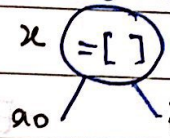
8-02-'19

→ Optimization Techniques for Basic Blocks (Contd).

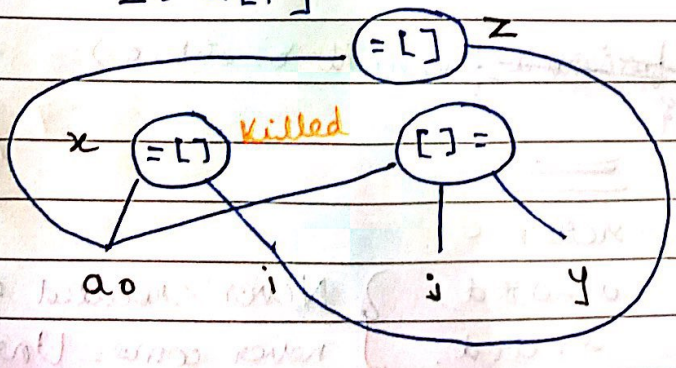
– Representation of Array Elements

(i) An assignment from an array with x = a[i] is represented by creating a node with operator '=[]' a two children representing the initial value of the array. a_0 and index i. Variable x becomes a label of this new node. x = a[i]



(ii) Assignment to an array like a[j] = y is represented by a new node with operator '[] =' and three children representing a_0, j and y. There is no variable labelling this node. The creation of this node kills all currently constructed nodes whose value depends on a_0.

→ Eg. x = a[i]
a[j] = y
z = a[i]



To not lose information

Eg. b = x * c
a = b[i]

→ Peephole Optimization

- Eliminating Redundant Loads and Stores
- Eliminating Unreachable Code
- Flow Of Optimizations
- Algebraic Simplification and Reduction in Streng
- Uses of Machine Idioms

-Eliminating Redundant Loads and Stores

```
LD R0, x   } Redundant, no use
ST x, R0   }
```

- Eliminating Unreachable Code

```
eg. flag = 0
if (flag)          → Never true ∴ flag = 0
{
  - - - -   } Never reached as condition is
  - - - -   } always false. Unreachable.
}
```

```
eg. function (int a, int b, int c)
{
  ========
  return c;
  a = a * d;   } Never executed as control
  a = a + d;   } never comes. Unreachable.
}
```

- Flow of Optimizations

Eg. if (x>y) goto L1 | if (x>y) goto L3

L1: goto L2 | L1: goto L2

⇒

L2: goto L3 | L2: goto L3

L3: - - - - | L3: - - - - -

- - - - | - - - - -

- - - | - - - - - -

- Algebraic Simplification and Reduction in Strengths

Eg. x = x + 0

x = x - 0 ⇒ x } Algebraic Simplification

x = x * 1

x = x / 1

Use Cheaper Operations in place of Expensive Ones - Reduction in Strengths

- Use of Machine Idioms

Eg. x = x + 1

→ We must use register to store 1, then add it. So use INC instead

Eg. Use XCHG for exchange

Q. - Construct LL Parser

$S \rightarrow L = R / R$

$L \rightarrow *R / id$

$R \rightarrow L$

→ $FIRST(S) = \{*, id\}$

$FIRST(L) = \{*, id\}$

$FIRST(R) = \{*, id\}$

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

$FOLLOW(L) = \{=, \$\}$ [$= FIRST(=R) \cup FOLLOW(R)$
$= (=) \cup (R) = \{\$, =\}$

$FOLLOW(R) = \{=, \$\}$ [$= FOLLOW(L) \cup FOLLOW(S)$
$= (=) \cup \{\$\}) = \{=, \$\}$

	*	=	id	$
S		S → L=R S → R	S → L=R S → R	
L		L → *R	L → id	
R		R → L	R → L	

$FIRST(S) = FIRST(L=R) \cap FIRST(R) = \{*, id\} \cap \{*, id\}$

$\neq \phi$

∴ The given grammar is not LL1.

Q. - Construct LL(1) Parser

$S \rightarrow i\,E\,t\,S \mid i\,E\,t\,S\,e\,S$

$E \rightarrow T\,r\,T \mid T$

$T \rightarrow id \mid num$

→ FIRST(S) = {i} FOLLOW(S) = {e, $}

FIRST(E) = {id / num} FOLLOW(E) = {t}

FIRST(T) = {id / num} FOLLOW(T) = {r, t}

	i	t	e	r	id	num	$
S							
E							
T							

FIRST(S) = FIRST(iEtS) ∩ FIRST(iEtSeS) = {i} ∩ {i} ≠ ∅

05-03-'19

① $S \to L=R$ ② $S \to R$ ③ $L \to *R$ ④ $L \to id$ ⑤ $R \to L$

Q. Eg. Construct SLR for:

① $S \to L=R/R$ ② $L \to *R/id$ ③ $R \to L$

→ Augment the grammar

$S' \to \cdot S, \$$

$S \to \cdot L=R, \$$ } FIRST(βa) = FIRST($G\$$) = $\$$

$S \to \cdot R, \$$

$L \to \cdot *R, =$ } FIRST(βa) = FIRST($=\$$) = $=$

$L \to \cdot id, =$

$R \to \cdot L, \$$ } FIRST(βa) = FIRST($G\$$) = $\$$

$L \to \cdot *R, \$$ } FIRST(βa) = FIRST($G\$$) = $\$$

$L \to \cdot id, \$$

I0:
$S' \to \cdot S, \$$
$S \to \cdot L=R, \$$
$S \to \cdot R, \$$
$L \to \cdot *R, =$
$L \to \cdot id, =$
$R \to \cdot L, \$$
$L \to \cdot *R, \$$
$L \to \cdot id, \$$

I1:
GOTO (I0, S) →
$S' \to S \cdot, \$$

~~I2~~ I2:
GOTO (I0, L) =
$S \to L \cdot = R, \$$
$R \to L \cdot, \$$

I3:
GOTO (I0, R)
~~$S \to L \cdot = R, \$$~~
$S \to R \cdot, \$$

I3:
GOTO (I0, *)
$S \to *\cdot R, =$
$R \to \cdot L, =$

I4:

GOTO(I0, *)

S → *·R, =

R → ·L, =

L → ·*R, =

L → ·id, =

L → *·R, $

R → ·L, $

L → ·*R, $

L → ·id, $

I5:

GOTO(I0, id)

L → id·, =

L → id·, $

I6:

GOTO(I2, =)

S → L = ·R, $

R → ·L, $

L → ·*R, $

L → ·id, $

I7:

GOTO(I4, R)

L → *R·, =

L → *R·, $

I8:

GOTO(I4, L)

R → L·, =

R → L·, $

I9:

GOTO(I4, *)

L → *R, $

≈ I4

~~I10:~~

GOTO(I4, id)

L → id·, =

≈ I5

I9:

GOTO(I6, R)

S → L = R·, $

I10:

GOTO(I6, L)

R → L·, $

I11:

GOTO(I6, *)

L → *·R, $

R → ·L, $

L → ·*R, $

L → ·id, $

I12:

GOTO(I6, id)

L → id·, $

I13:

GOTO (I11, R)

L → *R•, $

I14:

GOTO (I11, L)

R → L•, $

~~I15~~

GOTO (I11, *)

L → •*R, $

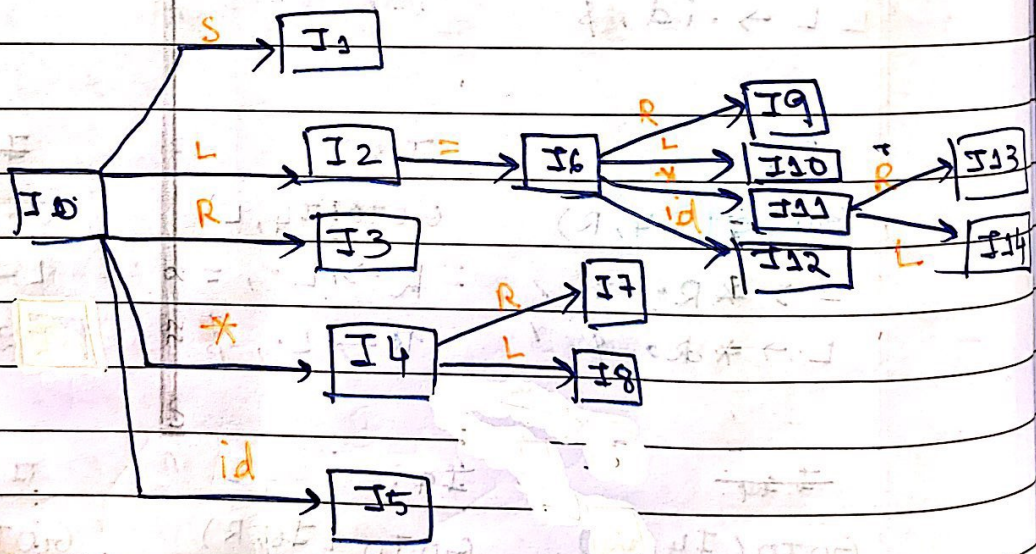
≈ I11

~~I15~~

GOTO (I11, id)

L → id•, $

≈ I11



P.T.O.

Table:

	TERMINALS				VARIABLES		
	*	=	id	$	S	L	R
0	S4		S5		1	2	3
1				ACCEPT			
2		S6		R5			
3				R2			
4	S4		S5			8	7
5		R4		R4			
6	S11		S12			10	9
7		R3		R3			
8		R5		R5			
9				R1			
10				R5			
11	S11		S12			14	13
12				R4			
13				R3			
14				R5			

P.T.O.

→

06-03-'19

Q. Construct the collection of LR(0) itemsets and draw the GOTO graph for: $S \to SS/a/\epsilon$

→ ① $S \to SS$ ② $S \to a$ ③ $S \to \epsilon$

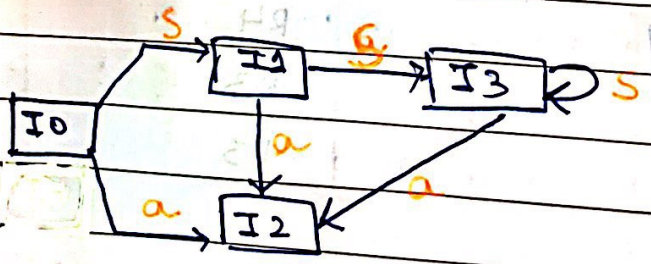
Here, $S \to \epsilon \Rightarrow S \to \cdot$

I_0:	I_1:	I_2:	I_3:
$S' \to \cdot S$	GOTO(I_0, S)	GOTO(I_0, a)	GOTO(I_1, S)
$S \to \cdot SS$	$S' \to S \cdot$	$S \to a \cdot$	$S \to SS \cdot$
$S \to \cdot a$	$S \to S \cdot S$		$S \to S \cdot S$
$S \to \cdot$	$S \to \cdot SS$		$S \to \cdot SS$
	$S \to \cdot a$		$S \to \cdot a$
	$S \to \cdot$		$S \to \cdot$

~~I_4:~~ GOTO(I_1, a) $S \to a \cdot \approx I_2$

~~I_5:~~ GOTO(I_3, S) $S \to SS \cdot \approx I_3$

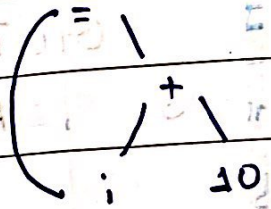
~~I_6~~ GOTO(I_3, a) $S \to a \cdot \approx I_2$



	ACTION		GOTO
	a	$	S
0	S2		1
1	S2		3
2			
3	S2		3

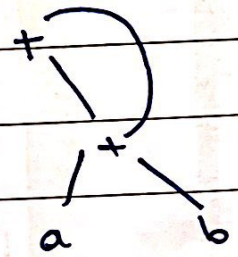
Q. Construct the DAG and identify the value numbers for the sub-expressions of the following expressions assuming '+' associates from the left.

→ (i) i = i + 10



1	i			Value
2	10			number
3	+	1	2	
4	=	3	1	

(ii) a + b + (a + b)



1	a	integer	
2	b	integer	
3	+	1	2
4	+	3	3

if (a > b && a < c || a > d)

Q. Show that the following grammar is LL(1) but not SLR(1)

S → AaAb / BbBa A → ∈ B → ∈

→ FIRST(S) = FIRST(AaAb) ∩ FIRST(BbBa)

= {a} ∩ {b} = ∅

=> LL1 grammar

I0: Augment the grammar: S' → · S

I0:
S' → · S
S → · AaAb
S → · BbBa
A → ·
B → ·

I1: GOTO(I0, S)
S' → S·
S → · AaAb
S → · BbBa
A → ·
B → ·

I2: GOTO(I0, A)
S → A · aAb

I3: GOTO(I0, B)
S → B · bBa

I4: GOTO(I2, a)
S → Aa · Ab
A → ·

I5: GOTO(I3, b)
S → Bb · Ba
B → ·

I_6:	I_7:	I_8:	I_9:
GOTO(I_4, A)	GOTO(I_5, b)	GOTO(I_6, b)	GOTO(I_7, a)
$S \rightarrow AaA \cdot b$	$S \rightarrow BbB \cdot a$	$S \rightarrow AaAb \cdot$	$S \rightarrow BbBa \cdot$

	ACTION			GOTO		
	a	b	$	S	A	B
0						
1						
2						
3						
4						
5						
6						
7						
8						
9						

Q. Write the three address code for: $x = a + b + c[i]$

→ $x = a + b + c[i]$

$t1 = a + b$

$t2 = i * 4$

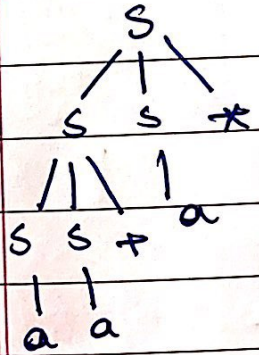
$t3 = c + t2$

$t4 = t3$

$x = t1 + t3$

Q. Construct LL(1) table and generate the strings, i.e., what is the language accepted by: $S \to SS+ / SS* / a$

→



$\Rightarrow aa+a*$

Language Accepted:
Postfix Expression

$S \to SS+ \quad S \to SS* \quad S \to a$

$S \to SS+ / SS* / a$

$S \to S\alpha_1 / S\alpha_2 / \beta \quad [\alpha_1 = S+, \alpha_2 = S*]$

⇒ By eliminating left recursion

$S \to aS'$

$S' \to S+S' / S*S' / \epsilon$

⇒ By eliminating left factoring

$S \to aS'$

$S' \to S+S' / S*S' / \epsilon \quad \Rightarrow \quad A \to \alpha A' \quad A' \to \beta_1 / \beta_2$

$\alpha \ \beta_1 \quad \alpha \ \beta_2$

$\Rightarrow S \to aS'$

$A' \to SA'' / \epsilon$

$A'' \to +S' / *S'$

$S \rightarrow aS'$ $S' \rightarrow SA''/\epsilon$ $A'' \rightarrow +S'/*S'$

Augment the grammar: $S'' \rightarrow \cdot S$

I0:

$S'' \rightarrow \cdot S$

07-03-19

Q. Convert into Three Address Code: if a<b then a=a+5 else a=a+7

→ if a<b then a=a+5 else a=a+7

if a<b goto L1	100: if a<b goto 102
	101: goto 103
L1: a = a+5	102: a = a+5
L2: a = a+7	103: a = a+7

→ Note: Instead of a<b, if there is a complex/compound expression, replace it with its 3 address code.

Q. Convert into Three Address Code:

→

while (i<10)	L1 : if i<10 goto L2	100: if i<10 goto 101
{	goto L.next	goto 104
x = 0	L2 : x = 0	101: x = 0
i = i+1	i = i+1	102: i = i+1
}	goto L1	103: goto 100
	L.next:	104:

Q. Write the Three Address Code:

→ if [(a<b) and ((c>d) or (a>d))]
then
z = x + y * z
else
z = z + 1

if a<b is F, goto else
if a<b is T → check c>d
↳ T => goto then F => goto a>d.
↳ T => goto then
F => goto else

```
100: if a<b goto 102
101: goto 110
102: if c<d goto 106
103: goto 104
104: if a>d goto 106
105: goto 110
106: t1 = x + y
107: t2 = t1 * z
108: z = t2
109: goto 112
110: t3 = z + 1
111: z = t3
112:
```

Q. Write the three address code for

→ while A>B and A<=2*B-5
do A = A+B

```
L1: if A>B goto L2
    goto Lend
L2: t1 = 2*B
    t2 = t1-5
    if A<=t2 goto L3
    goto Lend
L3: A = A+B
    goto L1
Lend:
```

Q. Write the three address code for:

→

```
main()
{
int i;
int a[10];
i = 1;
while (i <= 10)
{
a[i] = 0;
i = i + 1;
}
}
```

```
i = 1
L1: if i < 10 goto L2
    goto L3
L2: t1 = i * 4
    a[t1] = 0
    i = i + 1
    goto L1
L3:
```

Q. Write the three address code for:

→ while (A < C and B > D) do
if A > 1 then C = C + 1
else while A <= D do
A = A + B

```
t1 = A < C
t2 = B > D
L1: if t1 and t2 goto L2
    goto Lend
L2: if A > 1 goto L3
    goto L4
L3: C = C + 1
    goto L1
L4: if A <= D goto L5
    goto L1
L5: A = A + B
    goto L4
L1 Lend
```

15-03-'19

Q → Construct Recursive descent parser for:

bexpr → bexpr or bterm | bterm

bterm → bterm and bfactor | bfactor

bfactor → not bfactor | (bexpr) | true | false

P → P or Q | Q

Q → Q and R | R

R → not R | (P) | true | false

→ P → P or Q/Q Q → Q and R/R R → not R/(P)/ true/false

P → QP'

P' → or QP'/∈

Q → RQ'

Q' → and RQ'/∈

```
main()
{
  bexpr()
}

P()
{
  Q()
  P'()
}

P'()
{
  if(ip=='o')
    ip++;
    if(ip=='r')
      ip++;
      Q();
      P'();
}

Q
{
  R()
  Q'()
}

Q'
{
  if(ip=='a')
    ip++
    if(ip=='n')
      ip++
      if(ip=='d')
        ip++
        R()
}

R
{
  if(ip=='n')
    ip++
    if(ip=='o')
      ip++
      if(ip=='t')
        ip++
        R()
  else if(ip=='(')
    ip++
    P()
    if(ip==')')
      ip++
  else if(ip=='t')
    ip++
    if(ip=='r')
      ip++
      if(ip=='u')
        ip++
        if(ip=='e')
          ip++
  else if(ip=='f')
    ip++
    if(ip=='a')
      ip++
      if(ip=='l')
        ip++
        if(ip=='s')
          ip++
          if(ip=='e')
            ip++
}
```

Q. Show that the grammar is LR(1) but not LALR

$S \to Aa / bAc / Bc / bBa \quad A \to d \quad B \to d$

I_0: $S' \to .S, \$$
$S \to .Aa, \$$
$S \to .bAc, \$$
$S \to .Bc, \$$
$S \to .bBa, \$$
$A \to .d, a$
$B \to .d, c$

I_1: GOTO(I_0, S)
$S' \to S., \$$

I_2: GOTO(I_0, A)
$S \to A.a, \$$

I_3: GOTO(I_0, b)
$S \to b.Ac, \$$
$A \to .d, c$
$S \to b.Ba, \$$
$B \to .d, a$

I_4: GOTO(I_0, B)
$S \to B.c, \$$

I_5: GOTO(I_0, d)
$A \to d., a$
$B \to d., c$

I_6: GOTO(I_2, a)
$S \to Aa., \$$

I_7: GOTO(I_3, A)
$S \to bA.c, \$$

I_8: GOTO(I_3, d)
$A \to d., c \quad B \to d., a$

I_9: GOTO(I_3, B)
$S \to bB.a, \$$

I_{10}: GOTO(I_4, c)
$S \to Bc., \$$

I_{11}: GOTO(I_7, c)
$S \to bAc., \$$

I_{12}: GOTO(I_9, a)
$S \to bBa., \$$

1) $S \to Aa$
2) $S \to bAc$
3) $S \to Bc$
4) $S \to bBa$
5) $A \to d$
6) $B \to d$

Merge States I_5, I_8: $A \to d., a, c$
$B \to d., a, c$

Table :→ CLR

	ACTION					GOTO		
	a	b	c	d	$	S	A	B
0		S3		S5		1	2	4
1					ACCEPT			
2	S6							
3				S8			7	9
4			S10					
5	R5		R5					
6					R1			
7			S11		R2			
8	R6		R5					
9	S12							
10					R3			
11					R2			
12					R4			

✓ LR(1) grammar

✓ CLR

✗ conflict

↳ Parse bdc

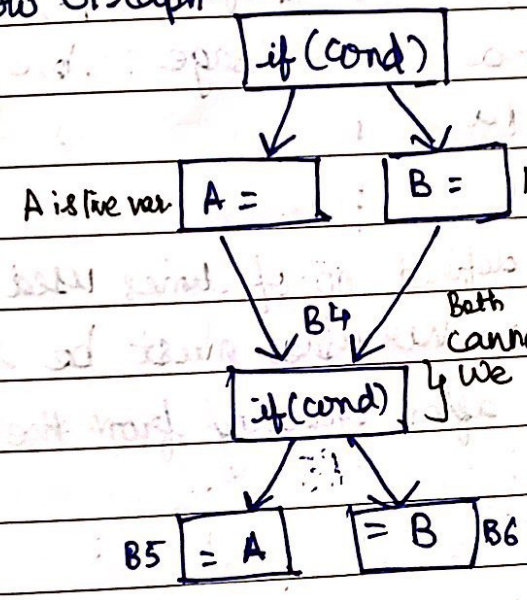
Table :→ LALR

	ACTION					GOTO		
	a	b	c	d	$	S	A	B
0		S3		S5		1	2	4
1					ACCEPT			
2	S6							
3				S8			7	9
4			S10					
58	R5,R6		R5,R6					
6					R1			
7				S11	R2			
9	S12							
10					R3			
12					R2			
12					R4			

✗ LALR grammar since R/R conflict occurs

Register Allocation and Assignment

- Allocation : Which variables are stored in register?
- Assignment : Which register is assigned ?
- Flow Graph:



- Two Methods:
  (i) Usage Counts
  (ii) Graph Colouring
- Usage Counts - Register Allocation
  • Using single loops, we allocate the registers
  (i) For every usage of a variable `v' in a basic block until it is first defined do
  savings(v) = savings(v) + 1
  After `v' is defined, it stays in the register any way and all further references are to that register.
  (ii) For every variable 'v' computed in a basic block, if it is live on exit from the basic block, count the savings of 2 since it is not necessary to store it at the end of basic block.

Σ (savings (V, B) + 2 * live and compuder (V, B))
(not live)

↳ it may be 0 or 1 (live)

↳ Total Savings Per Variable 'V'

Eg. Basic Block

a = b + c Defined: a, d, c

d = b + a Usage: b, c, a, e

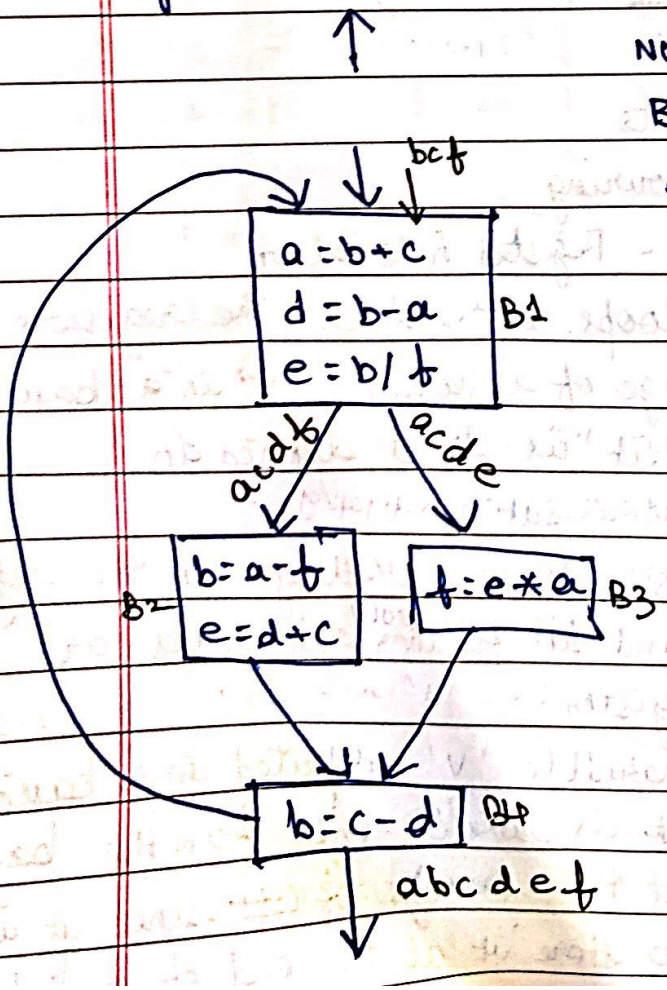
c = e + a

→ Now: b =

↳ Before defined, no. of times used = savings + 1

↳ Only live variables must be stored in the register after exiting from the basic block.

Eg.



NOTE:

Before defining 'a', we didn't use 'a' in BB1, so, 0 + 2 since it is live variable after going to next block.

We have used 'b' three times before defining it, so 3 + 0, since it is not live variable in the next block.

Savings for the Variables

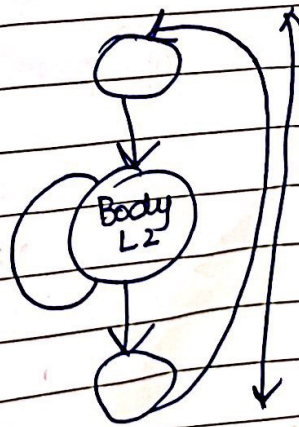
Variable	B1	B2	B3	B4	Total
a	(0+2)	+ (1+0)	+ (1+0)	+ (0+0)	= 4
b	(3+0)	+ (0+0)	+ (0+0)	+ (0+2)	= 5
c	(1+0)	+ (1+0)	+ (0+0)	+ (1+0)	= 3
d	(0+2)	+ (1+0)	+ (0+0)	+ (1+0)	= 4
e	(0+2)	+ (0+2)	+ (1+0)	+ (0+0)	= 5
f	(1+0)	+ (1+0)	+ (0+2)	+ (0+0)	= 4

→ Variables with more no. of savings = 'b', 'e' are stored in registers; then, follow lexicographic order and store 'a' in register.

∴ Store a, b, e in registers

– For Nested Loops:

- If any variable is local to L2, before exiting L2, store the values.
- If it is common to both L1 and L2, no load and store.
- If it is present in L1 and not L2, store before entering L2



* Case 1: Variables 'x', 'y', 'z' assigned registers in L2 but not in L1

→ Load x, y, z on entry to L2
Store x, y, z on exit from L2

*Case 2: Variables 'a', 'b', 'c' assigned registers in L1 but not in L2

→ Store a,b,c on entry to L2

Load a,b,c on exit to L2

* Case 3: Variables 'p' and 'q' assigned registers in both L1 and L2

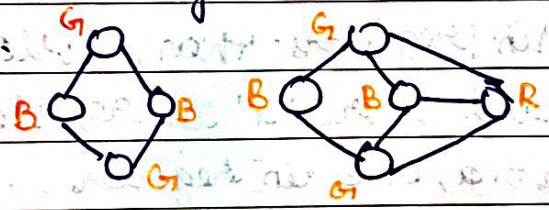
→ No load operation required

→ No store operation required

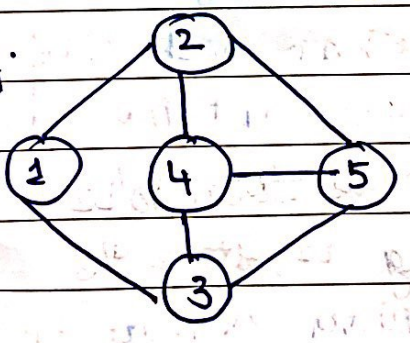
- Graph Colouring

• No two adjacent nodes must have same colour

• Eg:



• Eg.

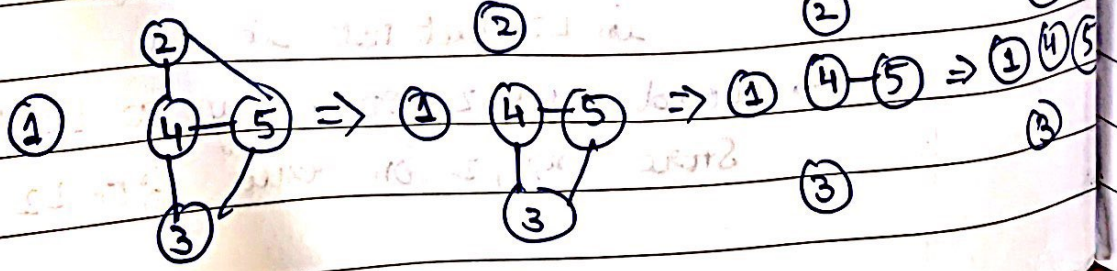


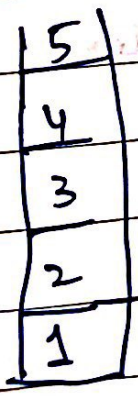
K = No. of colours required = 3

Store max. value less than K = 2

Vertex with degree 2 = no. of edges incident on node = 1

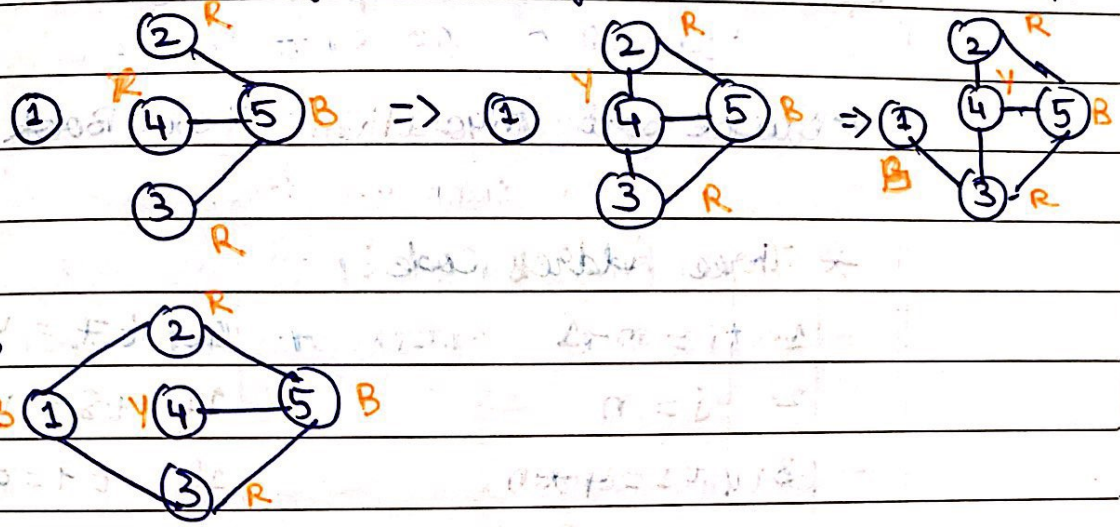
Now delete edges from 1 and push node (1) to stack.





To colour the graph:

- Pop the TOS and give it a colour, and to a node adjacent to it, give it another colour.



→ Machine Independent Optimization

- Optimization w.r.t. block - Local Optimization
- Optimization not w.r.t. block - Global Optimization
- Optimization w.r.t. target code - Target Code Optimization
- Optimization not w.r.t. target code - Machine Independent Optimization

- Data Flow Analysis: Analyze the data, live variable analysis, dead variable analysis, reaching definitions, available expressions.
- Reaching Definitions

  s = a+b

  s = exp. does it reach a point

P.T.O. →