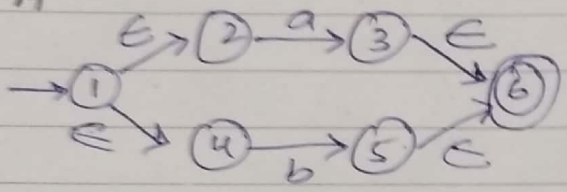
① $L(ab) = \{ab\}$

NFA



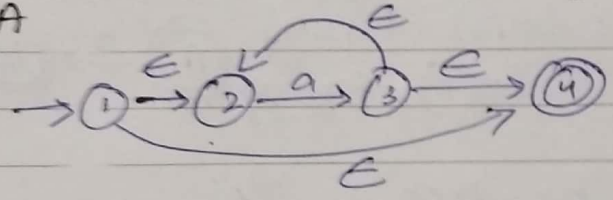
② $L(a|b) = \{a, b\}$.

NFA



③ $L(a^*) = \{\epsilon, a, aa, \dots\}$.

NFA



Eclosures

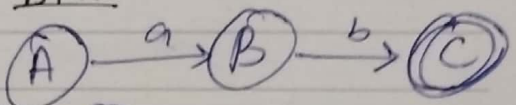
ϵ closure (1) = {1}.
ϵ " (2) = {2, 3}.
" (3) = {3}
" (4) = {4}.

ϵ closure (1) = {1, 2, 4}.
" (2) = {2}.
" (3) = {3, 6}.
" (4) = {4}.
" (5) = {5, 6}.
" (6) = {6}.

ϵ closure (1) = {1, 2, 3, 4}.
" (2) = {2}.
" (3) = {3, 2, 4}
" (4) = {4}.

① $L(ab)$.

DFA



start ϵ closure {1} = {1} — (A)

ϵclos (move (A, a)) = ϵclos (2) = {2, 3} — (B)

ϵclos (move (A, b)) = ϕ.

ϵclos (move (B, a)) = ϕ

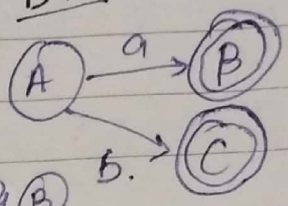
ϵclos (move (B, b)) = ϵclos (4) = {4} — (C)

ϵclos (move (C, a)) = ϕ

ϵclos (move (C, b)) = ϕ.

② L(a|b)

DFA



Start $\in clos(1) = \{1, 2, 4\}$ — (A)

$\in clos(move(A, a)) = \in clos(3) = \{3, 6\}$ (B)

$\in clos(move(A, b)) = \in clos(5) = \{5, 6\}$ (C)

$\in clos(move(B, a)) = \phi$

$\in clos(move(B, b)) = \phi$

$\in clos(move(C, a)) = \phi$

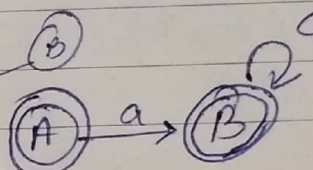
$\in clos(move(C, b)) = \phi$

③ L(a*)

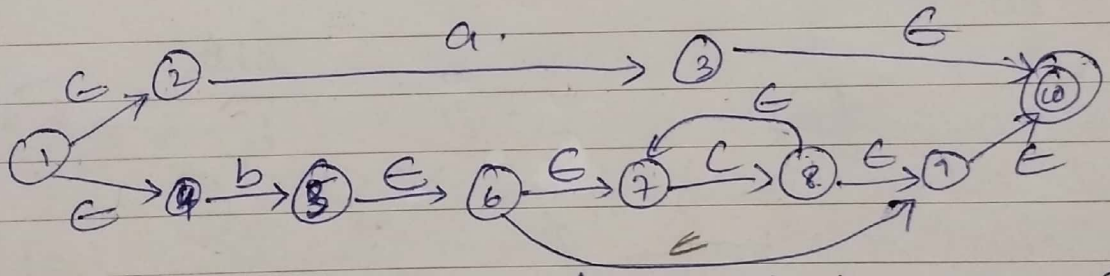
Start $\in clos(1) = \{1, 2, 4\}$ — (A)

$\in clos(move(A, a)) = \in clos(3) = \{3, 2, 4\}$ (B)

$\in clos(move(B, a)) = \in clos(3)$ — (B).



L(a|bc*) = {a, b, bc, bcc, ... }.



$\in clos\{1) = \{1, 2, 4\}$.

(2) = {2}.

(3) = {3, 10}.

(4) = {4}.

(5) = {5, 6, 7, 9, 10}.

(6) = {6, 7, 9, 10}.

(7) = {7}.

(8) = {8, 7, 9, 10}.

(9) = {9, 10}.

(10) = {10}.

Start $\in clos(1) = \{1, 2, 4\}$ — (A).

30 | MAR' 2020 MONDAY
1441 شعبان 5

MARCH
M T W T F S S
30 31 1
2 3 4 5 6 7 8
9 10 11 12 13 14 15
16 17 18 19 20 21 22
23 24 25 26 27 28 29

Start $\in$clos (1) = {1, 2, 4} → (A)

$\in$clos(move(A,a)) = $\in$clos(3) = {3, 10} → (B)
$\in$clos(move(A,b)) = $\in$clos(5) = {5, 6, 7, 9, 10} → (C)
$\in$clos(move(A,c)) = ϕ

$\in$clos(move(B,a)) = ϕ
$\in$clos(move(B,b)) = ϕ
$\in$clos(move(B,c)) = ϕ

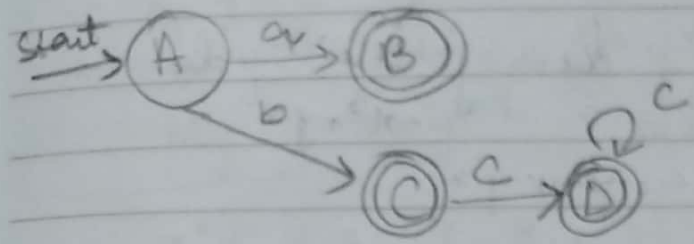
$\in$clos(move(C,a)) = ϕ
$\in$clos(move(C,b)) = ϕ
$\in$clos(move(C,c)) = $\in$clos(8) = {8, 7, 9, 10} → (D)

$\in$clos(move(D,a)) = ϕ
$\in$clos(move(D,b)) = ϕ
$\in$clos(move(D,c)) = $\in$clos(8) → (D)

MAR' 2020 TUESDAY
1441 شعبان 6

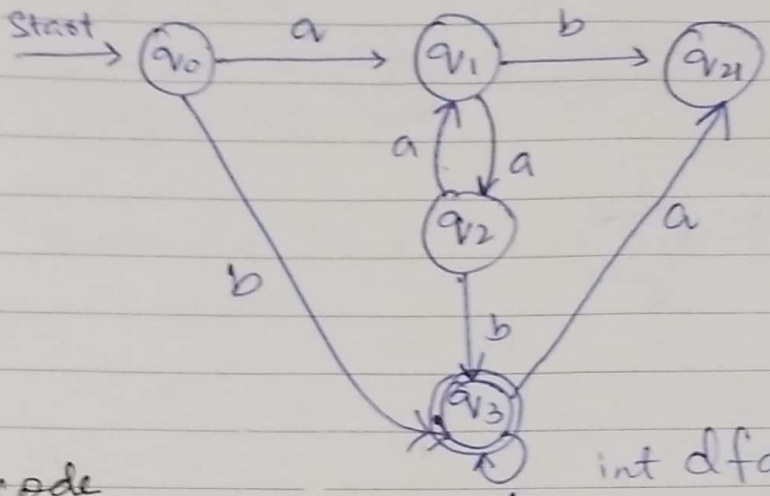
MARCH
M T W T F S S
30 31 1
2 3 4 5 6 7 8
9 10 11 12 13 14 15
16 17 18 19 20 21 22
23 24 25 26 27 28 29

L(a|bc*)



27-11-23

DFA



Language Name

DFA code

Group members Info

Abstract | Lexeme | Pattern | Token

Lexical component

int dfa = 0

code

```
isAccepted(char[] str){
  int i, len = str.len(str)
  for(int i=0; i<= len; i++){
    if (dfa == 0) state(str[i]) = dfa
    else if (dfa == 1)  dfa = state(str[i])
    else if (dfa == 2)  state2(str[i])
    else if (dfa == 3)  state3(str[i])
    else if (dfa == 4)  state4(str[i])
    else dfa = -1; break
  if(dfa == 3){
    return 1
  }
  else{
    return 0
  }
}
```

```
start(char c){
  if (c == "a"){
    dfa = 1
  } else if(c == "b")
    dfa = 3
  }. else { dfa = -1 }.
}
```

```
State 1(char c){
  if (c == "a"){
    dfa = 2
  } else if (c == "b"){
    dfa = 4
  } else {
    dfa = -1
  }.
}
```

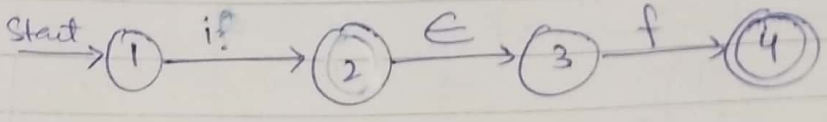
```
state2(char c){
  if (c == "a"){
    dfa = 1
  } else if (c == "b"){
    dfa = 3
  } else {
    dfa = -1
  }
}
```

```
State 3(char c){
  if(c == "a"){
    dfa = 4
  } else if (c == "b"){
    dfa = 3
  } else {
    dfa = -1
  }.
}
```

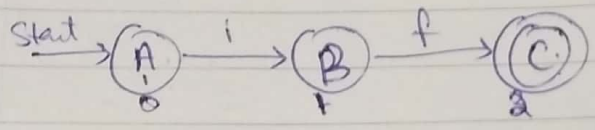
```
State 4(char c){
  if (c == "a"){
    dfa = -1
  }.
}
```

L(if) = {if}.

NFA



DFA



Code:

```
int dfa = 0.
isAccepted(char[] str){.
    for(int i=0; i<= str.length(str); i++){.
        if (dfa==0)
            start(str[i])
        else if (dfa == 1)
            state1(str[i]
        else if (dfa == 2)
            state2(str[i])
        else
            dfa=-1
            break.
    }.
    if(dfa == 3)
        return 1
    else
        return 0.
}.
```

```
start0(char c) {.
    if(c == "i")
        dfa=1
    else
        dfa=-1
}
```

```
state1(char c) {
    if(c == "f").
        dfa=2
    else
        dfa=-1
}
```

```
state2(char c){.
    dfa=-1
}.
```

29-11-23.

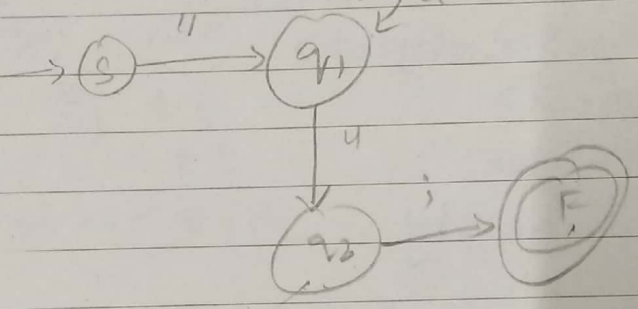
Files to be submitted. 8-Dec (Deadline).

- .cpp → } LEX.
- .flex → }
- .pdf → PLS

incorrect_file.nex } sample code.
correct_file.nex }

Learning Flex

(1) My name is Saad (2) Saad



int i = "saad";

string s

Make a regular expression OR separated.
(if | else | then | &)*

SYNTAX ANALZER / PARSER. 6-12-23

- order of tokens will matter.

TYPES OF PARSER:

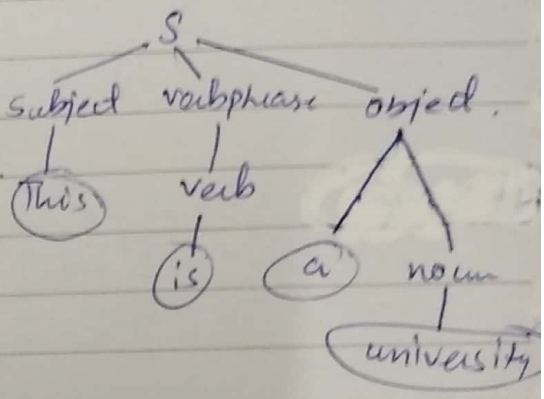
- Top-Down Parser (root to leaf)
- Bottom-up Paiser (leaf to root).
- Universal.

Grammar: It is a set of rules by which valid sentences in a language are constructed.

Terminals — Non-terminals.
↳ Tokens

Example:

This is a university.
↳ SA



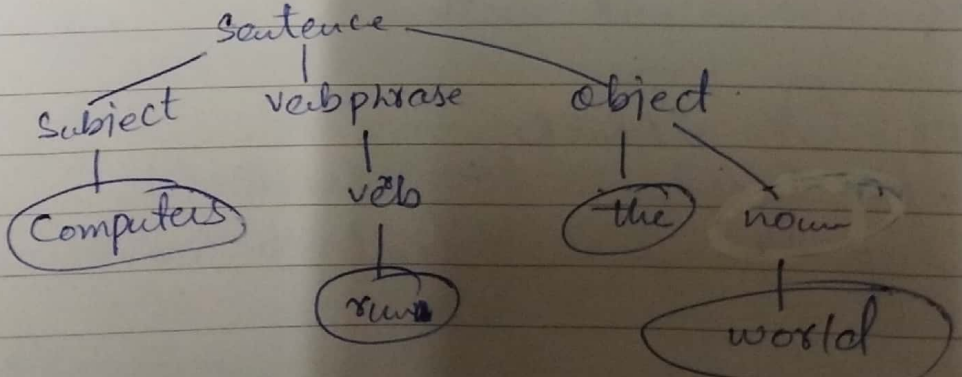
Leftmost Derivation.

Sentence → <subject> <verbphrase> <object>.
→ This <verbphrase> <object>
→ This is <object>
→ This is a <noun>
→ This is a university

Example: Computers run the world.

LMD: Sentence → <subject> <verbphrase> <object>
→ Computers <verbphrase> <object>.
→ Computers <verb> <object>.
→ Computers runs <object>.
→ Computers runs the <noun>
→ Computers runs the world.

Parse Tree:



Definition of Grammar.

① Terminals :- Tokens, leaves in a parse tree.
② Productions :- Rules of the grammar.
③ Non-terminals :- syntactic variables, intermediate.
④ Start symbol ; A designation of one of the non-terminal

Derivations.

$E \rightarrow E+E \mid E*E \mid -E \mid (E) \mid id$.

① $E \rightarrow E+E$
$\rightarrow id+E$
$\rightarrow id+E*E$
$\rightarrow id+id*E$
$\rightarrow id+id*id$

② $E \rightarrow E*E$
$E \rightarrow E+E*E$
$\rightarrow id+E*E$
$\rightarrow id+id*E$
$\rightarrow id+id*id$.

When your parser generate more than one tree, Then the grammar is ambiguous. or LMD.

Solution for ambiguity.

operators on same line have same associativity & precedence

left-associative : $+ -$
" " : $* /$

$\Rightarrow$ $E \rightarrow E+E \mid E*E \mid -E \mid (E) \mid id$. ✗

collecting grammar.

$E \rightarrow E+T \mid E-T \mid T$
$T \rightarrow T*F \mid T/F \mid F$
$F \rightarrow id \mid (E) \mid -E$ ✓

PAPERWORK

Elimination of Left Recursion.

- Left recursion.

  $A \rightarrow A\alpha / \beta$ → left recursive, left variable with a right variables are same.

- Top-down parser can't handle left-recursion.

General form

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

solution non-left-recursive.

Example

① $E \rightarrow E+T \mid T.$ $\quad A \rightarrow E,\ \alpha \rightarrow +E,\ B \rightarrow T.$

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

② $E \rightarrow E+T \mid E-T \mid T.$

$E \rightarrow TE'E^2$
$E' \rightarrow +TE' \mid \epsilon$
$E^2 \rightarrow -TE^2 \mid \epsilon$
$E' \rightarrow +TE' \mid -TE' \mid \epsilon$

✓✓

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

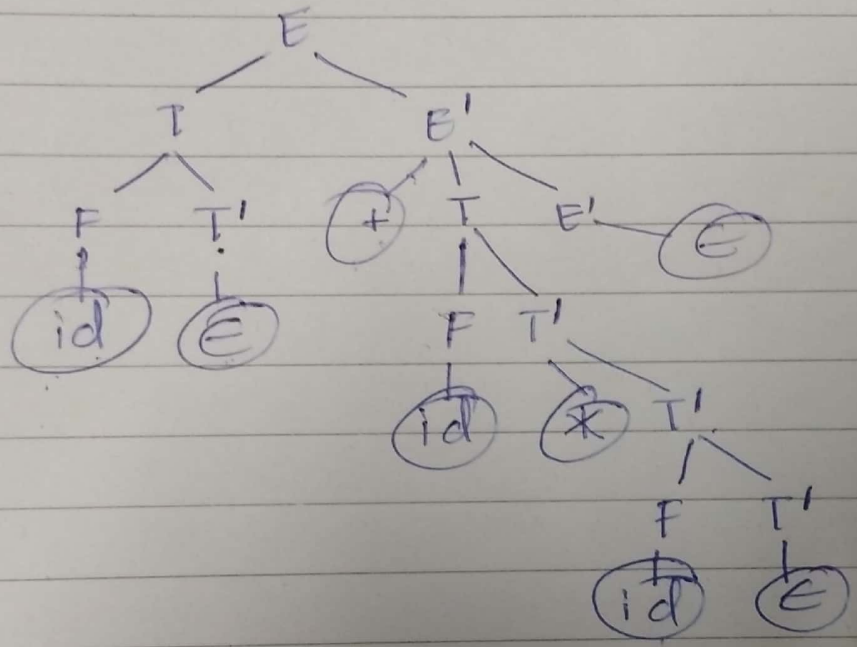
③ $T \rightarrow T*F \mid T/F \mid F$

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

④ $F \rightarrow id \mid (E) \mid -E)$

$F \rightarrow id \mid (E) \mid -E$

id + id *id



Start → E → .T E'

E' → + TE' | - TE' | ϵ

T → FT'

T' → * FT' | / FT' | ϵ

F → id | (E) | -E

18-12-23.

FINALS

Left Factoring:

① $A \rightarrow \alpha\beta_1 \mid \alpha\beta_2$

$\downarrow$

$A \rightarrow \alpha A'$

$A' \rightarrow \beta_1 \mid \beta_2$.

② stmt → if expr then stmt else stmt | if expr then stmt

$\downarrow$

stmt → if expr then stmt A'

A' → else stmt | ∈

③ $A \rightarrow A + B \mid B$.

$B \rightarrow int \mid (A)$

$\downarrow$

$A \rightarrow A\,A + B'$

$B' \rightarrow B \mid \in$

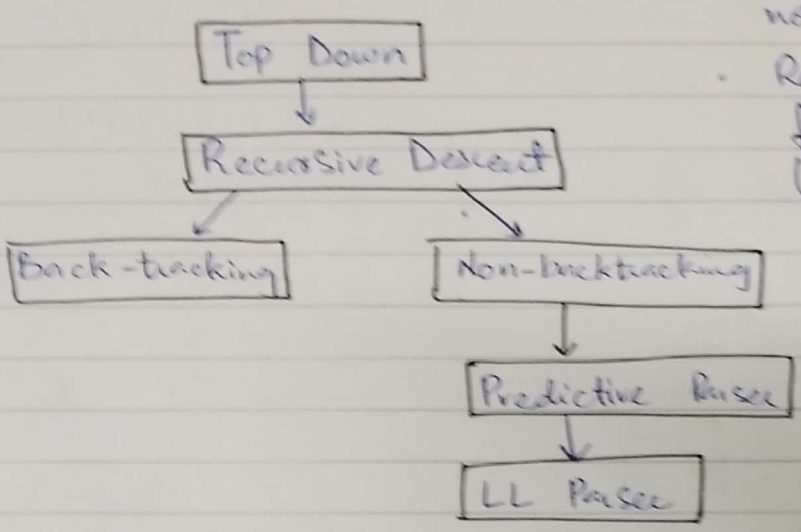
Eliminating left recursion →

$A \rightarrow B\,A'$

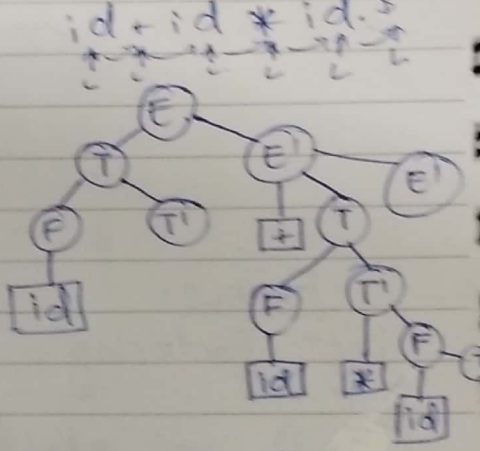
$A' \rightarrow + B\,A' \mid \in$

$B \rightarrow int \mid (A)$.

TAXONOMY



- Top down parser should not have left-recursion.
- Recursive descent parser for your language (recursive calls).



Grammar:

$E \rightarrow TE'$

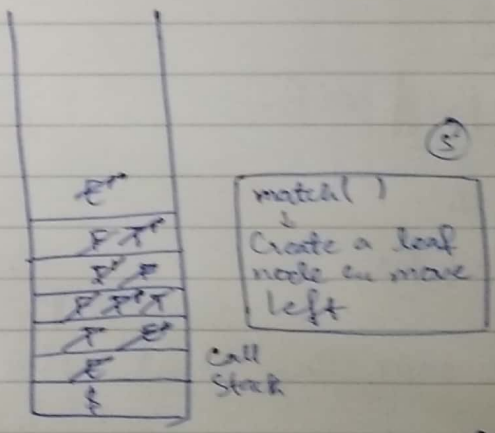
$E' \rightarrow +TE' \mid \epsilon$

$T \rightarrow FT'$

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

$F \rightarrow (E) \mid id$

L -> lookahead



- Make all non-terminals, void functions.

```
① void E() {
      T();
      E'();
  }

② void E'() {
      if(L == '+') {
          match('+');
          T();
          E'();
      } else {return;}
  }

③ void T() {
      F();
      T'();
  }

④ void T'() {
      if(L == '*') {
          match('*');
          F();
          T'();
      } else {return;}
  }

⑤ void F() {
      if(L == '(') {
          match('(');
          E();
          match(')');
      } else if(L == 'id') {
          match('id');
      }
  }

int main() {
    E();
    if(L == '$') {
        printf("Success");
    } else {
        return;
    }
}
```

ACTIVITY

20-12-23.

Qa. Implement a recursive descent parser for the following grammar.

$S \rightarrow STu \mid Sv \mid v$

$T \rightarrow Tx \mid y.$

↓

$S \rightarrow vS'$

$S' \rightarrow TuS' \mid \in \mid vS'$

$T \rightarrow yT'$

$T' \rightarrow xT' \mid \in$

Recursive descent Parser.

①
```
void{ S() {
    if (L == 'v') {
        match('v');
    }. S'();
    } else {
        return;
    }
}
```

②
```
void S'() {.
if(L=='y'){T();
    if (L == 'u') {
        match('u');
        S'(); }.
    } else if (L=='v'){.
        match('v');
        S'();
    }else {
        return;
    }
}
```

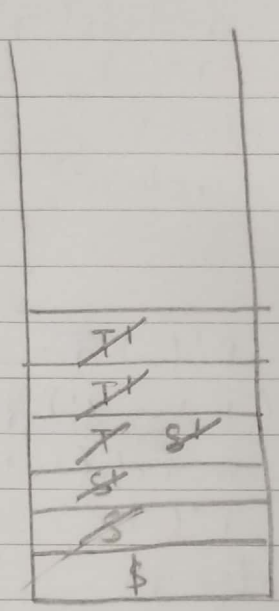
③
```
void T() {
    if(L=='y') {.
        match('y');
        T'();
    } else {
        return;
    }
}.
```

④
```
void T'() {.
    if(L=='x') {.
        match('x');
        T'();
    } else {
        return;
    }
}.
```

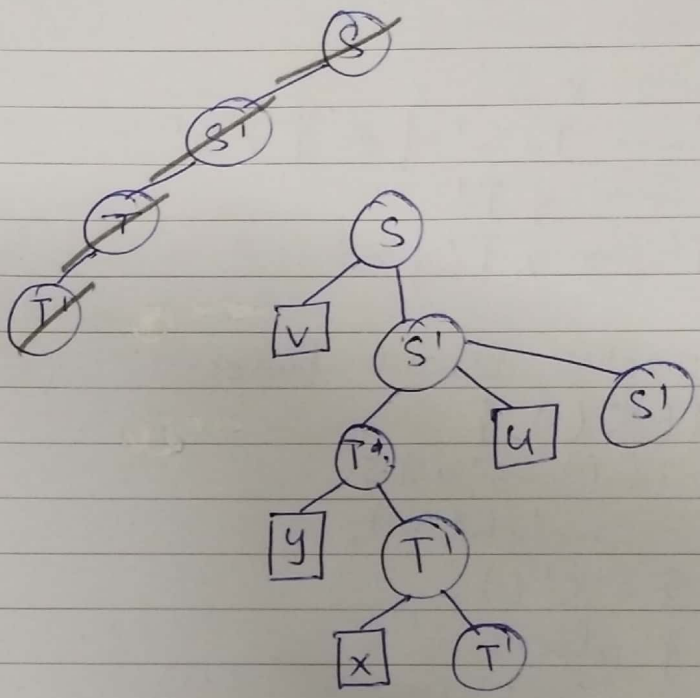
```
int main() {.
    S();
    if(L=='$') {.
        printf("Success");
    else {.
        printf("failed");
    }
    return 0;
}.
```

Parse the following tokens, construct the syntax tree and show the stack operations.

y x u $



V y x U $



★ Add Grammar and sample code under Syntax Analyzer heading in the PLS.

Top-down parser → LL(1) Parser
Bottom-up parser → LR(0) Parser.

LL => L means Left to right reading
L means Left most derivation

LL(k) Parser.

k means, lookahead will read k words from the input.

LL(k)
- FIRST()
- FOLLOW()

LHS → RHS
E → TE'
E' → +TE' | ∈

- nonterminal
- terminal
- ∈

FIRST(terminal) = {terminal}
FIRST(∈) = {∈}

GRAMMAR	FIRST()
E → TE'	FIRST(E) = FIRST(T) = {(, id}
E' → +TE' | ∈	FIRST(E') = FIRST(+) ∪ FIRST(∈) = {+, ∈}
T → FT'	FIRST(T) = FIRST(F) = {(, id}
T' → *FT' | ∈	FIRST(T') = FIRST(*) ∪ FIRST(∈) = {*, ∈}
F → (E) | id	FIRST(F) = FIRST(() ∪ FIRST(id) = {(, id}

PAPERWORK

3-1-24

Grammar:

$S \to aBDH$ — First(S) = First(a) = {a}.

$B \to cC$ — First(B) = First(c) = {c}

$C \to bC \mid \epsilon$ — First(C) = First(b) ∪ First(ϵ) = {b, ϵ}.

$D \to EF$ — First(D) = (First(E) − ϵ) ∪ First(F) = {g, f, ϵ}.

$E \to g \mid \epsilon$ — First(E) = First(g) ∪ First(ϵ) = {g, ϵ}

$F \to f \mid \epsilon$ — First(F) = First(f) ∪ First(ϵ) = {f, ϵ).

~~First~~

Grammar.

$S \to AaAb \mid BbBa$. ∴ First(S) = (First(A) − ϵ) ∪ First(a) ∪ ((First(B) − ϵ) ∪ First(

$A \to \epsilon$ — First(A) = First(ϵ) = {ϵ}

$B \to \epsilon$ — First(B) = First(ϵ) = {ϵ}.

∴ First(S) = {a, b}.

FOLLOW ( )

Grammar:	First()	Follow()
$E \to TE'$	First(E) = {(, id}	Follow(E) = {$,)}
$E' \to +TE' \mid \epsilon$	First(E') = {+, ϵ}	Follow(E') = Follow(E) = {$,)}
$T \to FT'$	First(T) = {(, id}	Follow(T) = First(E') = {+, $,)}
$T' \to *FT' \mid \epsilon$	First(T') = {*, ϵ}	Follow(T') = Follow(T) = {+, $,)}
$F \to (E) \mid id$	First(F) = {(, id}	Follow(F) = {First(T') − ϵ} ∪ Follow(T') = {*, +, $,)} ∪ Follow(T)

LHS — Follow() RHS — First()

Follow(T) = [{First(E') − ϵ} ∪ Follow(E)] ∪ {First(E') − ϵ} ∪ Follow(E')
= {+, $,)}

* Follow does not contain ϵ

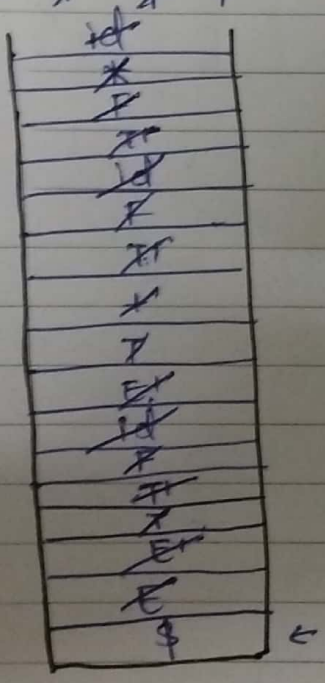
Grammar:	First	Follow
$S \to aBDh$	First(S) = {a}	Follow(S) = {$}
$B \to cC$	First(B) = {c}	Follow(B) = {First(D) − ϵ} ∪ First(h) = {g, f, h}
$C \to bC \mid \epsilon$	First(C) = {b, ϵ}	Follow(C) = Follow(B) ∪ Follow(C) = {g, f, h}
$D \to EF$	First(D) = {g, f, ϵ}	Follow(D) = First(h) = {h}
$E \to g \mid \epsilon$	First(E) = {g, ϵ}	Follow(E) = {First(F) − ϵ} ∪ Follow(D) = {f, h}
$F \to f \mid \epsilon$	First(F) = {f, ϵ}	Follow(F) = Follow(D) = {h}

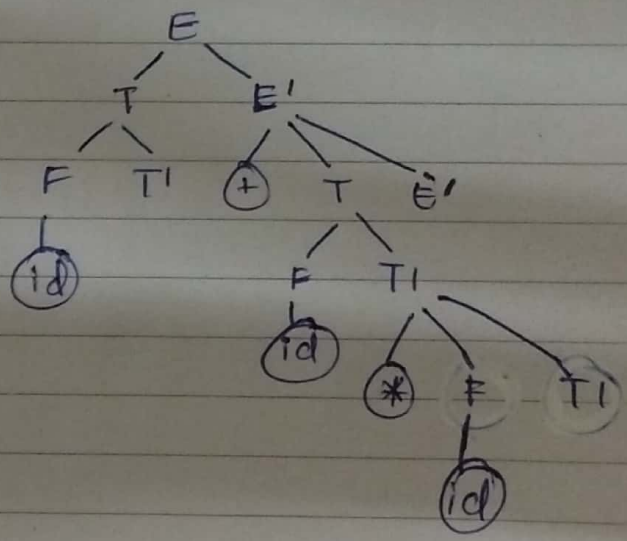
Grammar	First	Follow
$S \to AaAb \mid BbBa$	First(S) = {a, b}	Follow(S) = {$}
$A \to \epsilon$	First(A) = {ϵ}	Follow(A) = First(a) ∪ First(b) = {a, b}
$B \to \epsilon$	First(B) = {ϵ}	Follow(B) = First(b) ∪ First(a) = {b, a}

LL(1) Parsing Table.

Non-terminals	Terminals: +	*	(	)	id	$
E			E→TE'		E→TE'	
E'	E'→+TE'			E'→ϵ		E'→ϵ
T			T→FT'		T→FT'	
T'	T'→ϵ	T'→*FT'		T'→ϵ		T'→ϵ
F			F→(E)		F→id.	

id + id * id.$





	a	b	c	f	g	h	$
S	S→aBDh						
B			B→cC				
C		C→bC		C→ϵ	C→ϵ	C→ϵ	
D				D→EF	D→EF	D→EF	
E				E→ϵ	E→g	E→ϵ	
F				F→f		F→ϵ	

SDT

10-1-24.

Project Presentation:

PLS document in hadoop - 1 page

1. Intro → title, group members.
2. Language abstract
3. Snapshots → Tokens
   Grammar.
4. Demo → miss ko project chalta hua dikhao.

S → Presentation
S → Lexical Analyzer
S → Parser
S → PLS Doc.

bonus marks for bison ka visualization of tree.

SDT : SYNTAX DIRECTED TRANSLATION
SEMANTIC ANALYSIS

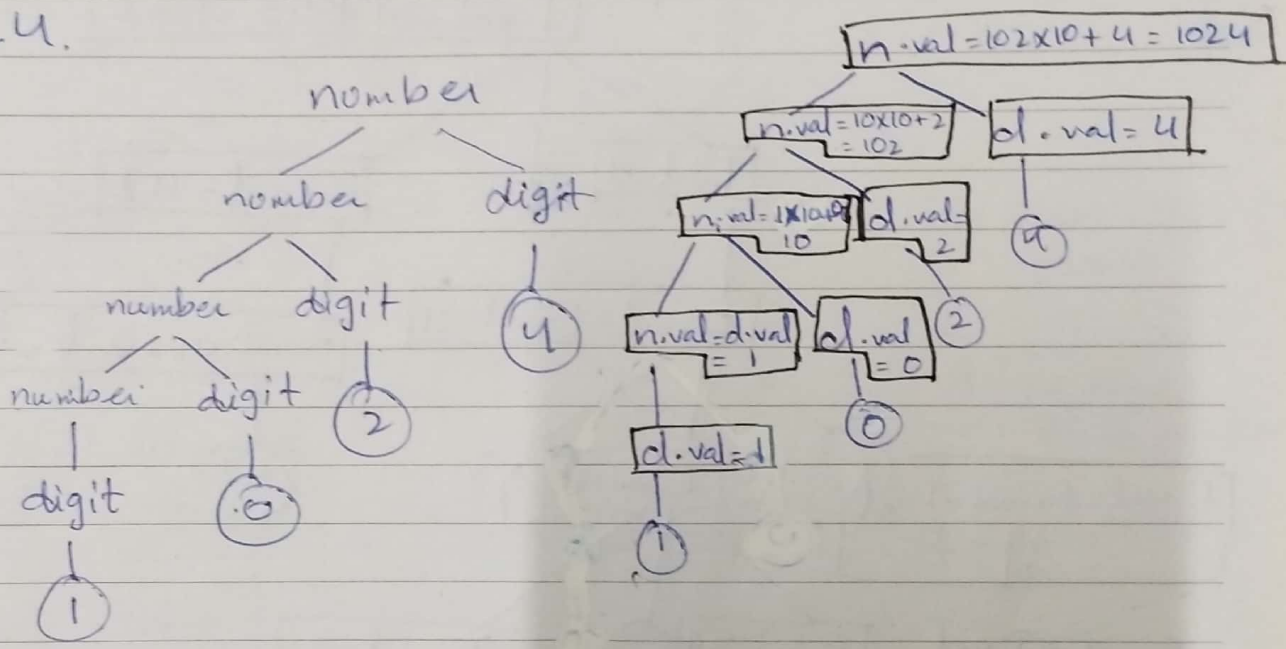
Syntax-Directed Definition. SDD
updating the grammar.
any attributes.

LL-Parsable → SDD is L-attributed.
LR-Parsable → SDD is S-attributed.

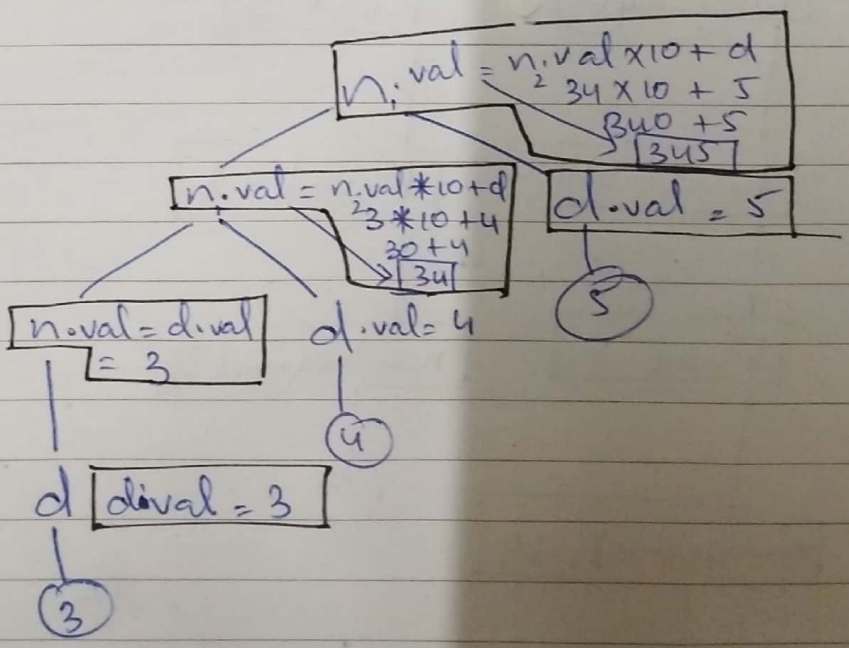
number → number digit | digit.
digit → 0|1|2|3|4|5|6|7|8|9.

Grammar Rule	Semantic Rule.
number$_1$ → number$_2$ digit	number$_1$.val = number$_2$.val * 10 + digit.val
number → digit	number.val = digit.val.
digit → 9	digit.val = 9.

1024.



345

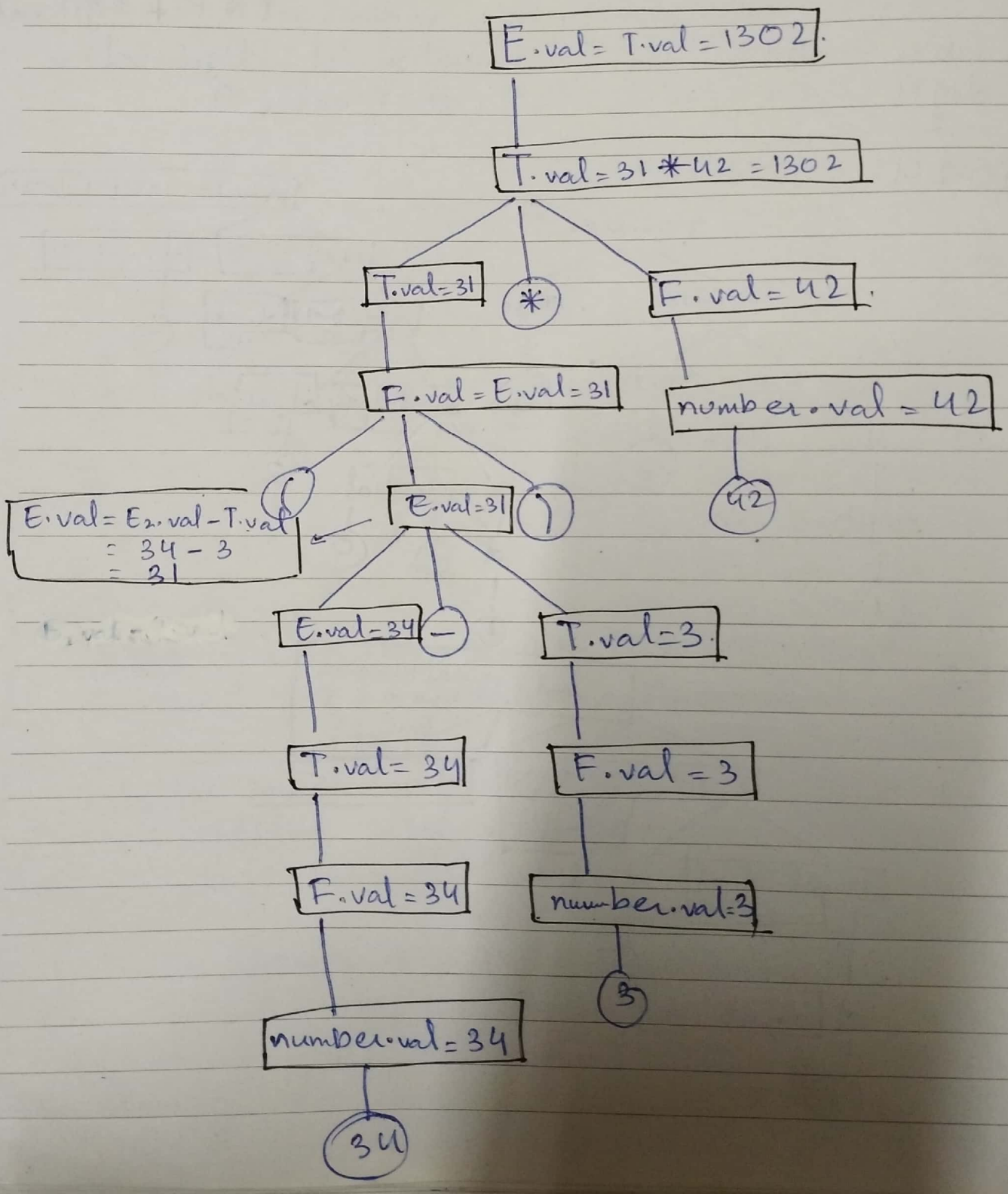


PAPERWORK

$(34-3)*42$.

```
  31
 x42
 162
124x
1302
```

exp → exp + term | exp − term | term.
term → term * factor | factor.
factor → (exp) | number

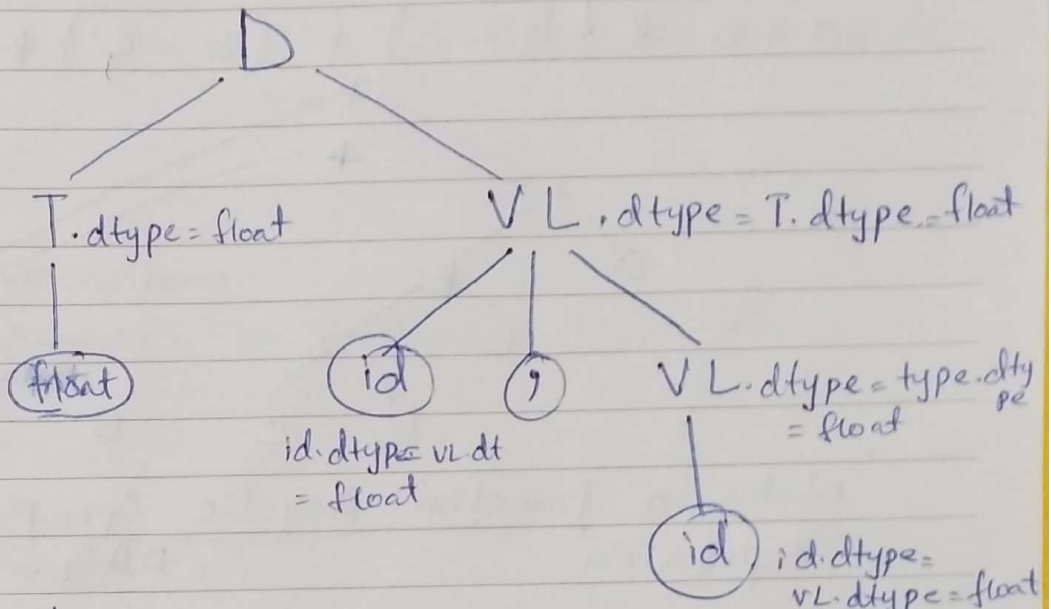


Ex: 3.

decl → type var-list.
type → int | float.
var-list → id, varlist | id

D → T VL
T → int | float.
VL → id, VL | id.

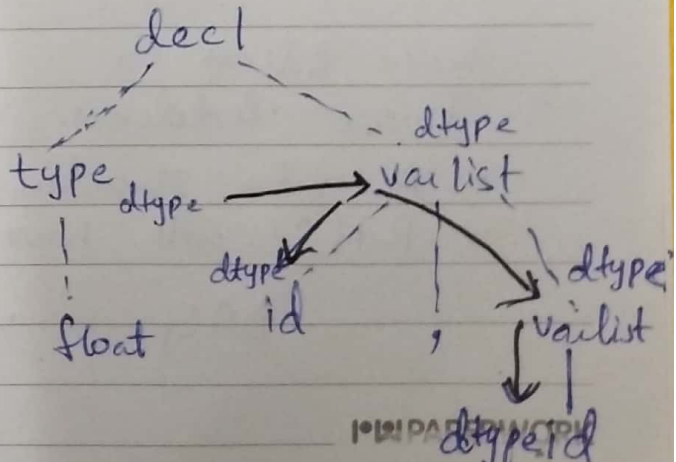
float x, y.



Synthesized attribute:
Child to parent e.g. val

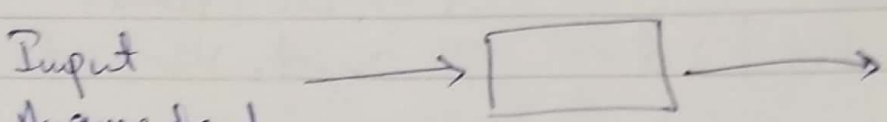
Inherited attribute:
Parent to child or siblings. e.g. dtype

DEPENDENCY GRAPH 17-1-24

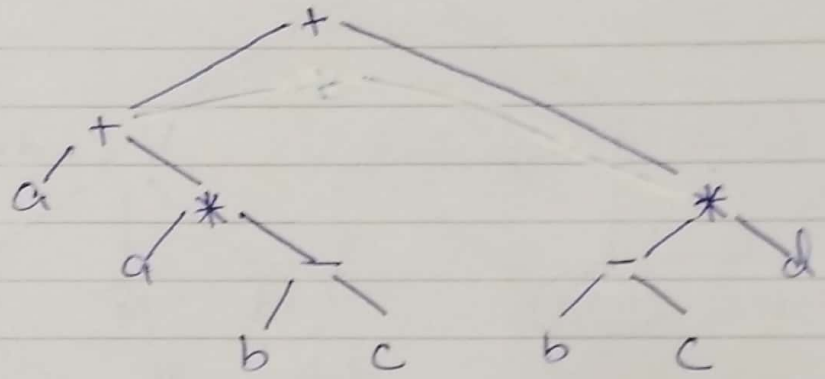


17-1-24

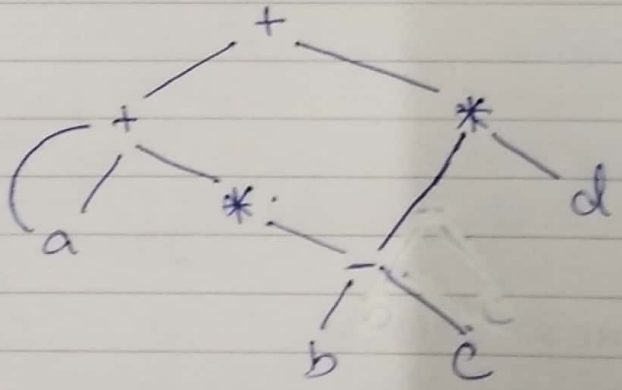
CHAPTER 5
INTERMEDIATE CODE GENERATION



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



Make a Directed Acyclic Graph for this expression
DAG.

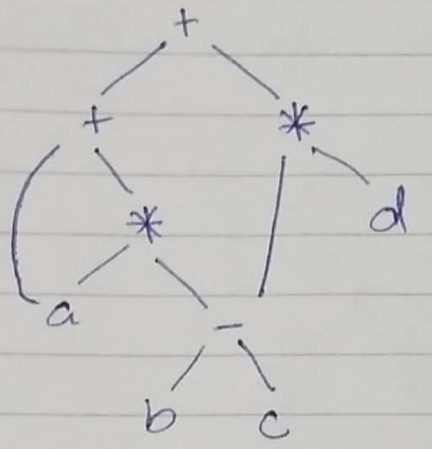


Three Adress Co

Three Address Code:

- A data structure to make intermediate code
- RHS will have at most 1 operator.

DAG ⟶ Three address code.



DAG

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

Three address code

Address & Instructions.
assignment, copy, conditional, procedure.

Example:

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

```
L: t1 = i+1
   i = t1
   t2 = i*8
   t3 = a[t2].
   if t3 <v goto L.
```

(a) Symbolic Labels

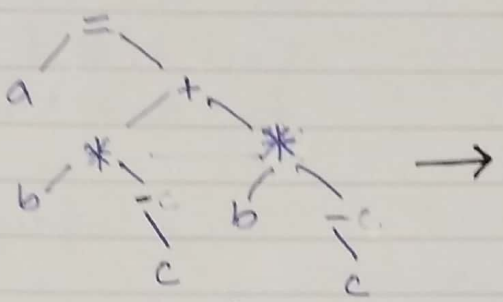
```
100: t1 = i+1
101: i = t1
102: t2 = i*8
103: t3 = a[t2]
104: if t3<v goto 100.
```

(b) Position numbers.

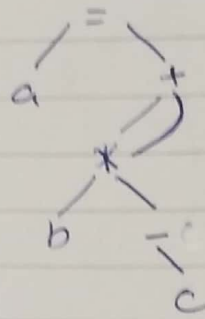
Quadruples:

op	arg1	arg2	result

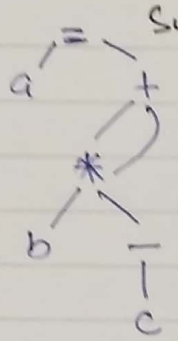
a = b * -c + b * -c ;



Syntax Tree



DAG



DAG

$t_1 = \text{minus } c.$
$t_2 = b * t_1$
$t_3 = t_2 + t_2.$
$a = t_3$

Three-address code

	op	arg1	arg2	result
0	minus	c		t_1
1	*	b	t_1	t_2
2	+	+	t_2	t_3
3	=	t_3		a

Quadruple

	op	arg1	arg2
0	minus	c	
1	*	b	0
2	+	1	1
3	=	a	2

Triples.

Another variant : Indirect tuple.
understant yourself ***

Static Single-Assignment Form, SSA

```
p = a+b
q = p-c
p = q*d
p = e-p
q = p+q
```

```
p1 = a+b
q1 = p1-c
p2 = q1*d
p3 = e-p2
q2 = p3+q1
```

There is possibility that we overwrite a value that has to be used later, So we will increase our space but not time for recalculations.

LECTURE 11

24-01-24.

RUN TIME ENVIRONMENT

Code
Heap
Free Memory
Stack.

Assume array size is 80
Make function call stack and activation tree (Slide 7)

Heap Management:
To store persistent data/variables.
dynamically ~~make~~ stored.

else.

LECTURE 12
CODE GENERATION I.

Main Tasks of Code Generator
- Instruction Selection.

- Registers allocation & assignment.
  registers selection & values assigning.

- Instruction ordering
  schedule the execution of code.

Simple Target-Machine
Expected output
assembly language.

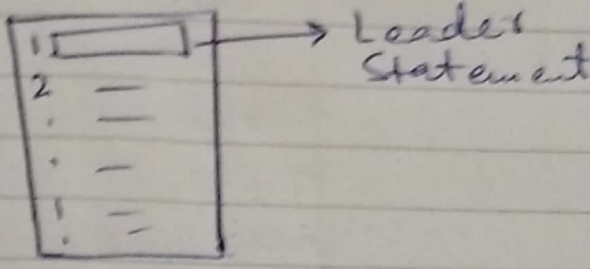
Cost
cost of an instruction = 1 + cost of operand
cost of register operands = 0.
cost of involving memory & constants = 1
cost of program = sum of instruction costs.

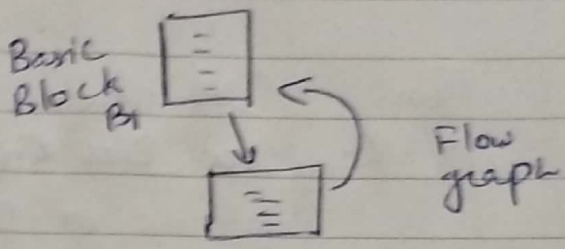
$X = Y - Z$

cost = 4.

1	LD	R_1, Y
1	LD	R_2, Z.
1	SUB	R_1, R_1, R_2
1	ST	X, R_1
4		

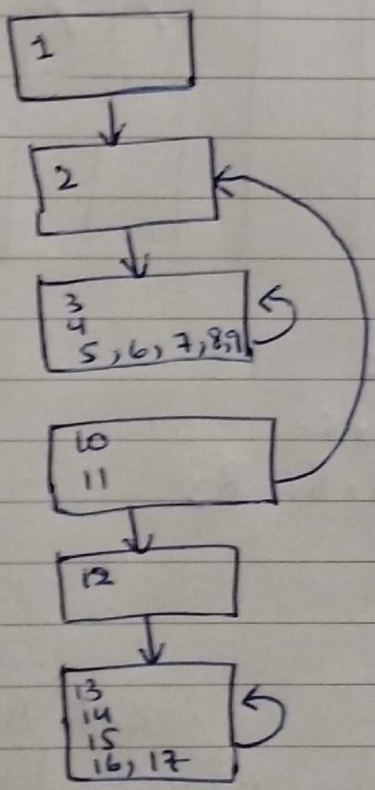
Basic Blocs & Flow Graphs:

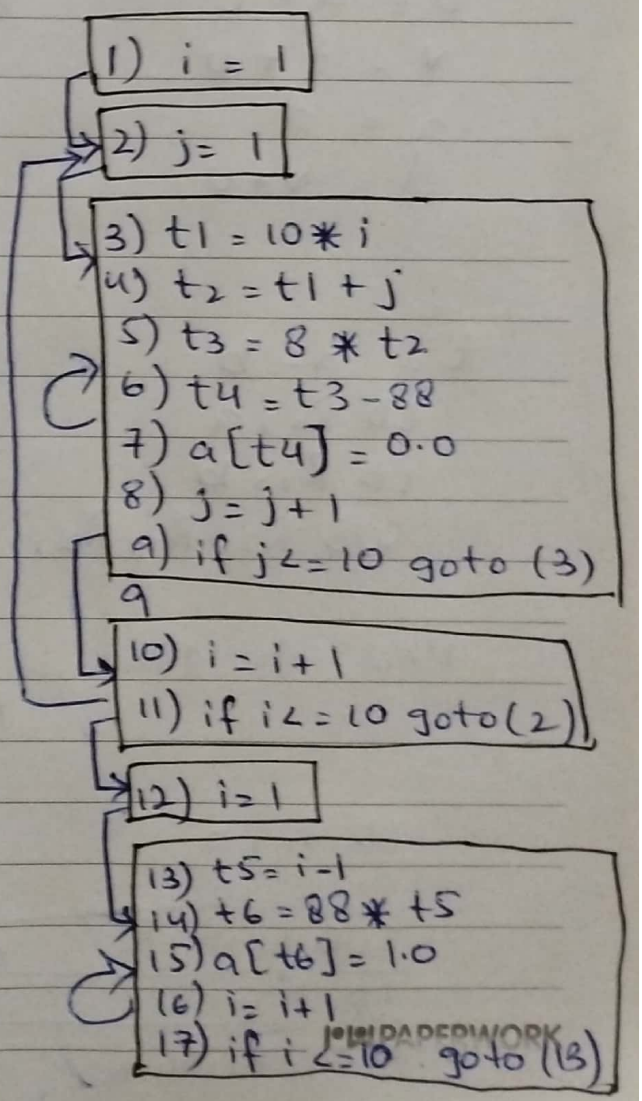




How to determine leader statements

① First three-address instruction is a leader
② Instruction which is target of conditional/unconditional a jump.
③ Instruction that immediately follo conditions / unconditional jump is a leader.





Lecture # 3
Code Generator II.

Two Structures.

① Register Descriptor
(hardware or OS)

② Address Descriptor:
. Compiler.

Example:

$t = a - b$
$u = a - c$
$v = t + u$
$a = d$
$d = v + u$

Reg Desc.

R_1	R_2	R_3
~~u~~	~~t~~	c

Addr Desc.

v	a	b	c	t	u	d
	~~R_1~~	~~R_2~~	R_3	R_2	R_1	

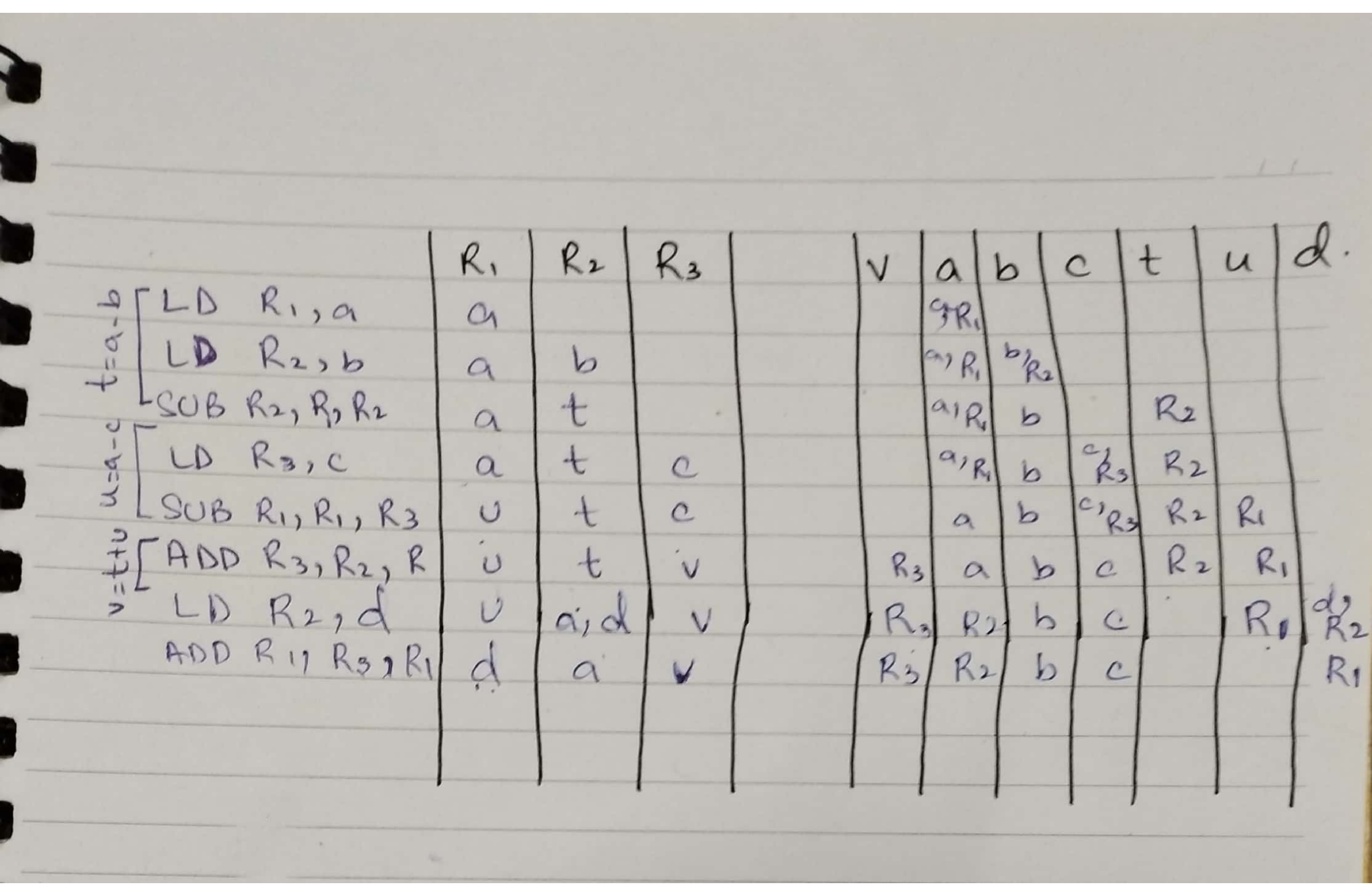
$t = a - b$.
LD R_1, a
LD R_2, b.
SUB R_2, R_1, R_2.

$u = a - c$
LD R_3, c.
SUB R_1, R_1, R_3.

$v = t + u$
ADD R_3, R_2, R_1

$a = d$
LD R_2, d

$d = v + u$
ADD R_1, R_3, R_1


t=a-b
u=a-c
v=t+u
LD R1, a
LD R2, b
SUB R2, R1, R2
LD R3, c
SUB R1, R1, R3
ADD R3, R2, R
LD R2, d
ADD R1, R3, R1
R1 R2 R3
a
a b
a t
a t c
u t c
u t v
u a,d v
d a v
v a b c t u d
a,R1
a,R1 b,R2
a,R1 b R2
a,R1 b c,R3 R2
a b c,R3 R2 R1
R3 a b c R2 R1
R3 R2 b c R1 d,R2
R3 R2 b c R1