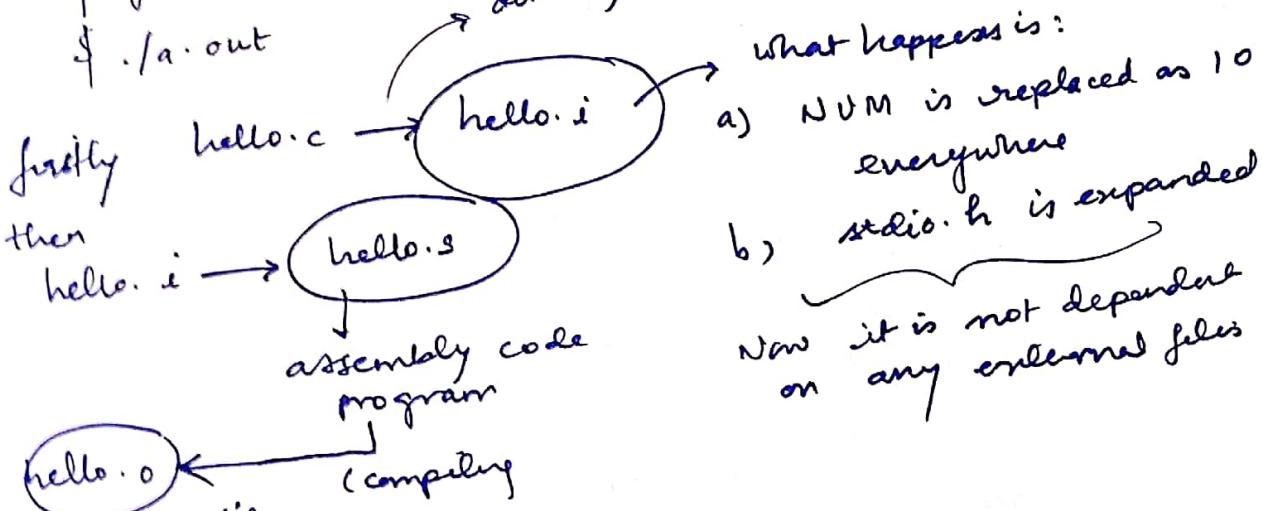
```
# define NUM 10        → macro.
# include <stdio.h>
int main()
{ int a = NUM;
  printf(" .... %d", a);
  return 0;
}
```

∴ $ gcc hello.c
$ ./a.out



Linking : done for math library (we use -lm in ubuntu)
(because all math functions are not defined in the math library)

Loader : Loads .exe file to RAM
e.g. the file when present in secondary memory, might have starting memory address as ffoo. On loading to RAM, the starting memory address might change to, say, 1f2d.
the loader recalculates everything w.r.t the new memory address.

IR - Intermediate Representation ↘ same
TAC - Three Address Code

---

pos = int + rate × 60

↓

Lexical Analysis

↓ Token stream

Syntax Analysis (Parser)

↓ Syntax Tree

Semantic Analysis

↓ Syntax Tree + semantically checked

Intermediate Representation (IR)

Three Access Codes (TAC)

↓

Machine Independent Optimization

↓

Target Code generation

↓

Machine dependent optimization

all are linked to ST (Symbol table)

Lexical Analysis : a | = | b | + | c | × | 60

↳ segregates into various parts (breaks into parts)

also it attaches labels.

e.g.: first part is an identifier (a)
second part is the assignment operator (=)

Syntax Analyser : generates syntax tree.

eg fahrenheit = centigrade * 1.8 + 32

↓ lexical analyser

<id, 1> <assign> <id, 2> <multop> ← multiplication operation

<fconst, 1.8> <addop> <iconst, 32>

↓ floating constant ↓ addition operation ↓ integer constant

$f = c * 1.8 + 32$

$b = a * 10 + a$

$v = a * t + u$

id = id * num + num

id = id * num + id

id = id * id + id

E = E * E + E

(E = (E * E) + E))

e.g. let's say we write

int 2per ; (not allowed as variable name cant start with number)

∴ error provided by lexical analysis.

if you write

a = + - x b error generated by parser!

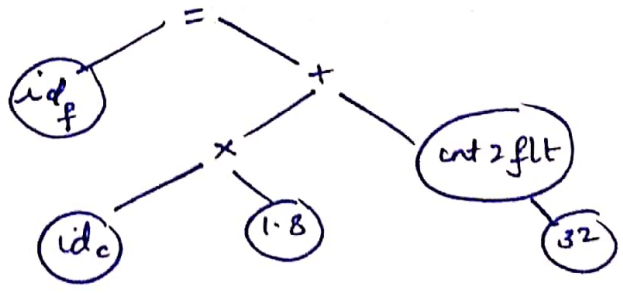
Intermediate code : Atmost 3 addresses involved

thus TAC !!

left to right scan for expressions.

e.g. $f = c \times 1.8 + 32$

NOTE: DO NOT OPTIMIZE the TAC



$t_1 = c$

$t_2 = t_1 \times 1.8$

$t_3 = (int\ 2\ flt)\ 32$

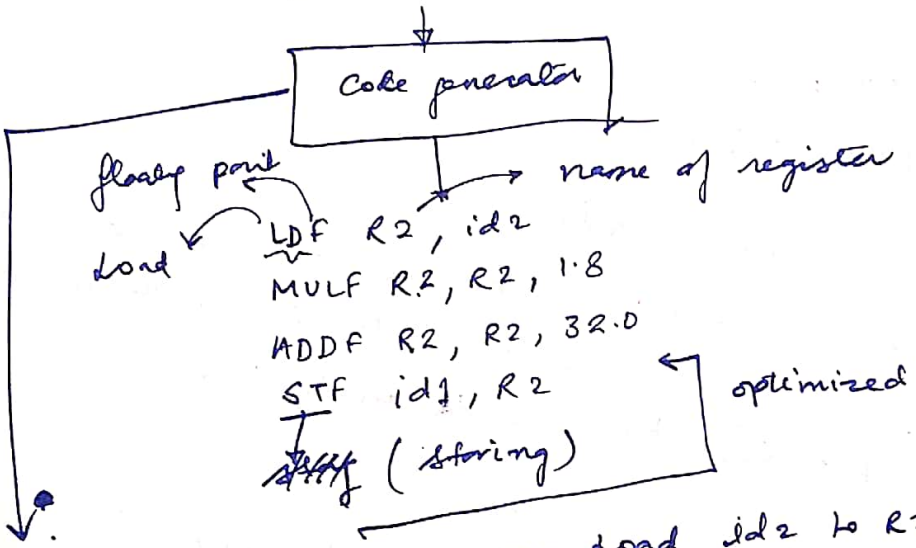
$t_4 = t_2 + t_3$

$f = t_4$

Target code generation :

Load variable to register
perform operations on the register

eg
$t1 = id2 * 1.8$
$id1 = t1 + 32.0$




```
LDF   R2, id2      ---- load id2 to R2
MULF  R2, R2, 1.8  ---- multiply 1.8 with R2
STF   t1, R2       ---- store value of R2 in t1 (as t1 is already stored in R
LDF   R3, t1       } redundant
ADDF  R3, R3, 32.0 } similarly
STF   id1, R3
```

not optimized yet

Python is an interpreter language. There is no exe file the source code is interpreted line by line as the program runs.

```
./configure
make
make install
make clean
```

OS: GNU / Linux, 64-bit, X86_64

Software: GCC, lex/flex and Yacc/Bison

Language: C++

Commands:

Hardware system info.

$ uname -a

$ cat/proc/cpuinfo

Main memory address:

Address: 36 bits physical, 48 bits virtual/logical

2^{36} bits

GPR → general purpose registers

Intel 64-bit registers:

GPRs: 64-bit, 16 in no.
rax, rbx, rcx, rdx, rsp, rbp, rsi, rdi, r8 ... r15
rbp → base pointer
rsp → stack pointer

FPR: 80 bit floating point register

MMX: 64 bit SIMD registers

XMM: 128 bit SSE registers (16)

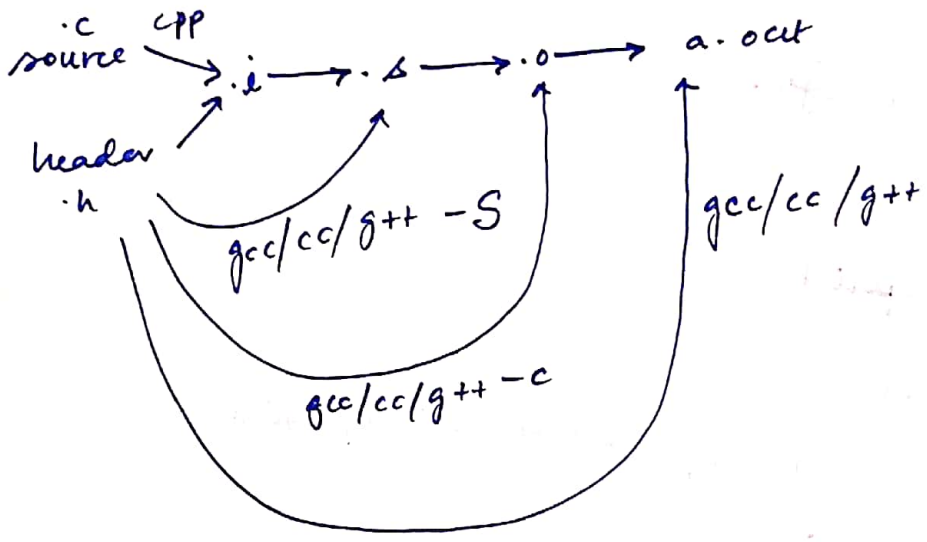
SSE: Streaming SIMD extension (16)
xmm0, ..., xmm15

```
0000  int main()
      {
0010    x = foo();
0011    jump 2000;
0100  }

2000  int foo()
      {
      ...
      return a;
      }
2020  jump 0011;
```

Static executable file vs. dynamic executable file

↳ everything loaded at once



Infix ⟶ Postfix (RPN) (Reverse Polish Notation)

consider an expression

((9 + 5) − 4)

(9 + (5 × 4))

We can get an algorithm such that with only 2 left-to-right movements we can evaluate any expression irrespective of everything.

1) ×, /
2) +, −
3) <, >, <=, >=, ==, ≤, ≥
4) =, ternary operator (a>b)? c : d; ← right associative right to left (⟸)

associative from left to right

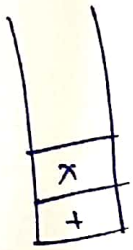
mean a+b+c = ((a+b)+c) ⟶

keep adding operators to stack if current operator is > operator on top of stack, else pop.

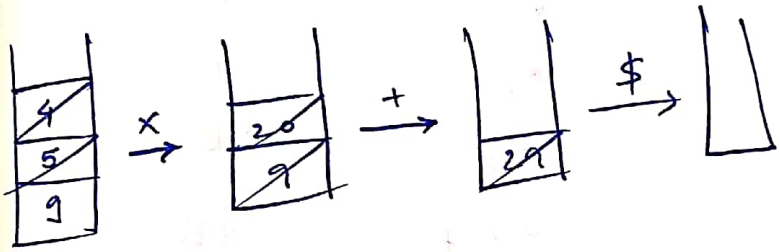
9 + 5 × 4 $

9 5 4 × + $

postfix



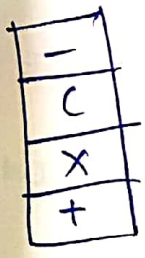
now next operator add the numbers to a stack in post-fix keep pushing until you reach an operator, if binary pop the 2 top operands, perform operation & push back.



9 + 5 × (3 − 4) $

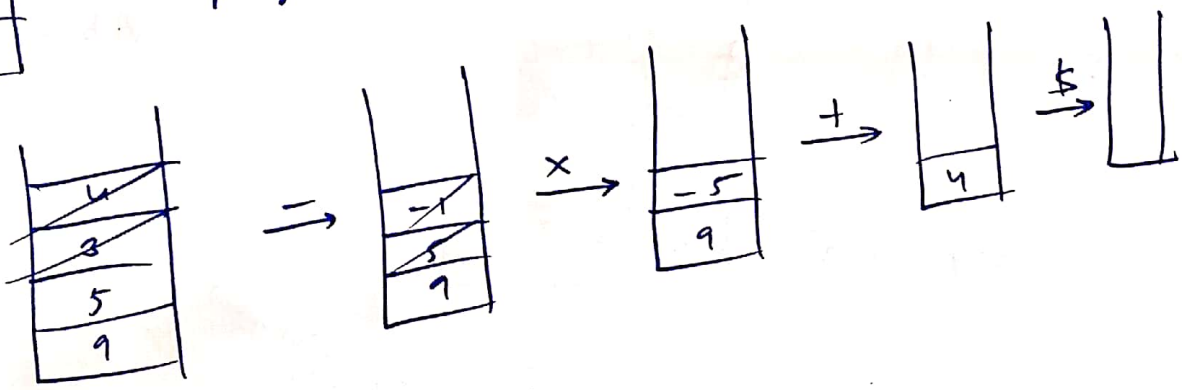
(whenever you see an opening parenthesis, just push it, no further action)

(when you see a closing parenthesis you pop until you find an opening parenthesis)



9 5 3 4 − × +

postfix.



Since '+' & '-' are left associative and equal precedence

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

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

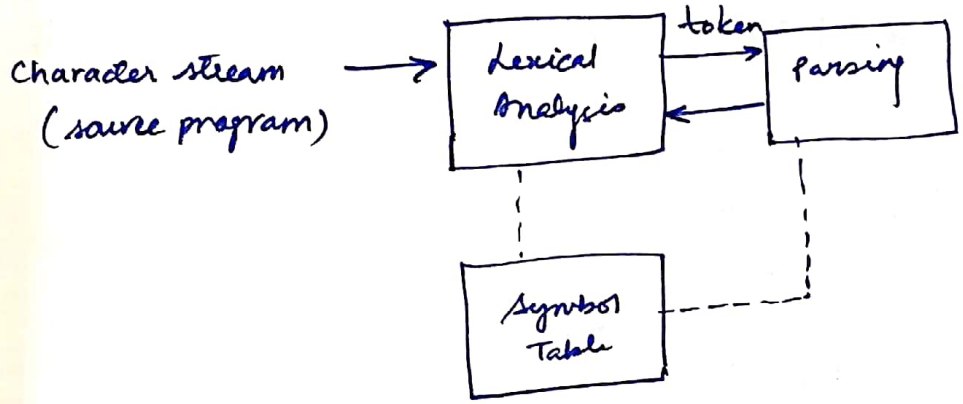
$a+b-c = (a+b)-c$

Expression

$A*(B+C*D)+E$

Current Symbol	Operator stack	Postfix string
A		A
*	*	A
(	*(	A
B	*(	AB
+	*(+	AB
C	*(+	ABC
*	*(+*	ABC
D	*(+*	ABCD
)	*	ABCD*+
+	+	ABCD*+*
E	+	ABCD*+*E
$		ABCD*+*E+

expression = syntax tree



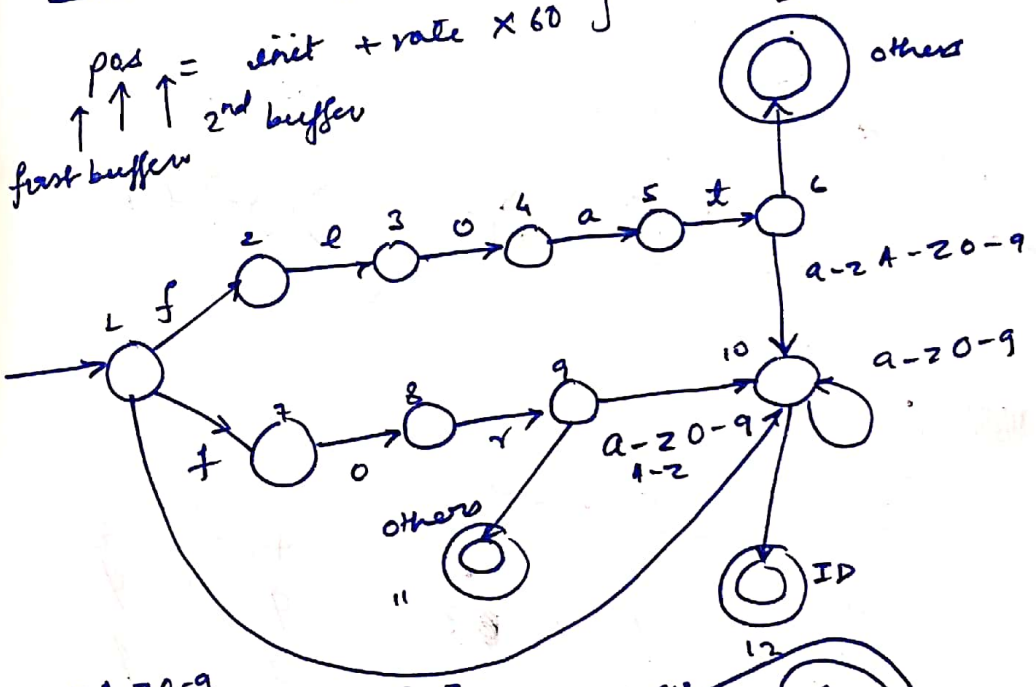
Lexeme
Token
Token name , Attribute value

Sentinel required to mark end of token name!!
- read by 2nd buffer.

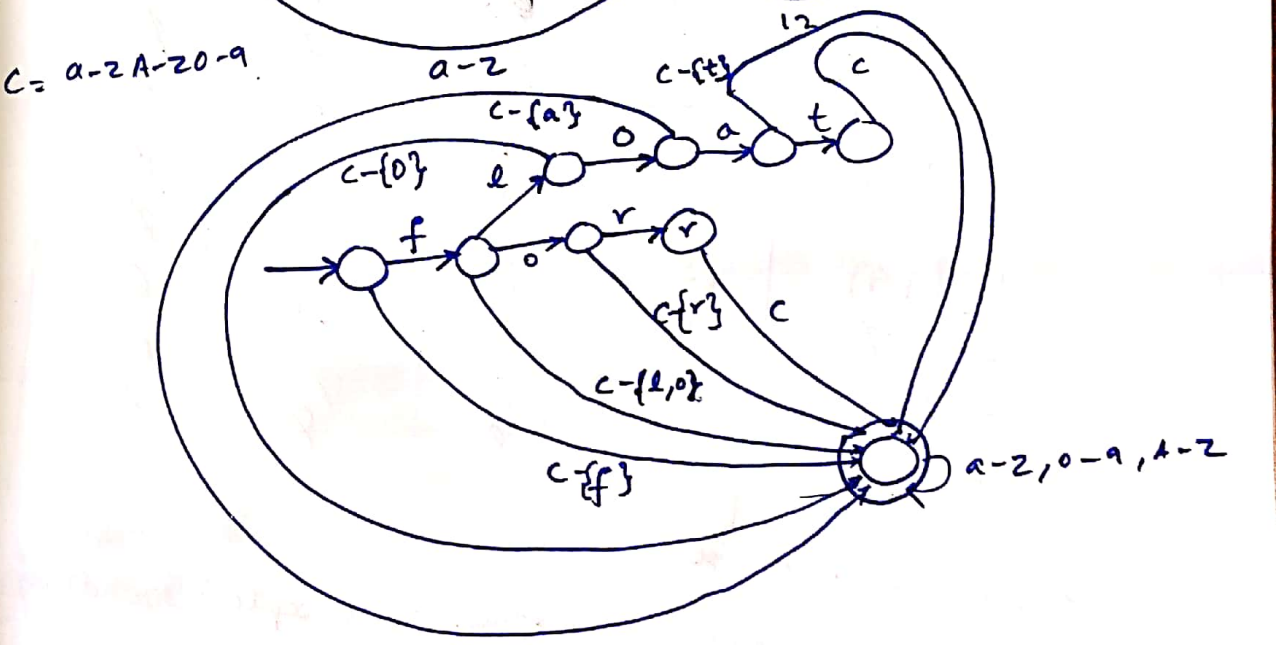
Double buffering :

pos = init + rate × 60 } lexeme

first buffer | 2nd buffer



C = a-z A-Z 0-9



Sl. no.	id	Memory	Type
1	pos	→	int
	init	→	
	rate	→	
	60	→	

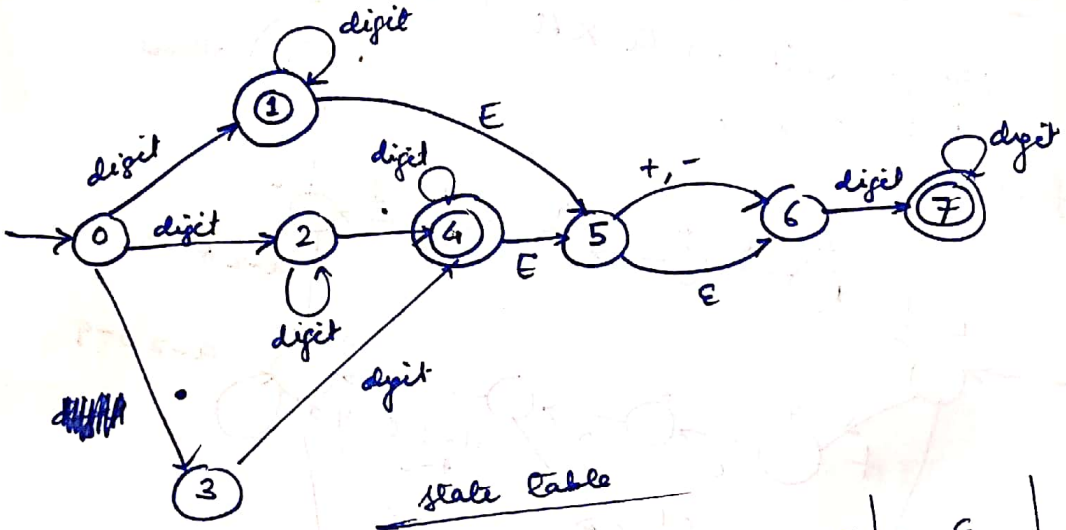
<(ID, link to ST)>

Character class :

id → letter (letter | digit)*

letter → a|b|c|...|z|A|B|C|...|Z

digit → 0|1|2|...|9

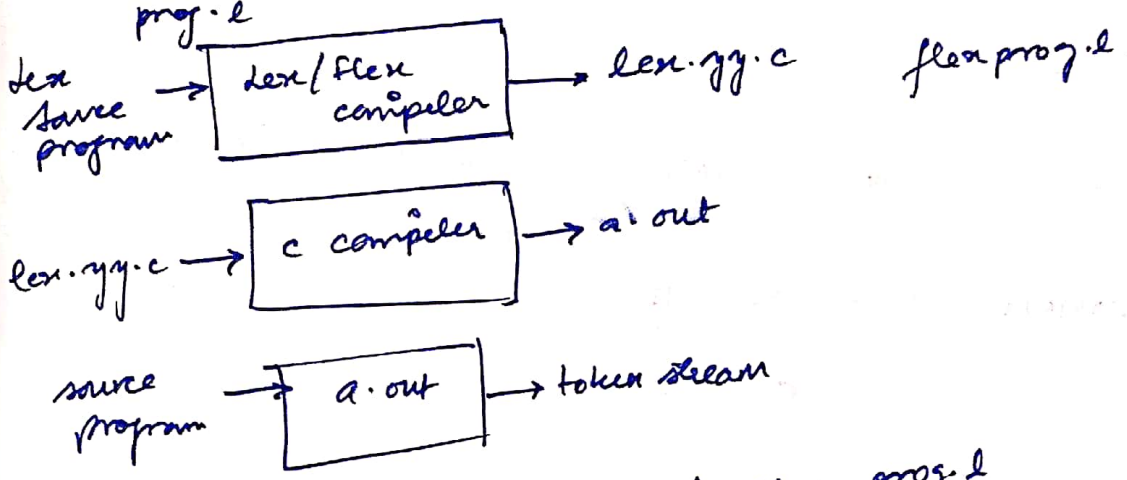


State Table

	digit	E	.	+, -	ε
0	{1,2,3}	ϕ	{3}	ϕ	ϕ
1	{1}	{5}	ϕ	ϕ	ϕ
2	{2}	ϕ	{4}	ϕ	ϕ
3	{4}	ϕ	ϕ	ϕ	ϕ
4	{4}	{5}	ϕ	{6}	{6}
5	ϕ	ϕ	ϕ	ϕ	ϕ
6	{7}	ϕ	ϕ	ϕ	ϕ
7	{7}	ϕ	ϕ		

ϕ are all error states :

we can write warning messages for exception handling




```
{  int x;
   int y;
   x = 2;
   y = 3;
   x = 5 + y * 4;
}
```

```
$ flex prog.l
$ cc lex.yy.c -lfl
$ ./a.out        C-executable file
```

./a.out <

flex specs:

```
INT     "int"
ID      [a-z][a-zA-Z0-9]*
PUNC    [;]
.
.
.
{INT}   { printf ("<KEYWORD, int>\n"); /* keyword rule */ }
{ID}    { printf ("<ID, %s>\n", yytext); /* identifier rule */ }
```

note we can replace {ID} with [a-z][a-zA-Z0-9]* and so on!

∴ we get → token name, attribute value

<SPECIAL SYBOL, {>
<KEYWORD, int>
<ID, x>
<PUNCTUATION, ;>

note:

{ID} is above {INT} for a reason, find out!

COMPILERS LAB :

#

#include

(decimal) → (Hexadecimal) → (ASCII)

6E 6F 63 65 73 20 79 6D
n o c e s ⇔ y m

61 72 67 6F 72 70 20 64
a r g o r p ⇔ d

NOTE: cpp is a pre-processor :

Preprocessor – Assembler – Linker:

def college n = 1234

Str[5] = "1234";

To get a string from an integer repeatedly

n = n%10

n = n/10

add in an array.

```
# include <stdio.h>
# define MAX 10
# include <stdio.h>
```

→ no error as these lines are included

```
# ifndef _MYPRINT_H
# define _MYPRINT_H
(if not defined, define)
```

```
#include<stdio.h>
void getnum ();
# define MAX 10
void getnum ();
```

→ error on linking

Creating a library :

```
$ C++ -Wall -c printInt.c++
$ ar -rcs libprintInt.a printInt.o        → generates archive file
$ C++ -Wall -c mainPrintInt.c++
$ C++ mainPrintInt.o -L. -lprintInt
$ ./a.out
$ Enter an integer : -123
$ -123
```

↑ NOTE here we need to manually link.

to remove this problem, construct a makefile, do not need to repeatedly compile.

make : checks the timestamps of all prerequisite files makes sure they are less than that of a.out

```
objdump -d a.out
```

Flex specs

```
{word}    { wordcount++; charcount += yyleng; }
[\n]      { charcount++; linecount++ }
.         { charcount++; }
```

Start condition in Flex specs :

- Declaration :
- Begin action : activated using BEGIN
- Inclusive : "%s"
- Exclusive : "%x"

PARSER

Grammar = $< T, N, S, P >$ is a context free grammar.

T : Terminals
N : Non terminals
S : $S \in N$ (start symbol)
P : Set of production rules

example:

$G = <\{id, +, *, (,)\}, \{E, T, F\}, E, P>$ where P is

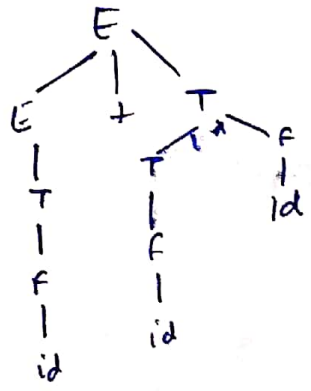
1. $E \rightarrow E+T$
2. $E \rightarrow T$
3. $T \rightarrow T*F$
4. $T \rightarrow F$
5. $F \rightarrow (E)$
6. $F \rightarrow id$

Let - most derivation of $id + id * \$$: ($\$$ → end of line)

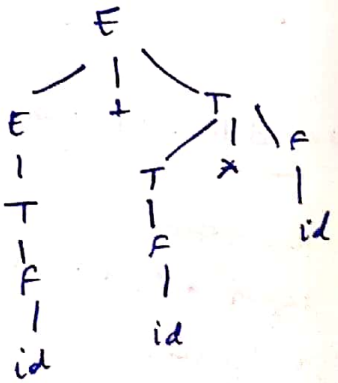
Augmented grammar, has one extra derivation:

$E' \rightarrow E\$ \Rightarrow E+T\$ \Rightarrow T+T\$ \Rightarrow F+T\$$
$\Rightarrow id+T\$ \Rightarrow id+T*F\$ \Rightarrow id+F*F\$$
$\Rightarrow id+id*F\$ \Rightarrow id+id*id\$$

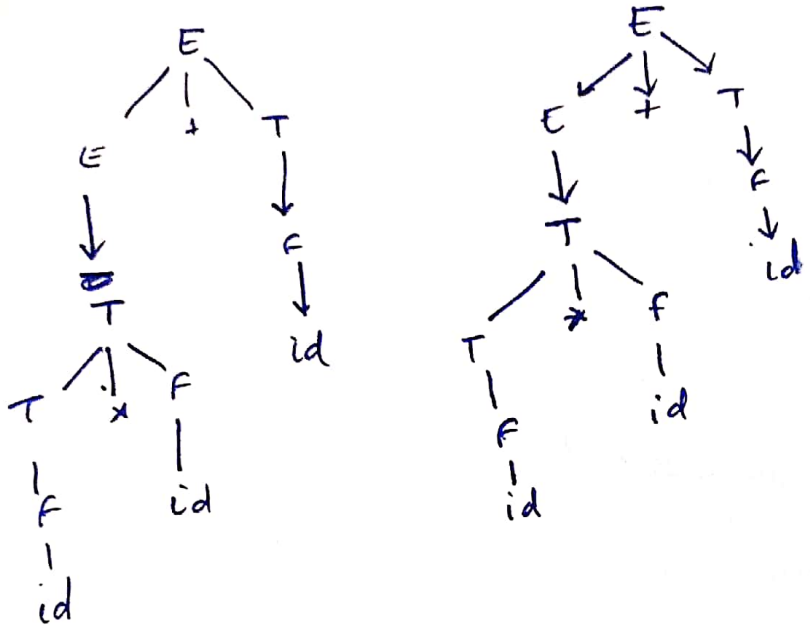
Leftmost derivation



Rightmost derivation



L.D.T for id * id + id



Derivation — Parsing — left recursive

leftmost — top-down — LL(1) (1 lookahead)

shift reduce

rightmost

RL - Right to left, leftmost derivation first } X not preferable

RR - Right to right, rightmost derivation first }

LL - preferable (top-down)

LR - preferable (bottom-up)

Recursive descent parser

$S \rightarrow cAd$
$A \rightarrow ab \mid a$

```
int main() {
    l = getchar();
    S();
    if (l == '$')
        printf ("successful")
    else printf ("Error")
    }
S() {
    match ('c');
    A();
    match ('d')
    }
A() {
    match ('a');
    if (l == 'b') {match ('b');}
    }
match (char t)
    if (l == t) { l = getchar(); }
    else printf ("Error")
```

example slip :

(i) cad $

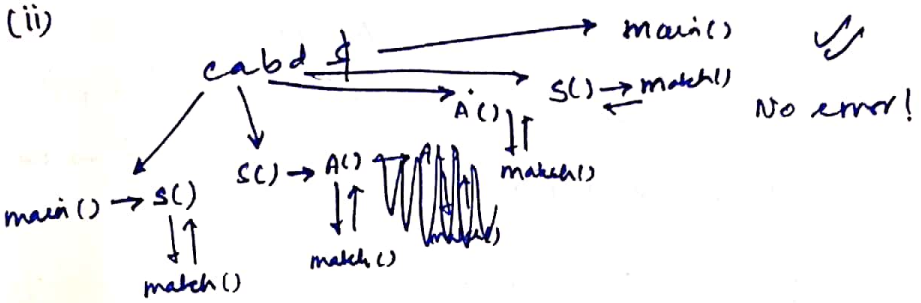
$a() \rightarrow s()$ now l has $ ∴ success.

$s() \rightarrow a()$

$main() \rightarrow s()$

A → f(A calls B)

(ii)



(iii) caad $

gives error!

A grammar is left recursive iff there exists a ...

Both left & right recursion are both problems as we do not know where to stop.

Parsers :

Top - down :

Bottom - up : Shifting (Read next symbol)
Reducing (Reduce the right side of the production)

Definitions :

Handle : A substring that matches the body of the production & whose reduction represents one step along the reverse of rightmost derivation !

Reliable prefix : A reliable prefix is a prefix of a right sentential - form that does not properly extend beyond the handle.

rules:

1 $E \rightarrow E + T$
2 $E \rightarrow T$
3 $T \rightarrow T * F$
4 $T \rightarrow F$
5 $F \rightarrow (E)$
6 $F \rightarrow id$

Step	Stack	Symbol	Input	Action
	0		id*id + id$	shift
	05	id	*id + id$	reduce by F → id
	03	F	*id + id$	
			⋮	

see from slides :

An LR parser is a DPDA.

Initial LR Parser construction :

$G_3 = \{S \to a\}$

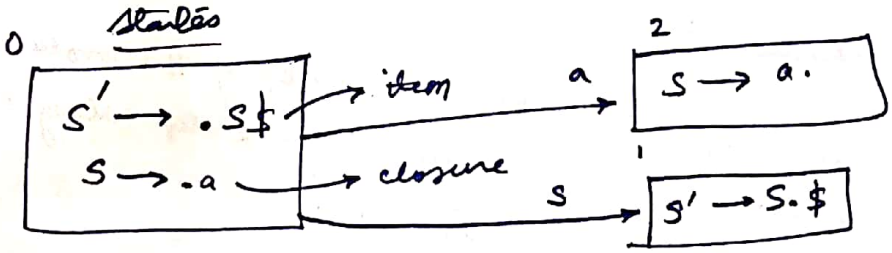
$G_3 = (\{S\}, \{a\}, P, S)$

Rules

S' → S — rule
S → a — rule

Production rules : $S' \to S\$$

$S \to a$



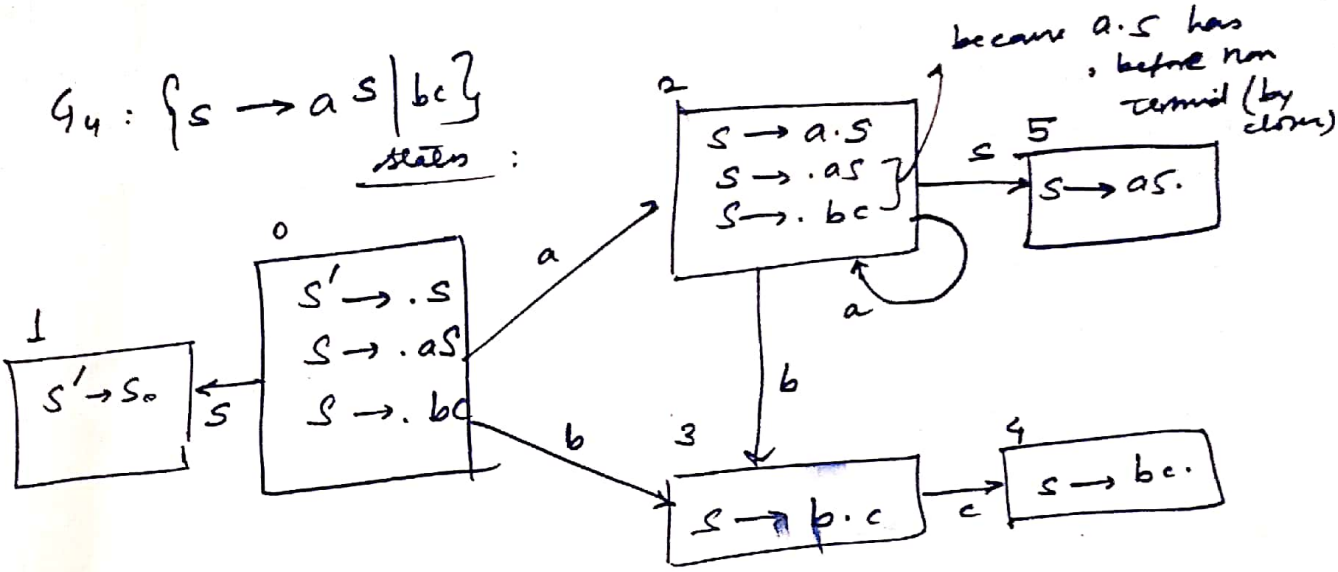
Closure property: if . is on left of some non-terminal, include all rules in which S derives to some other thing, in that state.

State	a	$	S
0	S2		1
1		Acc	
2	r1	r1	

S = shift
R = reduce

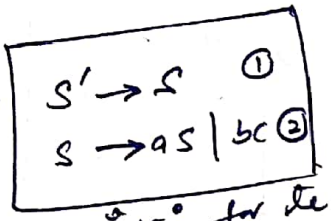
G_4 : $\{S \to aS \mid bc\}$

states :



NOTE: $ is implicit :

Rules:



scanning for terminal

state	a	b	c	$	S
0	s2	s3			1
1				Acc	
2	s2	s3			5
3			s4		
4	r2	r2	r2	r2	
5	r1	r1	r1	r1	

go-to (for non terminal)

also in go-to there is no change in the position of '.' while there is a change in scanning.

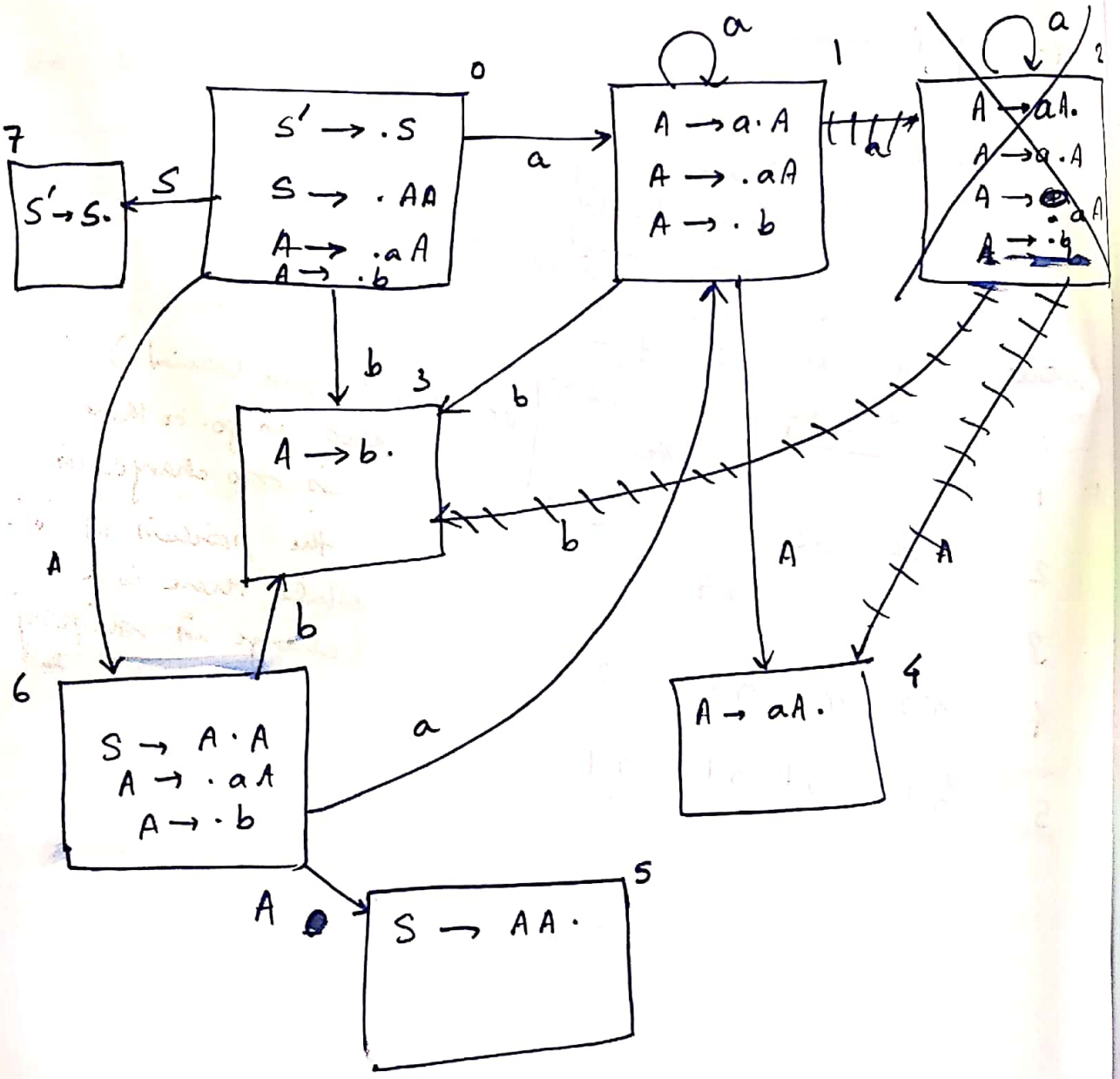
Construct an LR(0) parser for G7

1 : $S \rightarrow AA$

2 : $A \rightarrow aA$

3 : $A \rightarrow b$

add : 0 : $S' \rightarrow S$



State	a	b	$	S	A	B
0	s1	s3		g		

Reduce a production $S \to \ldots$ on symbols $K \in T$ if $K \in$ follow(S)

No. of pops : no. of terminals + non terminals on the right of the production rule, used for reduction.

Construct an SLR(1) parser for G8

1: $S \to E$

2: $E \to E + T$

3: $E \to T$

4: $T \to T * f$

5: $T \to f$

6: $f \to id$

LALR(1) has reduce-reduce conflict:

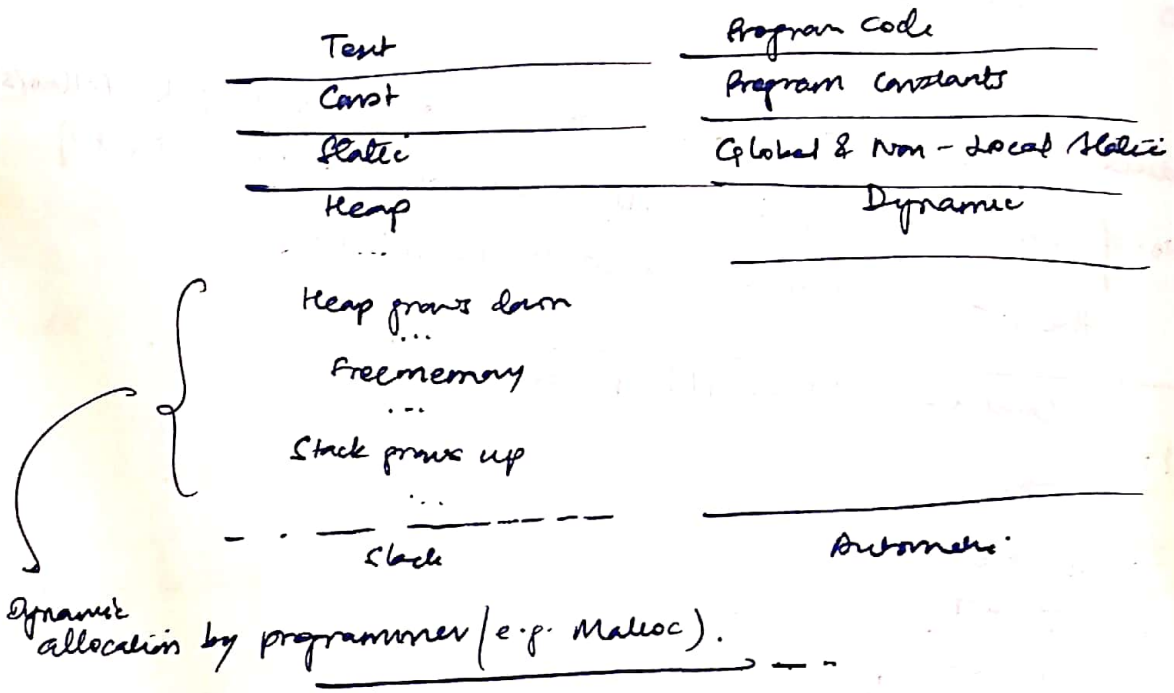
6: $A \to c.\ ,\ d$
$B \to c.\ ,\ e$

7: $A \to c.\ ,\ e$
$B \to c.\ ,\ d$

8: $A \to c.\ ,\ d/e$
$B \to c.\ ,\ e/d$

LL parsers are a subset of LR parsers

Storage organization :




```
f(int n)
{
  .
  .
  return v;
}
int main()
{
    int a;
    int b;
    b=f(a);
    b=f(4);
}
```

symbol tables: a) for global things
b) for main
c) for functions called

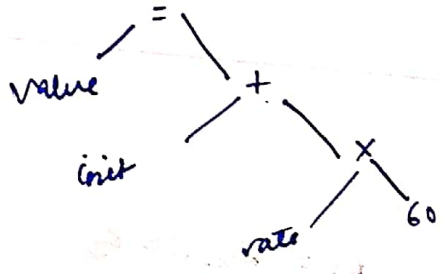
esp – stack pointer
ebp – base pointer
eip – instruction pointer
eax
ecx
:
and so on.

---

you missed some notes..

---

value = init + rate * 60

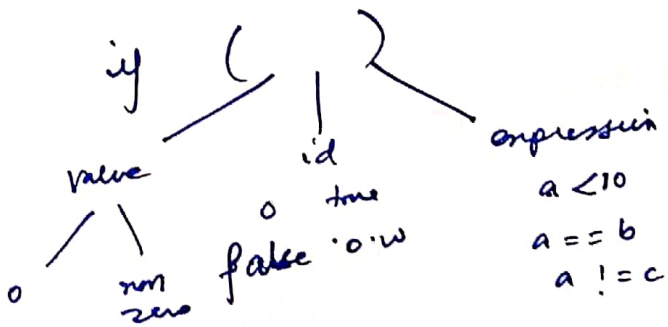


rate' = rate * 60
temp = init + rate'
value = temp

Address types

name, const.., compiler generated Temporary.
↓ ↓ +
source program, names diff. types {here rate' & temp}

Conditional statements:



x = y[i]

x = *(y + i x bytes)

x = y[i x bytes]

depends on type of y (for int 4 bytes/2 bytes for float 4 bytes)

Symbol table :

&

lex

par

IR

MC

MICO

MDCO

[\n]

= "\n"

[0]

"0" | "1"

Boolean expression :

$B_1 = B_2 \,||\, B_3$

first check B_2

If B_2 is true
B_1 is true

else if B_3 is true
B_1 is true

else
B_1 = false

$B_1 = B_2 \,\&\&\, B_3$

if B_2 is false
B_1 is false

else if B_3 is true
B_1 is true

Control Constructed grammar :

$S \rightarrow \{L\}$

$S \rightarrow id = E ;$

$S \rightarrow if\ (B) S$

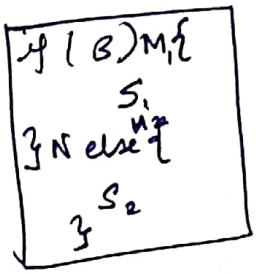
$S \rightarrow if\ (B) S\ else\ S$

$S \rightarrow while\ (B) S$

$L \rightarrow LS$

$L \rightarrow S$

S



S.next

if: M_1 make a decision (based on B)

execute S_1 skip S_1

2 decisions therefore M_1, N_1

make decision

execute S_2 / skip S_2

while loop: S

became check B

make decision

don't execute S

skip to next piece of code

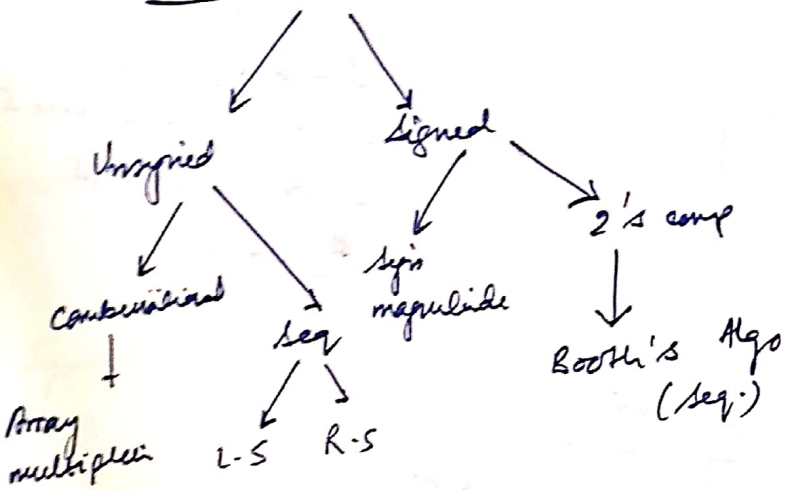
execute S

make decision

skip to next piece of code

reexecute loop

Binary Integer multiplication



Unsigned multiplication :

$$X = x_{n-1} x_{n-2} \ldots x_0 = \sum_{j=0}^{n-1} x_j 2^j$$

$$Y = y_{n-1} y_{n-2} \ldots y_0 = \sum_{j=0}^{n-1} y_j 2^j$$

$x_i, y_i \in \{0,$
$\forall\ i \in [0,$

$$P = X * Y = \sum_{i=0}^{n-1} 2^i x_i Y = \sum_{i=0}^{n-1} \left[\sum_{j=0}^{n-1} x_i \cdot y_j 2^{j+i} \right]$$

$y_{n-1}\ y_{n-2} \ldots y_1\ y_0$

$x_{n-1}\ x_{n-2} \ldots x_1\ x_0$

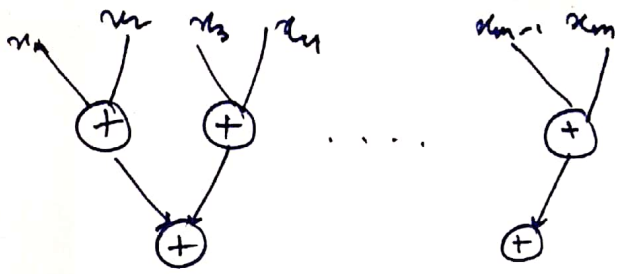
$x_0 y_{n-1}$ $\quad x_0 y_2\ x_0 y_1\ x_0 y_0$

$\quad\quad\quad\quad\quad x_1 y_1\ x_1 y_0$

$\quad\quad\quad\quad\quad\quad x_2 y_0$

1. FA $\quad$ HA $\quad$ and

} pure combinational circuit



m numbers
each n-bit long

$$O\left(\lg(n) + \lg\left(\frac{\quad}{\quad}\right)\right.$$

$\oplus$

signed magnitude mul.

$X = x_s x_{n-2} x_{n-3} \cdots x_0$

$Y = y_s y_{n-2} y_{n-3} \cdots y_0$

$P = P_s P_m$ & P_m we can calculate

$P_s = x_s \oplus y_s$

Unsigned sequential multiplication:

P: $2n$-bit register

1) left-shift version:

in $(i+1)^{th}$ iteration, calculate:

$$P_{i+1} = P_i + x_i 2^i Y, \quad P_0 = 0$$

then $\boxed{P_n = P}$ prove

2) right-shift version

$$P_0 = 0, \quad P_{i+1} = (P_i + x_i Y 2^n)2^{-1}, \quad 0 \le i < n$$

with $\boxed{P_n = P}$ prove

x_0 terms: L.S by n, R.S. by n $\Rightarrow$ zero shift

x_1 terms: L.S by n, R.S by $(n-1)$ $\Rightarrow$ 1 bit left shift