07-02-2022 Unit-1 & 2

Phases of Compiler

Lexical Analysis

Syntax Analysis

Semantic Analysis -> type checking

Intermediate Code Generator

Code ~~Generator~~ Optimization

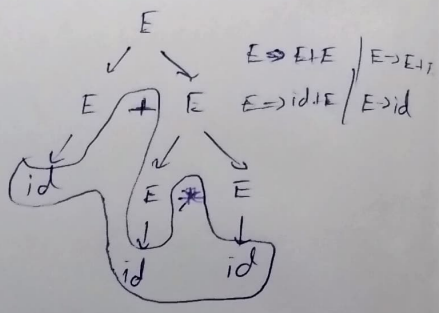
Code Generator

Three types of Intermediat code

Postfix Notation c = a+b → cab+=

Syntax tree E → E+E : id+id
E → E+E
E → id

Three address code



c = (a - b) * d

$T_1 = a - b$

$T_2 = T_1 * d$

$c = T_2$

MOV a, R1
MOV b, R2
SUB R_1, R_2
MOV d, R_3
MUL R_2, R_3
MOV R_3, c

Intermediate code for unit-3

$E \rightarrow E_{(1)} + E_{(2)}$ {E.val = E¹.val + E².val}

Concatenation

{E.code = E¹.code || E².code || op}

E ⇒ id {semantic action}

semantic Analysis
Intermediate code Generator } -> unit 3

08-02-2022

UNIT-3

Syntax Directed Translation (SDT)

-> SDT allows semantic actions to be attached to the productions of context free grammer.

-> It enables the compiler Designer to express the generation of intermediate code directly in terms of the ~~sem~~ syntatic

structure of source language.

productions + Semantic Actions → SDT

the syntatic actions can be

i) Computation of values of variables
ii) generation of intermediate code
iii) Printing of an error
iv) Placement of a value in a simple table

Translation/Attribute

A value associated with a grammer symbol is called the translational of that symbol.

Eg: X.Val, X.True

Synthesis Translation

It defines the value, the value of the translational of the non-terminal of the left side of the production as a function of the translations of the non-terminals of the right side.

Eg: A → XY {A.Val = X.Val * Y.Val}

Inherited translation

The translation of a non terminal of the rightside of the production is defined in terms of the translation of the non terminal of left side

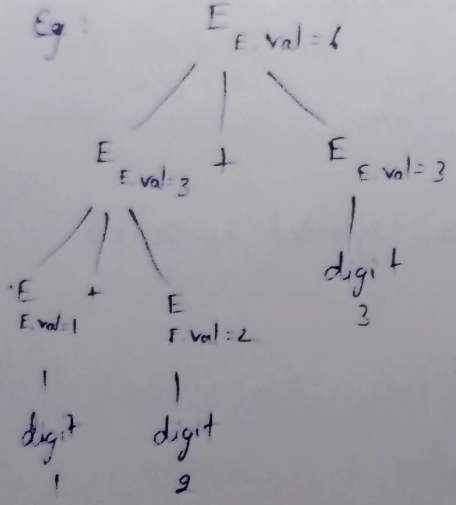
Eg: A → XY {Y.Val := 3* A.Val}

Annotated Parse tree

A Parse tree showing the values of the attributes at each node is called an Annotated Parse tree

The process of computing the attribute values at the nodes is called Annotating or decorating

Eg:

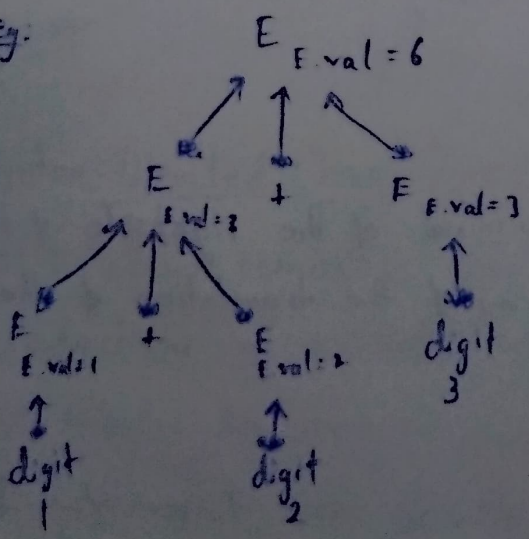


Dependency graph

→ Semantic rules set up dependencies between attribute which can be represented by a dependency graph.

→ The dependency graph determines, the order of evaluation of the semantic rules.

Eg:



Desktop Calculator

(Syntax Directed translation for Desktop Calculator)

$S \rightarrow E\,\$$ { print E.val }

$E \rightarrow E^{(1)} + E^{(2)}$ { E.val := $E^{(1)}$.val + $E^{(2)}$.val }

$E \rightarrow E^{(1)} * E^{(2)}$ { E.val := $E^{(1)}$.val * $E^{(2)}$.val }

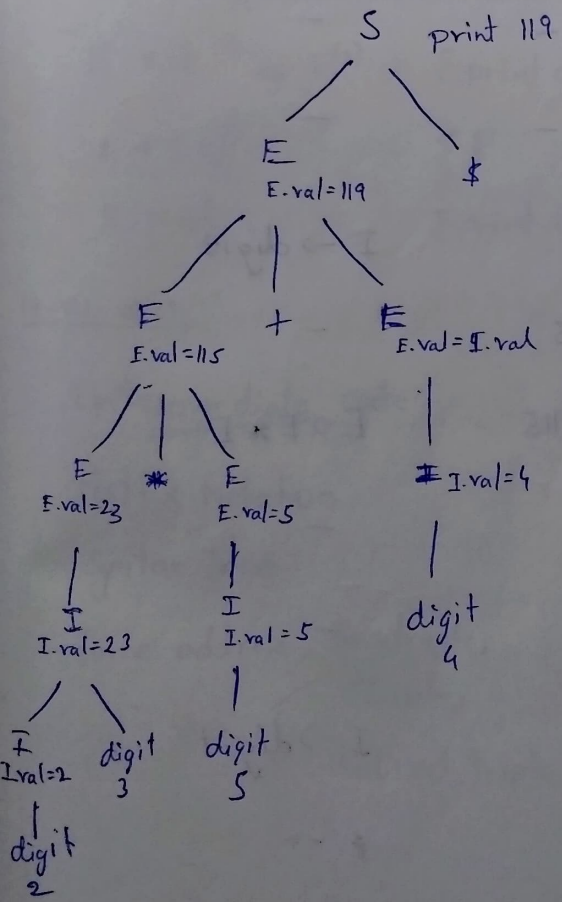
$E \rightarrow (E)$ { E.val := $E^{(1)}$.val }

$E \rightarrow I$ { E.val := I.val }

$I \rightarrow I\ digit$ { I.val = 10 * I.val + lexval }

$I \rightarrow digit$ { I.val = lex val }

23 * 5 + 4



print val[top]

val[top] = val[top] + val[top-1]

val[top] = val[top] * val[top-1]

val[top] = val[top-1]

none

val[top] = 10 * val[top] + lexval

val[top] = lexval

09-02-2022

Implementation of Desktop calculator -

Stack Sequence of moves

23 * 5 + 4

Input	State	Value	Production used
23 * 5 + 4 $	2	_	_
3 * 5 + 4 $	I	2	I -> digit
* 5 + 4 $	I 3	2 _	_
* 5 + 4 $	I	(23)	I -> I digit
* 5 + 4 $	E	(23)	E -> I
5 + 4 $	E *	(23) _	_
+ 4 $	E * 5	(23) _ _	_
+ 4 $	E * I	(23) _ 5	I -> digit
+ 4 $	E * E	(23) _ 5	_
+ 4 $	E	(23) * 5 115	E -> E * E
4 $	E +	115 _	_
$	E + 4	115 _ _	_
$	E + I	(115) _ 4	I -> digit
$	E + E	(115) _ 4	$ —
$	E	119	E -> E + E

Intermediate code

- Post fix Notation
- syntax tree
- Triple
- Quadruple

Post fix Notation

a * (b + c) — abc + *

(a + b) * c — ab + c *

(a + b) * (c - d) — ab + cd - *

if e then x else y — e x y ?

Production

Production	Semantic Action
$E \to E^{(1)}$ op $E^{(2)}$	{E.code = E'.code || E².code || op}
$E \to (E^{(1)})$	{E code = E' code}
$E \to id$	{E code = id}

if a then if a < b then c + d else c * d else a + b

a ab < cd + cd * ?

ab < ? ab + ?

Semantic Action

Implementation of Postfix Notation

$E \to E^{(1)}$ op $E^{(2)}$	{print op}
$E \to (E^{(1)})$	{ }
$E \to id$	{print id}

14-02-2022

Intermediate code

Postfix Notation

Syntax Tree

Three address, Quadruple,
Triple,
Undred Triple

Post fix Notation

$E \to E^{(1)}$ op $E^{(2)}$ {E code = E' code || E' code || op} — translation

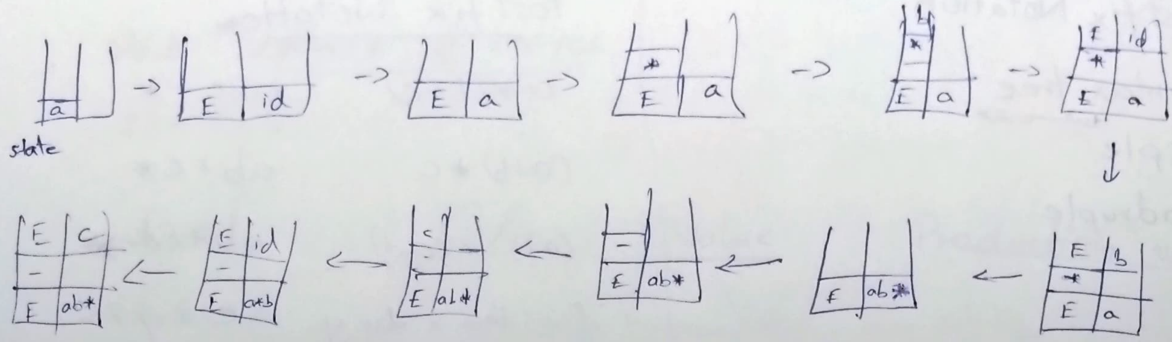
$E \to (E)'$ {E code = E' code}

$E \to id$ {E code = id}

a * (c - d) — (a + b) - (c / d)

a c d - * — a b + c d / -

a*b - c



ab*c -

Implementation

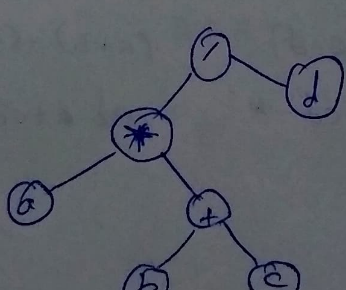
$E \rightarrow E^1 \; op \; E^2$ {print op}

$E \rightarrow (E^1)$ {}

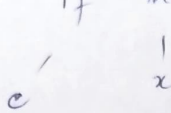
$E \rightarrow id$ {print id}

Production	Semantic Action
$E \rightarrow E^1 \; op \; E^2$	{E.val = NODE (op, E^1.val, E^2.Val)}
$E \rightarrow (E^1)$	{E.val = E^1.val}
$E \rightarrow - E^1$	{E.val = UNARY(-, E^1.val)}
$E \rightarrow id$	{E.val = Leaf(id)}

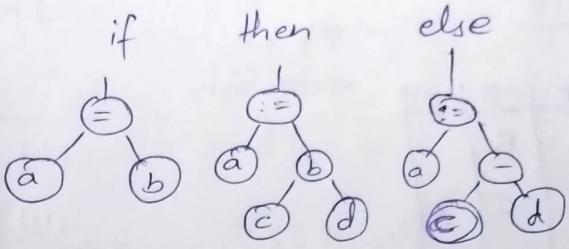
a* (b+c)/d



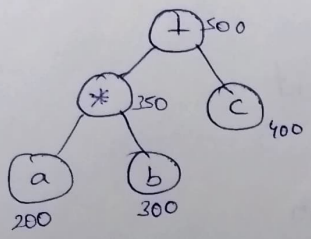
if c then x else y



if (a = b) then a := c + d else a := c - d



a * b + c



Three address code

A + B * C

$T_1 := B * C$

$T_2 := A + T_1$

A := - B * (C - D)

$T_1 := -B$

$T_2 := C - D$

$T_3 := T_1 * T_2$

$A := T_3$

Quadruple

	op	arg1	arg2	Result
(0)	-	B		T_1
(1)	-	C	D	T_2
(2)	*	T_1	T_2	T_3
(3)	:=	T_3		A

15-02-2022

Triple

	op	arg1	arg2
(0)	-	B	
(1)	+	C	D
(2)	*	(0)	(1)
(3)	:=	A	(2)

Indirect triple

triple

	op	arg1	arg2
(0)	-	B	
(1)	+	C	D
(2)	*	(0)	(1)
(3)	:=	A	(2)

Indirect triple

	Op1
(14)	(0)
(15)	(1)
(16)	(2)
(17)	(3)

Syntax Directed translation for assignment

Production	Semantic Action

$A \rightarrow id := E$	{ p = lookup(id.place)
	if (p != null)
	gen(p := E.place)
	else error
	}

$E \rightarrow E^1 + E^2$	{ T = newtemp() gen(T.place = E^1.place + E^2.place) }
$E \rightarrow E^1 * E^2$	{ T = newtemp() gen(T.place = E^1.place * E^2.place) }
$E \rightarrow -E^1$	{ T = new temp() gen(T.place := -E^1.place) }
$E \rightarrow (E^1)$	{ E.place = E^1.place }

$E \rightarrow id$

```
{ p = lookup (id.place)
  if (p != null)
     E.place = id
  else error
}
```

Eg: $x = -B * (C + D)$

$T_1 = -B$

$T_2 = C + D$

$T_3 = T_1 * T_2$

$x = T_3$

Input	state	place	Code Generated
$X := -B * (C+D)$	-	-	
$:= -B * (C+D)$	id	X	
$-B * (C+D)$	id :=	X-	
$B * (C+D)$	id := -	X- -	
$* (C+D)$	id := -id	X- - B	$T_1 = -B$
$* (C+D)$	id := - E	X := T_1 -	
$* (C+D)$	id = E *	X = T_1 - *	
$(C+D)$	id := E * (id	X - T_1 - - (	
$C+D)$	id := E * (id	X - T_1 - - C	
$+D)$	id := E * (E +	X - T_1 - - C -	
$D)$	id := E * (E + id	X - T_1 - - C - D	
$)$	id := E * (E + E	X - T_1 - - C - D	$T_2 = C - D$
$)$	id := E * (E	X - T_1 - - T_2	
$)$	id := E * (E)		

id := E * (E)	x - T_1 -- T_2 -	
id := E * E	x - T_1 - T_2	$T_3 = T_1 * T_2$
id := E	x - T_3	x = T_3

15-02-2022

Assignment statement for mixed datatypes

X = Y + I * J

Let X, Y - real

I, J - integer

T_1 := I *int J

T_2 := int to real T_1

T_3 := Y_{real} + T_2

X := T_3

E.place
= Mode - real/integer

$E \rightarrow E^1$ op E^2

X op Y

① X, Y - integer
② X, Y - real
③ X - integer
④ Y - ~~real~~ integer

T = newTemp()

If E^1.Mode = integer and E^2.Mode = integer then
begin
Gen(T = E^1.place int op E^2.place)
E.mode = INTEGER
end
else if E^1.Mode = real and E^2.mode = real then
begin
Gen(T = E^1.place ~~real~~ real op E^2.place)
E.Mode = REAL
end
else if E^1.Mode = integer

```
begin
    U = newtemp()
    Gen(U = int to real E¹.place)
    Gen(T = U op_real E².place)
    E.Mode = REAL
end.

else
  begin
    U = newtemp()
    Gen(U = int to real E².place)
    Gen(T = E¹.place op_real U)
    E.Mode = REAL
end
E.place = T
```

Boolean statement

Logical - AND, OR, NOT Relational - <, >, ≤, ≥, =, ≠

True - 1 False - 0

A AND B OR C

T_1 = A AND B

T_2 = T_1 OR C

x < y - True

T = 1

else

T = 0

if (x < y) then 1 else 0

(1) if x < y goto (4)

(2) T = 0

(3) goto (5)

(4) T = 1

(5) ----

x<y and a=b or c>d

if (a>b) and (a>c)

(1) if a>b goto (4)

(2) T = 0

(3) goto (8)

(4) if (a>c) goto (7)

(5) T=0

(6) goto (8)

(7) T=1

(8) end

21-02-2022

Boolean Expression

not A or B

T_1 := not A

T_2 := T_1 or B

A<B

if A<B then 1 else 0

(1) if A<B goto (4)

(2) T_1 := 0

(3) goto (5)

(4) T_1 := 1

Production

E -> E and E | E or E | not E | (E)

id | id.relop id

Semantic Actions

E -> E' and E''

{ E.place = newtemp()

Gen(E.place = E.place and E''.place)

}

E -> E' or E''

{ E.place = newtemp()

Gen(E.place := E.place or E''.place)

}

$E \rightarrow not\ E'$

{E.place := newtemp()
Gen (E.place = not E'.place)}

$E \rightarrow (E)$

{E.place = E'.place}

$E \Rightarrow id\ relop\ id$

{E.place := newtemp()
Gen(if (id.place relop id'.place)
Gen (Goto next Quad+3)
Gen (E.place := 0)
Gen (Goto next Quad+2)
} Gen (E.place := 1)

A < B or C

(1) if A<B goto 4)
(2) $T_1 := 0$
(3) goto (5)
(4) $T_1 := 1$
(5) $T_2 := T_1$ or C

if (A > B and A > C) then X := Y + Z

(1) if (A > B) goto 3
(2) goto (7)
(3) if (A > C) goto 5
(4) goto 7
(5) $T_1 := Y + Z$
(6) $X := T_1$
(7) —

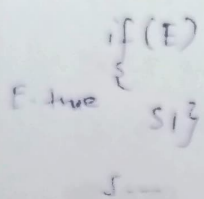
while (A > B or A > C)
{
X := A + B
Y := A - B
}

(1) if A > B goto 4
(2) if A > C goto 4
(3) goto 9
(4) $T_1 := A + B$
(5) $X := T_1$
(6) $T_2 := A - B$
(7) $Y := T_2$
(8) goto 1
(9) ....

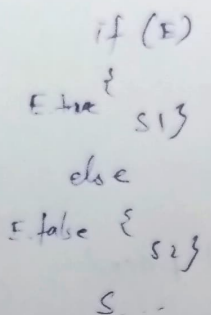
22-02-2022

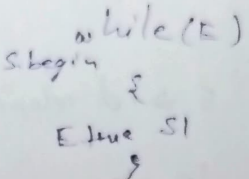
SDT for flow of control statement

if (E) then S1

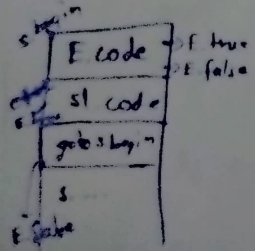


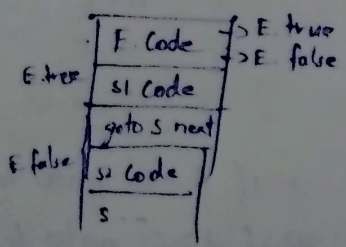
if E then S1 else S2





Production	Semantic Action
S -> if E then S1	{ E.true := newlabel E.false := S.next S1.next := S.next S.code := E.code || gen(E.true ':') || S1.code }
S -> if E then S1 else S2	{ E.true := newlabel E.false := newlabel S1.next := S.next S2.next = S.next S.code := E.code || gen(E.true ':' || S1.code || gen(goto S.next) || gen(E.false ':') || S2.code) }
S -> while E then S1	{ S.begin := newlabel E.true := newlabel S1.code := S.begin E.false := S.next S.code := gen(S.begin ':') || E.code || gen(E.true ':') || S1.code || gen(goto S.begin) }





SDT for Procedure call

$S \rightarrow call\ id(E.list)$	{ for each item P in the queue gen(param P) gen(call, id.place, length of queue) }
$E\ list \rightarrow E.list\ ,\ E$	{ append E.place at the end of queue }
$E\ list \rightarrow E$	{ initialize a queue to contain only E.place }

23-02-2022

SDT for switch-case statement

```
switch(E)
{
  case V1: S1
           break;
  case V2: S2
           break;
    :
  case Vn: Sn
           break;
  default: Sn
}
```

SDT

evaluate E into T
goto test
L_1 : Code for S_1
goto next
L_2 : code for S_2
goto next
:
L_{n-1} : Code for S_{n-1}
goto next
L_n : Code for S_n
goto next
test : if $T = V_1$ goto L_1
if $T = V_2$ goto L_2
if $T = V_3$ goto L_3
:
if $T = V_{n-1}$ goto L_{n-1}
goto L_n
next:

Eg:
```
switch (x*y)
{
    case 1: c=a+b;
            break;
    case 2: c=a-b
            break;
    default: c=0
```

SDT

```
T := x * y
goto test
L1: T1 := a+b
    c := T1
    goto next
L2: T1 := a-b
    c := T1
    goto next
default:
L3: c = 0
    goto next
test: if T==1 goto L1
      if T==2 goto L2
      goto L3
next:
```

Symbol table

Operations
allocate()
free
lookup()

```
int x;
int proc1()
{
  int one;
  int two;
  {
    int three;
    int four;
  }
  int five;
  {
    int six;
    int seven;
  }
}
```

28-02-2022

Symbol table

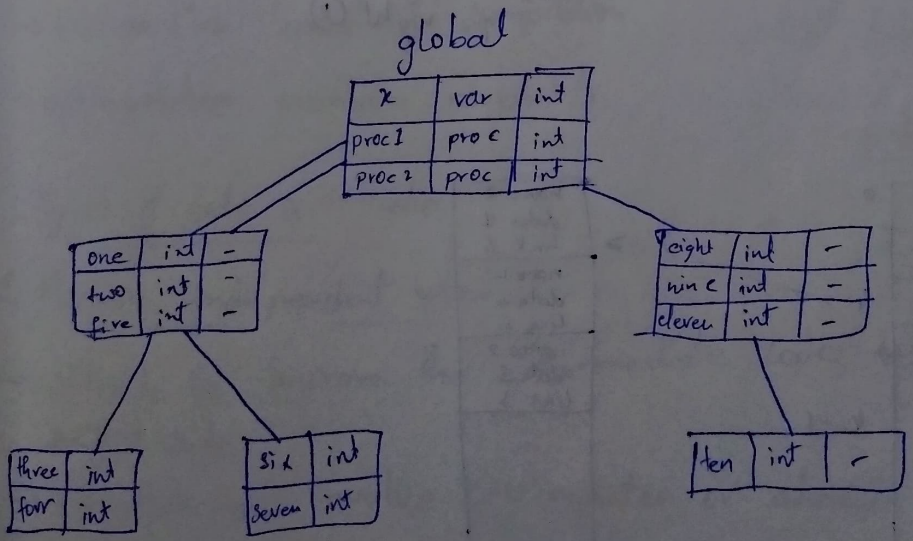
```
int x;
int proc1()
{
    int one;
    int two;
    {
        int three;
        int four;
    }
    int five;
    {
        int six;
        int seven;
    }
}
int proc 2()
{
    int eight;
    int nine;
    {
        int ten;
    }
    int eleven;
}
```

inner scope 1

inner scope 2

inner scope 3

Scope management



28-02-2022

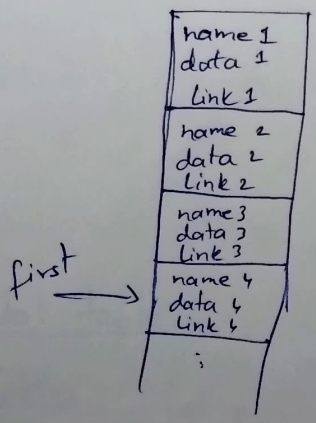
Data structures for implementation of Symbol table

There are three Data structure in this.

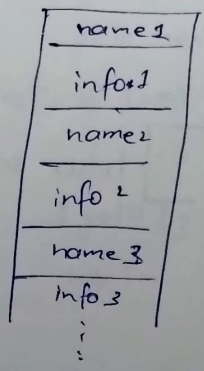
Linear List - self organizing list.

search tree

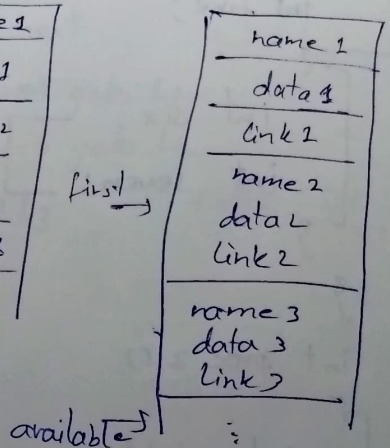
Hash table



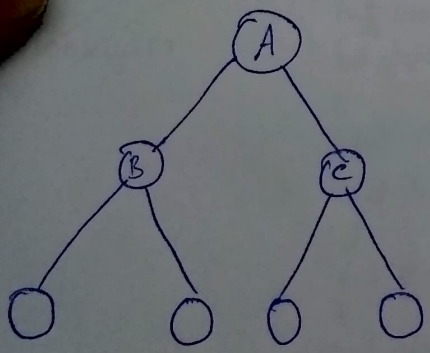
Linked List



self organizing list

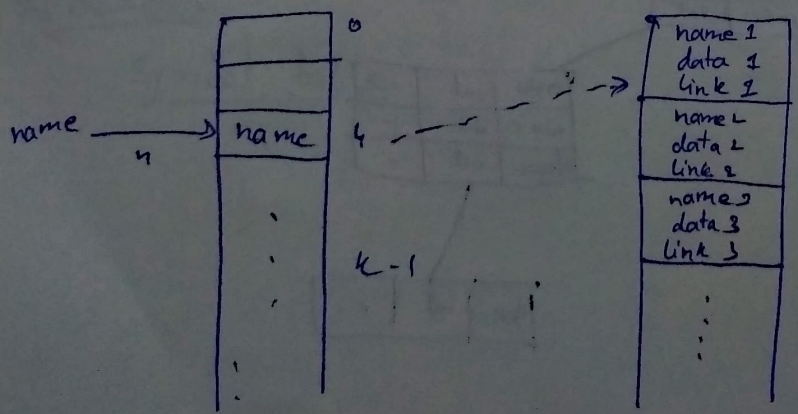


Search tree




```
if (Name = Name (P))
    then name forward
else if (name < name (P))
    then P := left (P)
else
    P := right (P)
```

Hash table



01-03-2022

Unit - IV

Code Optimization

Code optimization

It is a program transformation technique, while tries to improve the intermediate code by making it consume to more resource, so that faster machine code will result.

Rules for selection of Optimizing transformation:

-> The output code must ~~be~~ not in any way to change the meaning of the program.

-> Optimization should increase the speed of the program and if possible the ~~program~~ progress should use less resource.

-> optimization should not delay the overall compiling process

Need for optimization

-> To reduce the space consume

-> Increase the speed of compilation

-> optimization promotes reusability

Types of Code optimization

1. Machine independent optimization

-> Attends to improve the intermediate code to get better target code.

-> It does not involve CPU register or absolute memory location.

2. Machine Dependent Optimization :

-> Done after target code has been generated. It involves CPU register and absolute memory location.

Principle source of optimization:

-> Global optimization

-> local optimization

1. Common subexpression elimination
2. Copy propogation of variable propogation
3. Dead Code elimination
4. Constant folding

1. Common subexpression elimination

```
a = b + c
b = a - d
c = a + b
d = a - d
```

⇒

```
a = b + c
b = a - d
c = b + c
a = b
```

```
A[I+1] = B[I+1]
```

⇓

```
J = I+1
A[J] = B[J]
```

2. Copy Propogation

```
c = a * b
x = c
d = x + c * b
```

→

```
c = a * b
x = c
d = c + e * b
```

3 Dead code elimination

```
c = a * b
d = c + e * b
```

```
x = 0
if (x == 1)
{
printf("%d", x);
}
```

4. Constant folding

```
x = (22/7) * r
x = 3.14 * r
```

```
z = 2 * 3 * 6
  = 6 * 6
```

Loop optimization

1. Code motion
2. Reduction in strength
3. Induction variable elimination

1. Code motion (loop invarient) every time it executes x will be constant. x is independent of the loop.

```
for (i=1; i<=10; i++)
{
  x=10;
  a=x+i;
}
```

→

```
x = 10;
for (i=1; i<=10; i++)
{
  a=x+i;
}
```

2. Reduction in strength using different ~~operatio~~ operators

```
x = 2 * I   → x = I + I
```

```
i=1
while (i<=5)
{
  T = 4*i
  A[I] = a[i]
  i = i+1
}
```

⇒

```
i=1
T:=0
while (i<=5)
{
  T = T+4
  A[I] = a[i]
  i = i+1
}
```

3. Induction variable elimination

variables which will either increase or decrease when you loop with a constant value when you loop

```
i=1
T=0
while (i<=5)
{
  T = T+4
  Sum = sum + A[T]
  T = i+1
}
```

c)
```
T=0
while (i<=20)
{
  T = T+4
  sum = sum + A[T]
}
```

02-03-2022

Basic Blocks

```
Prod := 0
I := 1
do
  begin
    prod := prod + A[I] + B[I]
    I := I + 1
  end
  while(I≤20)
```

Pre-address code for the above program

(1) prod := 0

(2) I := 1

(3) T_1 := 4 * I

(4) T_2 := addr(A) - 4

(5) T_3 := T_2 [T_1]

(6) T_4 := addr(B) - 4

(7) T_5 := T_4 [T_1]

(8) T_6 := T_3 * T_5

(9) prod := prod + T_6

(10) I := I + 1

(11) if I ≤ 20 goto (3)

Basic block is a sequencing consecutive statements which maybe entered only at the beginning and when entered are executed in the sequence without call of possibility of a branch.

partitioning into basic blocks

Algorithm

Input: List of 3 address statement

Output: List of basic blocks with each 3 address ~~cod~~ statement at exactly one block.

Method

(1) Determine the set of leader that is the first statement of each block

(i) The first statement is the leader

(ii) Any statement which is the target of the conditional or unconditional goto is a leader.

(iii) Any statement which immediate follows a conditional goto is leader.

(2) for each leader constructs it's blocks. which consists of leader and all the statement upto but not including the next leader or the end of a program.

B1

```
prod := 0
I := 1
```

B2

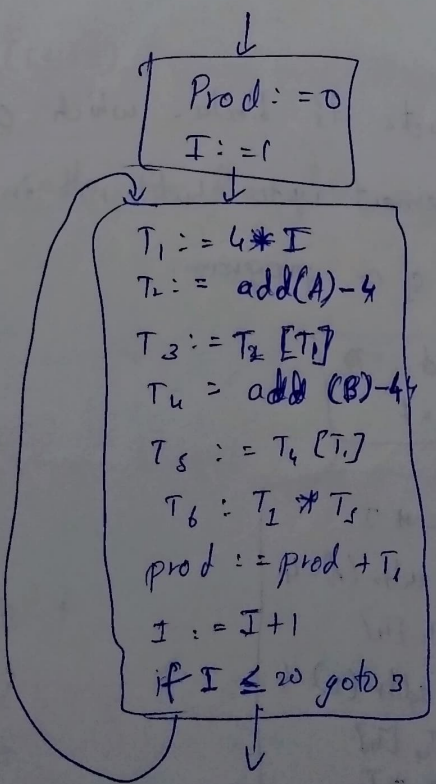
```
T1 := 4 * I
T2 := addr(A) - 4
T3 := T2 [T1]
T4 := addr(B) - 4
T5 := T4 [T1]
T6 := T3 * T5
I = 2 + 1
```

Flow graph

It is a directed graph with flow control information added to the basic blocks.

→ The basic blocks serve as nodes of the program. 1 node is distinguished as initial, it is the block whose leader is in the first statement.

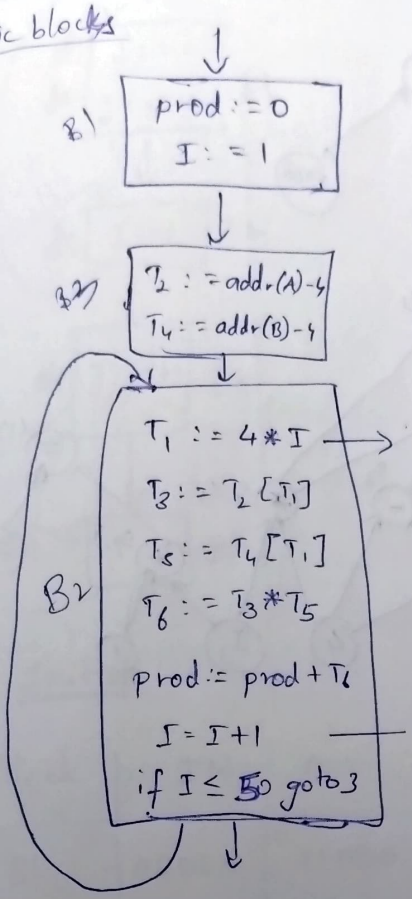
→ There is an directed edge from block B_1, ~~to~~ to Block$_2$ and if B_2 could immediately follow B_1, here is execution. That is there is a conditional or unconditional jump from the last statement of B_1 to the first statement of B_2 (or) B_2 immediately follows B_1 in the order of the program and B_1 does not end in a unconditional jump.

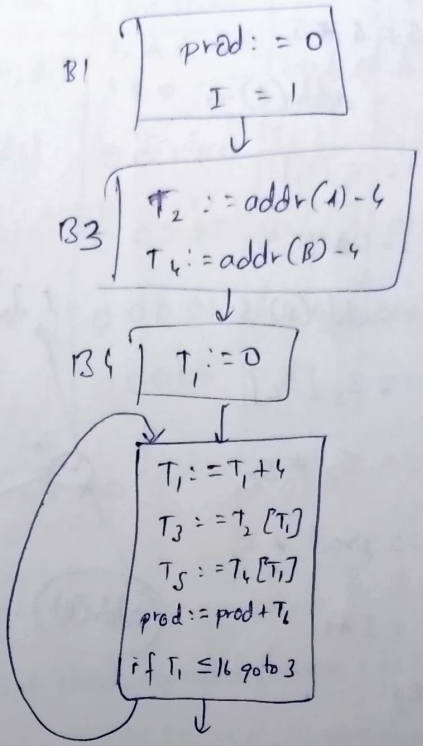



```
Prod := 0
I := 1
do
  begin
    prod := prod + A[I] + B[I]
    I := I + 1
  end
    while (I ≤ 20)
```

08-03-2022

Basic blocks

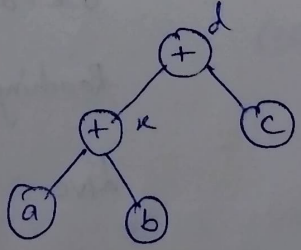




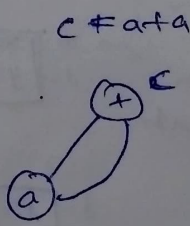
09-03-2022

Directed Acyclic Graph (DAG)

x := (a+b)
d := x + c



c = a+a



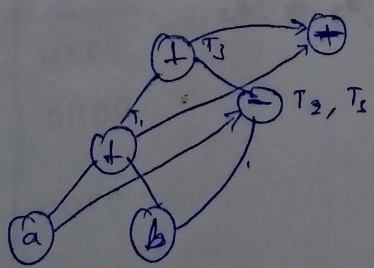
$T_1 := a+b$

$T_2 := a-b$

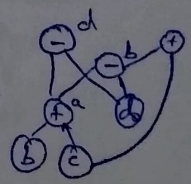
$T_3 := T_1 + T_2$

$T_4 := T_3 - T_1$

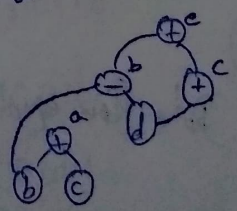
$T_5 = a-b$



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



a = b+c
b = b-d
c = c+d
e = b+c



(1) $S_1 := 4 * I$

(2) $S_2 := addr(A) - 4$

(3) $S_3 := S_2[S_1]$

(4) $S_4 := 4 * I$

(5) $S_5 := addr(B) - 4$

(6) $S_6 := S_5[S_4]$

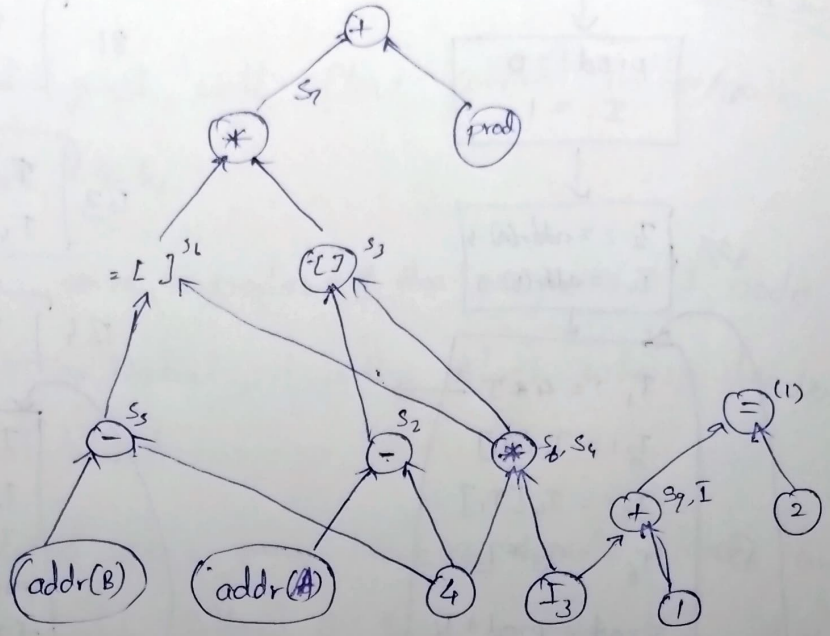
(7) $S_7 := S_3 * S_6$

(8) $S_8 := prod * S_7$

(9) $S_9 := I + 1$

(10) $I := S_9$

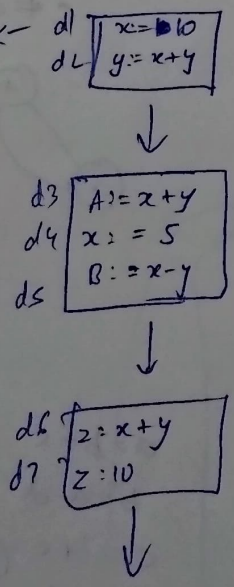
(11) If $I \leq 20$ goto (1)



14-03-2022

Global Data flow analysis

~~definition~~ definations



d_4, d_2, d_3, d_4, d_5

Use-definitions [x:=5, check before whether it is defined already]

Reaching definitions [what are the definations reaching that point] d_4, d_2, d_3, d_4, d_5

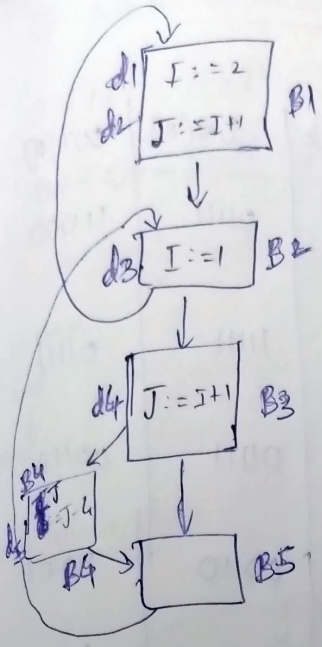
Live

(iv)

Data flow equation: $IN[B] = \bigcup_{P} OUT[P]$

P is a predecessor of B

$OUT[B] = (IN[B] - KILL[B]) \cup GEN[B]$



Block	Gen	Bit vector	Kill	Bit vector
B1	{d1, d2}	$d_1\ d_2\ d_3\ d_4\ d_5$ 1 1 0 0 0	{d3, d4, d5}	00111
B2	{d3}	0 0 1 0 0	{d1}	10000
B3	{d4}	0 0 0 1 0	{d1, d5}	01001
B4	{d5}	0 0 0 0 1	{d2, d4}	01010
B5	∅	0 0 0 0 0	∅	00000

OR

0+0 = 0	0-0 = 0
0+1 = 1	0-1 = 0
1+0 = 1	1-0 = 1
1+1 = 1	1-1 = 0

Initial

Block	IN	OUT
B1	00000	11000
B2	00000	00100
B3	00000	00010
B4	00000	00001
B5	00000	00000

Kill -> if check repeated (where i is used, j is used)

(00000 - 00111) + 11000

(00000 - 10000) + 00100

(00000 - 01001) + 00010

(0000 - 01010) + 00001

01000

15-03-2022

Pass I

Block	IN [B]	OUT [B]
B1	00100	11000
B2	11000	01100
B3	01100	00110
B4	00110	00101
B5	00111	00111

Pass II

Block	IN [B]	OUT [B]
B1	01100	11000
B2	11111	01111
B3	01111	00110
B4	00110	00101
B5	00111	00111

Pass III

Block	IN[B]	OUT[B]
B1	01111	11000
B2	11111	01111
B3	01111	00110
B4	00110	00101
B5	00111	00111

Pass III

Block	IN[B]	OUT[B]
B1	01111	11000
B2	11111	01111
B3	01111	00110
B4	00110	00101
B5	00111	00111

16-03-2022

Reaching definition

```
begin
  For each block B do
  begin
     IN[B] = ∅
  OUT[B] = GEN[B]
  end; /* initialize on the assumption IN[B] = ∅
                                  for all B */
CHANGE = true; /* get the while-loop going */
while CHANGE do
 begin
   CHANGE := False;
   for each block B do
     begin
        NEWIN := ∪ OUT[P];
                 P as Predecessor
                 of B
/* hold IN[B], computed by rule (2) in temporary to check for a change */
```

if NEWIN ≠ IN[B] then CHANGE := true;

IN[B] = NEWIN;

OUT[B] := IN[B] - KILL[B] ∪ GEN[B]

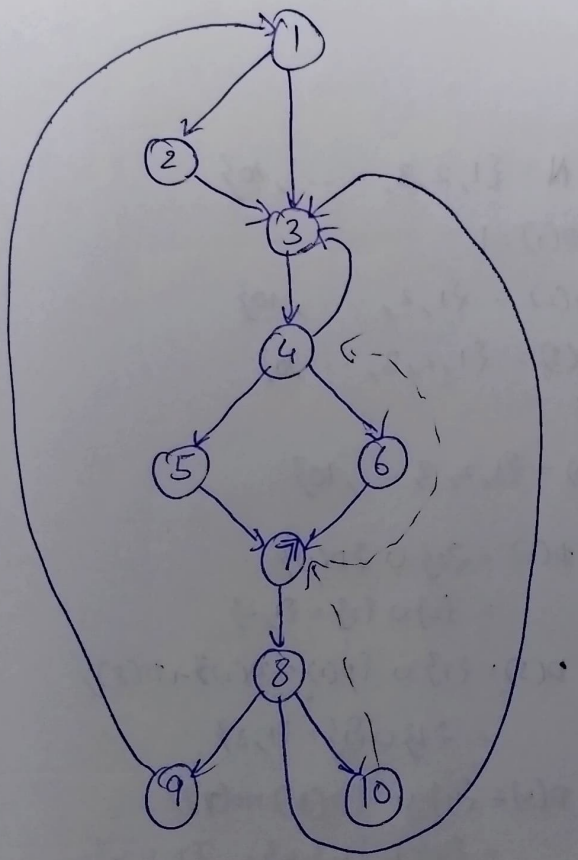
/* apply rule (1)

end

end

end

Deminators



used to find loops & flow graph

1. has one entry point

Reaching to 4

d dom n

1 dom 3

3 dom 4

1 dom 4

1. 1 is always a dominator

Algorithms for finding Dominators

begin

$D(n_0) = \{n_0\}$

for n in $N - \{n_0\}$

do

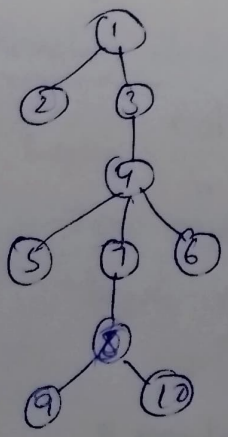
$D(n) = \{N\}$

CHANGE = True

while change do

```
begin
    CHANGE := false
    for n in N - {0} do
    begin
      NEWD := {n} ∪ (⋂ D(p))
                    p ∈ Predecessor of n
      if D(n) ≠ NEWD then CHANGE := True;
      D(n) := NEWD
    end
  end
end
```

Dominator tree



$N = \{1, 2, 3, \ldots, 10\}$

$D(1) = 1$

$D(2) = \{1, 2, \ldots, 10\}$

$D(3) = \{1, 2, 3, \ldots, 10\}$

$\vdots$

$D(10) = \{1, 2, 3, \ldots, 10\}$

$D(2) = \{2\} \cup \{D(1)\}$

$= \{2\} \cup \{1\} = \{1, 2\}$

$D(3) = \{3\} \cup \{D(2) \cap D(1)\} \cap D(8)\}$

$= \{3\} \cup \{1\} = \{1, 3\}$

$D(4) = \{4\} \cup \{D(3) \cap D(7)\}$

$= \{4\} \cup \{1, 3, 4\} = \{1, 3, 4\}$

$D(5) = \{5\} \cup \{D(4)\}$

$= \{5\} \cup \{1, 3, 4\} = \{1, 3, 4, 5\}$

$D(6) = \{6\} \cup \{D(4)\}$

$= \{6\} \cup \{1, 3, 4\} = \{1, 3, 4, 6\}$

Algorithm for finding loops

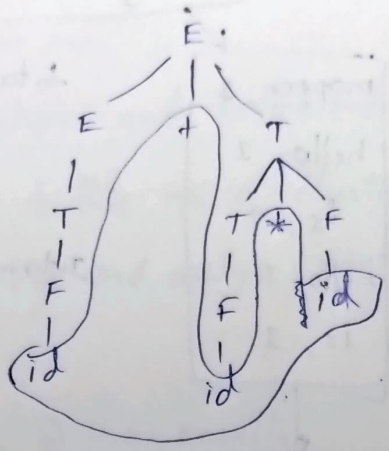
```
procedure INSERT(m)
  if m' is not in LOOP then
    begin
      LOOP = LOOP ∪ {m'}
      push m onto STACK
    end
/* main program */
    STACK := empty
    LOOP := {d}
    INSERT(n)
    while STACK is not empty do
    begin pop 'm', the 1st element off the stack
    end for each predecessor 'p' of 'm' do INSERT(p)
```

21-03-2022

Compiler Design

Unit-II

Top down parser



$E \to E * T / T$

$T \to T * F / F$

$F \to (E) / id$

$id + id * id$

Recursive descent parser

Backtracking, left recursion

→ first nth element left recursion

$A \to A\alpha / \beta$

$\Downarrow$

$A \to \beta A'$

$A' \to \alpha A' / \varepsilon$

$E \to E + T / T$

$E \to T E'$

$E' \to + T E' / \varepsilon$

$T \to T * F / F$

$T \to F T'$

$T' \to * F T' / \varepsilon$

$F \to (E) / id$

$A \to \underset{a}{a\alpha_1} / \underset{a}{a\alpha_2} / \underset{a}{a\alpha_3}$

$\Downarrow$

$A \to \alpha A'$

$A' \to \alpha_1 / \alpha_2 / \alpha_3$

$S \to \underset{a}{i}\,c\,t\,\underset{\alpha_1}{S}\,e\,s / \underset{a}{i}\,c\,t\,\underset{\alpha_2}{S} / a$

$C \to b$

$S \to i\,c\,t\,S\,S' / a$

$S' \to e\,s / \varepsilon$

$C \to b$

First and Follow

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

(Eg:) $S \rightarrow ABCD$

$A \rightarrow a/\varepsilon$

$B \rightarrow CD/b$

$C \rightarrow e/\varepsilon$

$D \rightarrow Aa/d/\varepsilon$

S {First(A)} = {a, First(B)} = First{a, e, b, d, e}

a, e, b, d, First(C), First(D).

A {a, ε}

B {First(c), b} = {e, b, First(D))

{e, b, a, d, ε}

C {e, ε}

D {First(A), d, ε}

{a, d, ε}

22-03-2022

Predictive parser/LL(1) parser

$E \to E+T/T$

$T \to T+F/F$

$F \to (E)/id$

$E \to TE'$

$E' \to +TE'/\varepsilon$

$T \to FT'$

$T' \to *FT'/\varepsilon$

$F \to (E)/id$

(alphabets)

Non-terminal → E', T, E, ...

Terminal → +, *, (,), id

predictive parsing table (refer the table behind)

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

Take follow

input : id+id*id$

stack	input	Action
$E	id+id*id$	
$E'T	id+id*id$	$E \to TE'$
$E'T'F	id+id*id$	$T \to FT'$
$E'T'id	id+id*id$	$F \to id$
$E'T'	+id*id$	pop id
$E'	+id*id$	$T' \to \varepsilon$
$E'T+	+id*id$	$E' \to +TE'$
$E'T	id*id$	pop +
$E'T'F	id*id$	$T \to FT'$
$E'T'id	id*id$	$F \to id$
$E'T'	*id$	pop id
$E'T'F*	*id$	$T' \to *FT'$
$E'T'F	id$	pop *
$E'T'id	id$	$F \to id$
$E'T'	$	pop id
$E'	$	$T' \to \varepsilon$
$	$	$E' \to \varepsilon$

$ E' T F	id * id $	T → F T'
$ E' T id	id * id $	F → id
$ E' T	* id $	pop(id)
$ E' T' F *	* id $	T → F T'
$ E' T' F	id $	pop *
$ E' T' id	id $	F → id
$ E' T'	$	pop(id)
$ E'	$	T' → ε
$	$	E' → ε
$	$	Accept

iput : id + *

stack	Input	Action
$ E	id+*$	E → T E'
$ E T	id+*$	E → T E'
$ E' T' F	id+*$	T → F T'
$ E' T' id	id+*$	pop id
$ E' T'	+ * $	T' → ε
$ E'	+ * $	T' → ε
$ E' T +	+ * $	E' → + T E'
$ E' T	* $	

② $S \to iCtSeS / iCtS / a$

$C \to b$

$A \to A\alpha / \beta$

$A \to \beta A'$

$A' \to \alpha A / \varepsilon$

left factoring

$A \to a\alpha_1 / a\alpha_2 / \ldots$

$A \to aA'$

$A' \to \alpha_1 / \alpha_2 / \ldots$

$S \to iCtSS' / a$

$S' \to eS / \varepsilon$

$C \to b$

	First	Follow
S	{i, a}	{$, First(S')} ⇒ {$, e}
S'	{e, ε}	{$, e}
C	{b}	{t}

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

③ $S \to (L) / a$

$L \to L, S / S$

input (a, a)

23-03-2022

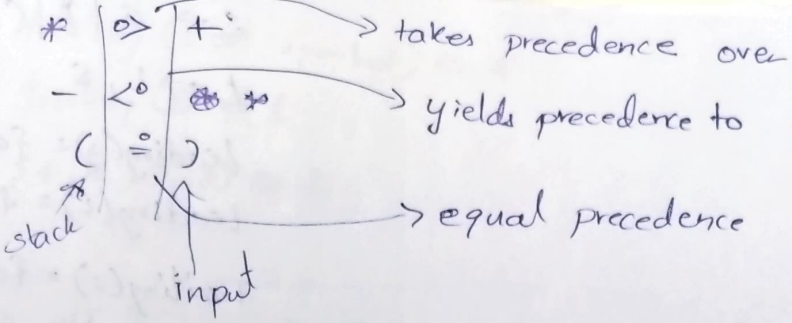
Operator precedence Parser

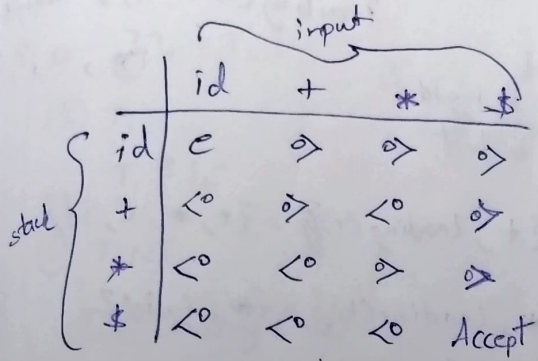
Operator grammar

1. No production ~~on the right side~~ should have ε in RHS

$A \to \varepsilon$ ✗

2. RHS should not contain 2 adjacent non terminal. $A \to \alpha$ ✗

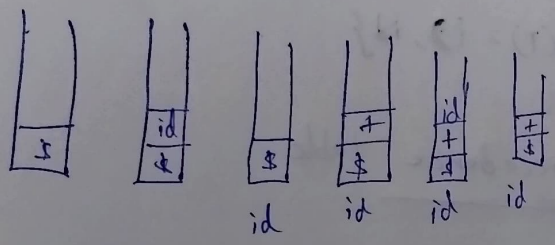


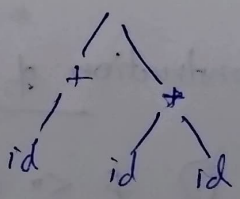


e - error

input: id + id * id $

< or = → push
> → pop





input

	id	+	*	(	)	$
id	e	>	>	e	>	>
+	<	>	<	<	>	>
*	<	<	>	<	>	>
(	<	<	<	<	=	e
)	e	>	>	e	>	>
$	<	<	<	<	e	Accept

① $E \to E+T/T$

$T \to T*F/F$

$F \to (E)/id$

$S \to a \mid \uparrow \mid (L)$

$L \to L, S$

Leading(S) = {a, ↑, (}

Leading(L) = {, , }

Trailing(S) = {a, ↑,)}

Trailing(L) = {, , Trailing(S)}

{, a, ↑,)}

Leading and Trailing

Leading → left to right

trailing → Right to left

$E \to E+T$

$E \to T$

$T \to T*F$

$T \to F$

$F \to (E)$

$F \to id$

Leading(E) = {+, leading(T)} = {+, *, (, id}

Leading(T) = {*, Leading(F)} = {*, (, id}

Leading(F) = {(, id}

Trailing(E) = {+, trailing(T)} = {+, *,), id}

Trailing(T) = {*, trailing(F)} = {*,), id}

Trailing(F) = {), id}

Construction of operator precedence table

① $ <· Leading(start symbol)

$ <· Leading(E)

$ <· {+, *, (, id}

input

stack	+	*	(	)	id	$
+	·>	<·	<·	·>	<·	·>
*	·>	·>	<·	·>	<·	·>
(	<·	<·	<·	=·	<·	e
)	·>	e	e	·>	e	·>
id	·>	e	e	·>	e	·>
$	<·	<·	<·	e	<·	Accept

Trailing (start symbol) => $

Trailing (E) => $

{+, *,), id) => $

② $T_1 . NT . T_2$ T -> Single Terminal

$T_1 = T_2$

③ $NT . T$

Trailing (NT) => T

③ $T_1 T_2$

$T_1 \doteq T_2$

④ T. NT

T <. Leading (NT)

+ <. Leading (T)

+ <. {*, (, id}

* <. Leading (T)

* <. {(, id}

(<. Leading (E)

(<. {+, *, (, id}

Trailing (E) => +

{+, *, (, id} => +

Trailing (T) => *

Trailing (E) =>)