BOTTOM-UP PARSING

Page No.
Date : / /

→ Here, from a string, we try to get a start symbol.

→ Here, we try to match our string with RHS of a production & replaces it with LHS of that production.

Two types of grammar

- Operator Precedence grammar
- LR grammar

Eg.
$S \rightarrow aABe$
$A \rightarrow Abc \mid b$
$B \rightarrow d$

i/p string abbcde

abbcde
↓
aAbcde handle
↓
aAde
↓
aABe
↓
S

→ this string belongs to the language generated by the grammar.

strings that can be derived by replacing is called handle

→ To find a suitable handle is a tough task & this technique is called handle pruning

→ Here, we get rightmost derivative in reverse order.

* All pausers of Bottom-Up-Parsing works on shift-reduce principle. & are called shift-reduce parsers.

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

$id_1 + id_2 * id_3$	$id_1 + id_2 * id_3$
$\Rightarrow E + id_2 * id_3$	$\Rightarrow E + id_2 * id_3$
$\Rightarrow E + E * id_3$	$\Rightarrow E + E * id_3$
$\Rightarrow E * id_3$	$\Rightarrow E + E * E$
$\Rightarrow E * E$	$\Rightarrow E + E$
$\Rightarrow E$	$\Rightarrow E$

Stack	i/p	Action
$	$id_1 + id_2 * id_3 \$$	shift id_1
$ id_1	$+ id_2 * id_3 \$$	Reduce $E \to id_1$
$ E	$+ id_2 * id_3 \$$	shift +
$ E +	$id_2 * id_3 \$$	shift id_2
$ E + id_2	$* id_3 \$$	Reduce $E \to id_2$
$ E + E	$* id_3 \$$	

** when top of stack is replaced by LHS, it is known as reduce action.

Here, we have problems. either we can shift * or reduce $E \to E+E$.

This situation is called "shift Reduce Conflict"

Solving further

$E+E*	id_3
$E+E*id_3	$
$E+E*E	$
$E+E	$
$E	$

start symbol → string belongs to grammar.

Operator Precedence Grammar

1). There should not be any ε production in grammar.

2). No two non-terminals can occur together

Eg. $E \to E+E \mid E-E \mid E*E \mid E/E \mid E \uparrow E \mid (E) \mid (-E) \mid id$

Step1: first we make operator-precedence Table & then derive any string using it.

- $a <\cdot b$ → b has higher precedence than a
- $a \cdot> b$ → a " " " b
- $a \doteq b$ → a has equal precedence as b.

Invalid cases:
1. id (
2.) id
3. id id
4.) (
5. $ $
6. ($
7. o $

IMP

→ $ <· a or, a ·> $

→ id ·> a [for given grammar, id has higher precedence than any terminal i.e. $\{(, *, +, \uparrow, \}$ etc.]

high precedence for same symbol → left to right associative

Eg. $E \to E+E \mid E*E \mid id$

	id	+	*	$
id	—	·>	·>	·>
+	<·	·>	<·	·>
*	<·	·>	·>	·>
$	<·	<·	<·	—

→ help to find handle

→ (end of string)

Operator precedence table.

IMP (we'll check precedence b/w any all two symbols & $ will replace them with precedence sign)

$ip \to \$ id + id * id \$$

$\Rightarrow \$ <\cdot id \cdot> + <\cdot id \cdot> * <\cdot id \cdot> \$$

$\Rightarrow \$ <\cdot id \cdot>$

$\Rightarrow \$ <\cdot + <\cdot id \cdot> * <\cdot id \cdot> \$$

$\Rightarrow \$ <\cdot + <\cdot * <\cdot id \cdot> \$$

$\Rightarrow \$ <\cdot + <\cdot * \cdot> \$$

$\Rightarrow \$ <\cdot + \cdot> \$ \to \$\$$ final

Consider <· as opening bkt $ ·> as closing. Now, whenever we get ·>, find closest <· we'll get an appropriate handle

↑ :- right associative.

Page No.
Date : / /

id * (id ↑ id) - id / id

	+	-	*	/	↑	id	(	)	$
+	⋗	⋗	⋖	⋖	⋖	⋖	⋖	⋗	⋗
-	⋗	⋗	⋖	⋖	⋖	⋖	⋖	⋗	⋗
*	⋗	⋗	⋗	⋗	⋖	⋖	⋖	⋗	⋗
/	⋗	⋗	⋗	⋗	⋖	⋖	⋖	⋗	⋗
↑	⋗	⋗	⋗	⋗	⋖	⋖	⋖	⋗	⋗
id	⋗	⋗	⋗	⋗	⋗	-	-	⋗	⋗
(	⋖	⋖	⋖	⋖	⋖	⋖	⋖	≐	-
)	⋗	⋗	⋗	⋗	⋗	-	-	⋗	⋗
$	⋖	⋖	⋖	⋖	⋖	⋖	⋖	-	-

here starting $

$ ⋖ id ⋗ * ⋖ (⋖ id ⋗ ↑ ⋖ id ⋗) ⋗ - ⋖ id ⋗ / ⋖ id ⋗ $

$ ⋖ * ⋖ (⋖ id ⋗ ↑ ⋖ id ⋗) ⋗ - ⋖ id ⋗ / ⋖ id ⋗ $

$ ⋖ * ⋖ (⋖ ↑ ⋖ id ⋗) ⋗ - ⋖ id ⋗ / ⋖ id ⋗ $

$ ⋖ * ⋖ (⋖ ↑ ⋗) ⋗ - ⋖ id ⋗ / ⋖ id ⋗ $

$ ⋖ * ⋖ (≐) ⋗ - ⋖ id ⋗ / ⋖ id ⋗ $

$ ⋖ * ⋗ - ⋖ id ⋗ / ⋖ id ⋗ $

$ ⋖ - ⋖ id ⋗ / ⋖ id ⋗ $

$ ⋖ - ⋖ / ⋖ id ⋗ $

$ ⋖ - ⋖ / ⋗ $

$ ⋖ - ⋗ $

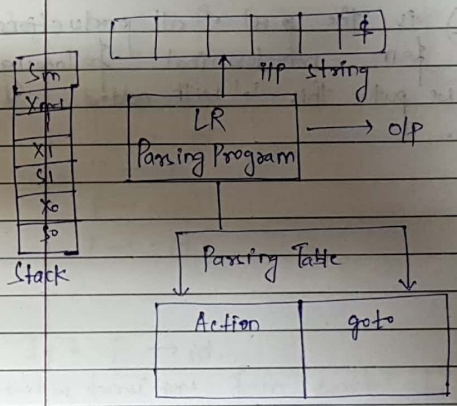
$ $

LR-Parsers

Page No.
Date : / /

LSR (Simple LR) | Canonical | LALR

* grammar generated is LR(1) only.
  - (1 lookahead symbol)
  - Left to right scanning
  - Right most derivative



* left factoring & left Recursion is not req. to be removed.
* LR grammar is a superset of LL grammar.

Augmented grammar

derive, start symbol of a given grammar with another symbol with unit production
i.e. E' → E, E is start symbol

Page No.
Date : / /

$E' \rightarrow .E$

↳ this '.' added in a production makes it an item.

Eg. $E' \rightarrow .EXa$

'.' ⇒ it mean our parser is expecting given string to be derived from production 'EXa' in this case.

Rule

#Closure (item) is the set of all productions derived from non-terminal following '.'. If we put this '.' with added productions.

Q. $E \rightarrow E+T$
$E \rightarrow T$
$T \rightarrow T*F$
$T \rightarrow F$
$F \rightarrow (E)$
$F \rightarrow id$

Note: add, $E' \rightarrow E$ to make it augmented.

Now: $I_0 : E' \rightarrow .E$

Closure (E): $E \rightarrow .E+T$, $E \rightarrow .T$

Closure (T): $T \rightarrow .T*F$, $T \rightarrow .F$

Closure (F): $F \rightarrow .(E)$, $F \rightarrow .id$

* Closure (item) ≡ together makes closure (E)

I_0 → represent initial state

all included in I_0.

• transitions from I_0 :-

(• is shifted)

transition (F)

$I_1 : E' \rightarrow E.$
$E \rightarrow E.+T$

transition (T)

$I_2 : E \rightarrow T.$
$T \rightarrow T.*F$

$I_3 : T \rightarrow F.$

$I_4 : F \rightarrow (.E)$
$E \rightarrow .E+T$
$E \rightarrow .T$
$T \rightarrow .T*F$
$T \rightarrow .F$
$F \rightarrow .(E)$
$F \rightarrow .id$

** Now closure of E

$I_5 : F \rightarrow id.$

Now state transitions from symbol I_1 :-

$I_6 : E \rightarrow E+.T$
$T \rightarrow .T*F$
$T \rightarrow .F$
$F \rightarrow .(E)$
$F \rightarrow .id$

Transition from I_2 :-

$I_7 : T \rightarrow T*.F$
$F \rightarrow .(E)$
$F \rightarrow .id$

closure of null * has

closure of * has pull.

symbols which I_0 can get for transition include the symbols following '.'. These symbols can be terminal as well as non-terminal.

make a new state for each transition symbol & add closure of each item in each state, which have been included.

(E.)
(E.+T)

transitions for I_4

I_8 : $F \to (E\cdot)$

$E \to E\cdot + T$ ≡ (1) IHP.

$E \to T\cdot$

$T \to T\cdot * F$ ∵ part of I_2 & already included.

∴ not to be included in state (I_8)

only transitions made to I_2 in dfg.

$T \to F\cdot$ ≡ I_3

$F \to (\cdot E)$

Now take closure.

{ all productions. } together ≡ I_4

only 1

I_8 : $F \to (E\cdot)$

$E \to E\cdot + T$

$F \to id\cdot$ ≡ I_5

transitions for I_6

I_9 : $E \to E + T\cdot$

$T \to T\cdot * F$

$T \to F\cdot$ ≡ I_3

$F \to (\cdot E)$ { E-closure } ≡ I_4

$F \to id\cdot$ ≡ I_5

∴ I_9 : $E \to E + T\cdot$

$T \to T\cdot * F$

transition for I_7

I_{10} : $T \to T * F\cdot$

$F \to (\cdot E)$ { E-closure } ≡ I_4

$F \to id\cdot$ ≡ I_5

∴ I_{10} : $T \to T * F\cdot$

transition for I_8

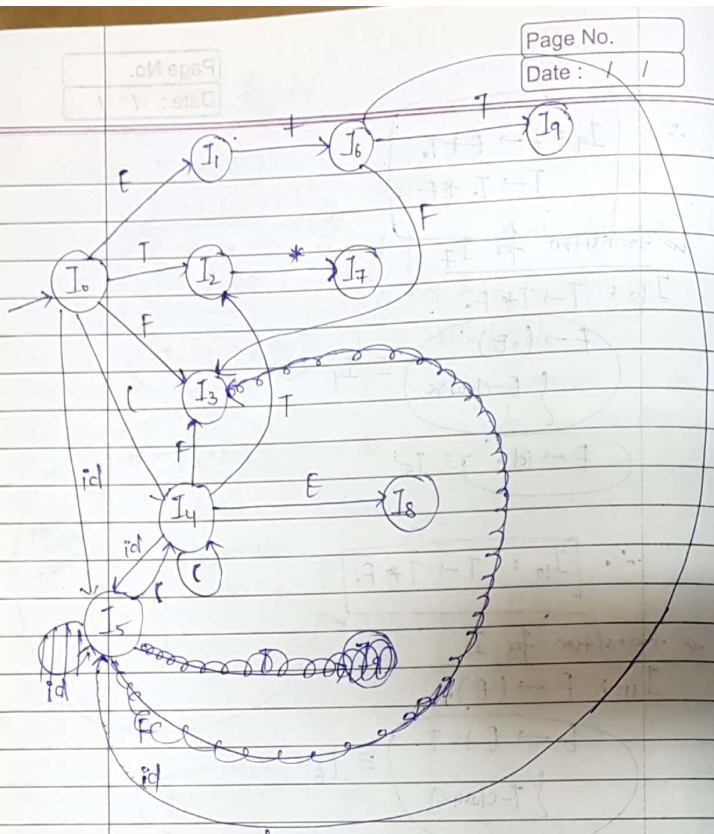
I_{11} : $F \to (E)\cdot$

$E \to E + \cdot T$ { T-closure } ≡ I_6

∴ $I_{11} = F \to (E)\cdot$

transition for I_9

I_{12} : $T \to T * \cdot F$ { F-closure } ≡ I_7



** and make further states

S' → S

Q. γ_1: S → L=R
γ_2: S → R
γ_3: L → *R
γ_4: L → id
γ_5: R → L

I_0: S' → .S
S → .L=R
S → .R
L → .*R
L → .id
R → .L

I_1: S' → S.

I_2: S → L.=R
R → L.

I_3: S → R.

I_4: L → *.R
R → .L

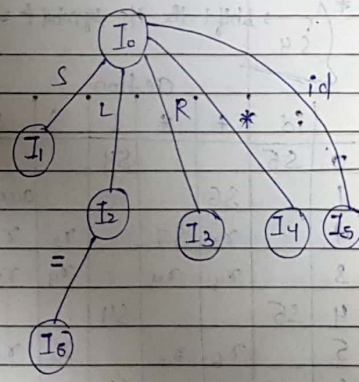
I_5: L → id.

I_6: S → L=.R
R → .L

I_7: L → *R.

I_8: R → L.

I_9: S → L=R.
for L, H will go to I_8



FIRST | FOLLOW

IMP // We'll not consider E' while constructing parsing table.

~~Follow (E')~~ =

FIRST (E) = { (, id }

FIRST (T) = { (, id }

FIRST (F) = { (, id }

Follow (E) = { $, +,) }

Follow (T) = { $, +,), * }

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

IMP // when, will get E' → S •, will make $ col = accept of that production.

* 1, 2, 11 → Represents states.

* → shift the i/p symbol to stack & move to state (4).
S4

	action						goto			
	id	+	*	(	)	$	E	T	F	
0	S5			S4			1	2	3	r1: E→E+T
1		S6				accept				r2: E→T
2		r2	S7		r2	r2				r3: T→T*F
3		r4	r4		r4	r4				r4: T→F
4	S5			S4			8	2	3	r5: F→(E)
5		r6	r6		r6	r6				r6: F→id
6	S5			S4				9	3	
7	S5			S4					10	
8		S6			S11					
9		r1	S7		r1	r1				
10		r3	r3		r3	r3				
11		r5	r5		r5	r5				

→ Name all productions

F → •id
F → id •

Page No.
Date : / /

→ if, we'll get '•' at the end (not augmented) the reduce it.

IMP //

E → T • → reduce T with E i.e. production r2 (E→T) & write it in follow of (E)

(1). FIRST (S) = { *, id }

FIRST (L) = { *, id }

FIRST (R) = { *, id }

FOLLOW (S) = { $ }

FOLLOW (L) = { =, $ }

FOLLOW (R) = { $, = }

∴ No possible conflicts

	=	*	id	$	S	L	R
0		S4	S5		1	2	3
1				accept			
2	S6, r5			r5			
3				r2			
4						8	7
5	r4			r4			9
6							
7	r3			r3			
8	r5			r5			
9				r1			

set of items.

F → T

F

A & R

S (A & R)
L = R (A & R)

Page No.
Date : / /

Q. ##

r1 : E → E+T
r2 : E → T
r3 : T → T*F
r4 : T → F
r5 : F → (E)
r6 : F → id

only states come.

Stack	I/P	Action
0	id + id * id $	Shift
0 id 5	+ id * id $	Reduce (r6)
0 F 3	+ id * id $	r4
0 T 2	+ id * id $	r2
0 E 1	+ id * id $	shift
0 E 1 + 6	id * id $	shift
0 E 1 + 6 id 5	* id $	r6
0 E 1 + 6 F 3	* id $	r4
0 E 1 + 6 T 9	* id $	shift
0 E 1 + 6 T 9 * 7	id $	shift
0 E 1 + 6 T 9 * 7 id 5	$	r6
0 E 1 + 6 T 9 * 7 F 10	$	r3
0 E 1 + 6 T 9	$	r1
0 E 1	$	accept

* on (5) at id we move to (5)

Reverse rightmost derivative

* Set of items is LR(0), ∵ No lookahead symbol is used
SLR(1)

Page No.
Date : / /

rules:

1. if, r : T → α.
We'll pop 2*len(α) from stack & will replace with LHS i.e. (T)

when, S4, (S6) comes it represents, shift i/p to stack and go to state 6.

Que. S → AS | b
A → SA | a

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

S' → S
S → AS | b
A → SA | a

I_0 : S' → .S ✓
S → .AS ✓
S → .b ✓
A → .SA ✓
A → .a ✓

I_1 : S' → S. ✓
A → S.A ✓
A → .SA ✓
A → .a ✓

I_2 : S → A.S ✓
S → .AS ✓
S → .b ✓

I_3 : A → SA. ✓

I_4 : A → S.A ✓
A → .SA ✓
A → .a ✓

I_5 : S → AS. ✓

I_6 : S → b.

I_7 : A → a.

Page No.
Date : / /

$S' \to S$

$r_1 : S \to AS$

$r_2 : S \to b$

$r_3 : A \to SA$

$r_4 : A \to a$

I_0 : $S' \to .S$, $S \to .AS$, $S \to .b$, $A \to .SA$, $A \to .a$

I_1 : $S' \to S.$, $A \to S.A$, $A \to .SA$, $A \to .a$, $S \to .AS$, $S \to .b$

I_2 : $S \to A.S$, $S \to .AS$, $S \to .b$, $A \to .SA$, $A \to .a$

I_3 : $S \to b.$

I_4 : $A \to a.$

I_5 : $A \to SA.$, $S \to A.S$, $S \to .AS$, $S \to .b$, $A \to .SA$, $A \to .a$

I_6 : $A \to S.A$, $A \to .SA$, $A \to .a$, $S \to .AS$, $S \to .b$

I_7 : $S \to AS.$, $A \to S.A$, $A \to .SA$, $A \to .a$

I_8 : $A \to SA.$, $S \to A.S$, $S \to .AS$, $S \to .b$

Page No.
Date : / /

S : a, b, $

A : a, b, $

	a	b	$	A	S
0	s3, s4	s3, s4	s3	2	1
1	s4, s3	s4, s3	accept, s3	5	6
2	s3, s4	s3, s4	s3	2	7
3	r2	r2	r2		
4	r4	r4			
5	s3, s4	s3, s4	s3	2	7
6	s3, s4	s3, s4	s3	5	6
7				5	

	a	b	$	A	S
0	s4	s3		2	1
1	s4	s3	accept	5	6
2	s4	s3		2	7
3	r2	r2	r2		
4	r4	r4	r4		
5	s4, r3	s3, r3	r3	2	7
6	s4	s3		5	6
7	r1, s4	r1, s3	r1,	5	6

CANONICAL LR PARSER

Page No.
Date : / /

$S' \to S$

$S \to CC$: r1
$C \to cC$: r2
$C \to d$: r3

→ $ will follow (S).

I_0: $S' \to .S, \$$ ✓
$S \to .CC, \$$ ✓
$C \to .cC, c/d$ ✓
$C \to .d, c/d$ ✓

I_1: $S' \to S., \$$ ✓ (LHP.)

(When making transition i.e. shifting dot, copy follow of parent production.)

I_2: $S \to C.C, \$$ ✓
$C \to .cC, \$$ ✓
$C \to .d, \$$ ✓

while taking closure, fill follow using $S \to C.C$

I_3: $C \to c.C, c/d$ ✓
$C \to .cC, c/d$ ✓
$C \to .d, c/d$ ✓

using $C \to c.C$

I_4: $C \to d., c/d$ ✓

I_5: $S \to CC., \$$ ✓

I_6: $C \to c.C, \$$ ✓
$C \to .cC, \$$ ✓
$C \to .d, \$$ ✓

* { SLR Parser is a weak Parser.
There are chances of both shift & reduce for same symbol.

→ Canonical Parser is comparitively strong than SLR.
(But no. of states in canonical parser are high.
∴ More computation)

In Canonical, to avoid shift & reduce, we use Lookahead Symbol.
∴ It has, LR(1) set of items

In SLR, we do not use Lookahead.
∴ High chances of shift and reduce.
It has, LR(0) set of items.

Diff.

CANONICAL RULES:

$A \to \alpha.B\beta, a$
$B \to \gamma$, first(β) if β is ϵ follow(A)

I_7: $C \to d., \$$ ✓

I_8: $C \to cC., c/d$ ✓

I_9: $C \to cC., \$$ ✓

	c	d	$	S	C
0	s3	s4		1	2
1			accept		
2	s6	s7			5
3	s3	s4			8
4	r3	r3			
5			r1		
6	s6	s7			9
7			r3		
8	r2	r2			
9			r2		

* LALR Parser

advantage
no. of states get reduced.

LALR Parser
(here, we will merge states whose productions are same and lookahead are diff^{n}.)

	c	d	$	S	C
0	s36	s47		1	2
1			accept		
2	s36	s47			5
36	s36	s47			89
47	r3	r3	r3		
5			r1		
89	r2	r2	r2		

* LALR is less efficient than canonical.

Semantic Analysis

⟹ When we put values of variables then it forms a annotated parse tree.

- inherited attribute : (inherit parent value)
- synthesized attribute : (combine values of two nodes)

- annotated parse tree is i/p to intermediate code generation.

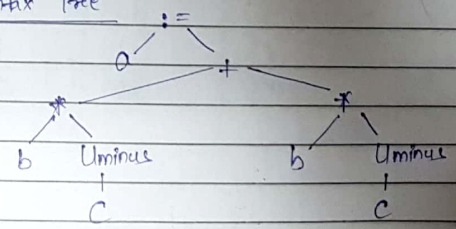
Intermediate Code Generation :-

i) Post-fix notation
ii) Syntax tree / DAG → Directed acyclic graph
iii) 3-address code

Eg. $a := b * -c + b * -c$ → unary operator.

① $abc - * bc - * + :=$ (post fix)

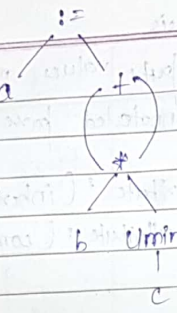
② Syntax Tree



Page No.
Date : / /

(DAG)

a).



(5). 3-Address code.

1. $t1 = -c$
2. $t2 = b * t1$
3. $t3 = -c$
4. $t4 = b * t3$
5. $t5 = t2 + t4$
6. $a = t5$

} all the temporary needs to be entered in symbol Table. (variables)

* Ways of storing 3-Address code in system.
  i) Quadruples
  ii) triples
  iii) Indirect triples

IIT Kharagpur
TV [illegible] shhhhd.

Page No.
Date : / /

(i). Quadruples (4 col)

	op	arg1	arg2	result
(0)	-	c		t1
(1)	*	b	t1	t2
(2)	-	c		t3
(3)	*	b	t3	t4
(4)	+	t2	t4	t5
(5)	=	t5		a

these are ptr to symbol table for corresponding values.

(address of rows only mentioned in case of *(a=b))

(ii) Triples (3 col)

	op	arg1	arg2
(0)	-	c	
(1)	*	b	(0)
(2)	-	c	
(3)	*	b	(2)
(4)	+	(1)	(3)
(5)	=	a	(4)

(instead of writing temporary variables, we mention address of rows containing these operations.)

Page No.
Date : / /

(iii) Indirect Triples :

(0)	(14)
(1)	(15)
(2)	(16)
(3)	(17)
(4)	(18)
(5)	(19)

	op	arg1	arg2
(14)	-	c	
(15)	*	b	(14)
(16)	-	c	
(17)	*	b	(16)
(18)	+	(1)	(17)
(19)	=	a	(18)

Eg $x[i] := y$

$x[i]$ (for LHS)

	op	arg1	arg2
(0)	[]=	x	i
(1)	=	(0)	y

$x = y[i]$

$y[i]$ (for RHS)

	op	arg1	arg2
(0)	=[]	y	i
(1)	=	x	(0)

Quadruple

$x[i] = y$

Page No.
Date : / /

	op	arg1	arg2	result
[]	=	x	i	t1
	=	y		t1

$x = y[i]$

	op	arg1	arg2	result
	=[]	y	i	t1
	=	t1		x

param x1
param x2
param x3

Call p, 3

return

return y.

y[i] = =[]

```
begin
    prod := 0;
    i := 1;
    do begin
        prod := prod + a[i] * b[i];
        i = i+1
    end
    while i<=20
end.
```

→ no need to write 3address code for keywords, such as begin, end, int, float, etc.
→ this info is already present in symbol table.

→ Allot line no. to each 3-address code.

#3-Address code :-

(1) prod = 0
(2) i = 1
~~(3) if (i>20) goto (13)~~ IMP.
(4) t_1 = i * 4 // here we are assuming, int takes 4 bytes.
(5) t2 = a[t1]
(6) t3 = i * 4 ⇒ real address is. (i*4)
(7) t4 = b[t3] ↳ (index)
(8) t5 = t2 * t4
(9) t6 = prod + t5
(10) prod = t6
(11) i = i + 1
(12) ~~goto (3)~~ if (i<=20) goto (4)
~~(13) exit~~

IMP

eg. for (i=0; i<5; i++)
{
}

(1) i = 0
(2) if (i>=5) goto (L) → this label is end for loop as to be encountered in next steps.
(3) 1- address code for body of loop.
(4) i = i+1 // inc. statement at last
(L) goto (2)
(L) Exit

→ Similarly for ~~do while and~~ while loop.
→ In case of do while include 'if' condition at last of body.

* By default assume ~~assume~~ array to be of int type i.e. 4 bytes
else, according to the given que.

Page No.
Date : / /

function

Eg:
```
int dot_product (int x[], int y[])
{
    int d, i;
    d=0;
    for (i=0 ; i<10 ; i++)
        d += x[i] * y[i];

    return d;
}
```

3 Address code :-

fn name.

(1) func begin dot product
(2) d=0
(3) i=0
(4) if (i>=10) goto (14)
(5) t1 = 4*i
(6) t2 = x[t1]
(7) t3 = 4*i
(8) t4 = y[t3]
(9) t5 = t2*t4
(10) t6 = d + t5
(11) d = t6
(12) i = i+1
(13) goto (4)
(14) return d

Page No.
Date : / /

```
void quicksort (m,n)
int m,n;                                   x
{                                          x   no 3-address code for these code lines.
int i,j;                                   x
int v,x;                                   x
if (n<=m) return;                          x
i=m-1; j=n ; v=a[n];
while (1) {
  do i=i+1; while [a[i] < v);
  do j=j-1 ; while( a[j] > v);
  if (i>=j) break;
    x=a[i]; a[i]=a[j]; a[j]=x;
}
x=a[i]; a[i]=a[n]; a[n]=x;
quicksort(m,j);
quicksort(j+1,n);
}
```

no need to write 3 address code for these lines.

(1) i = m-1
(2) j = n
(3) t1 = 4*n
(4) v = a[t1]
(5) i = i+1
(6) t2 = 4*i
(7) t3 = a[t2]
(8) if (t3 < v) goto (5)
(9) j = j-1
(10) t4 = 4*j
(11) t5 = a[t4]
(12) if (t5 > v) goto (9)

(13) if (i>=j) goto (23)
(14) t6 = 4*i
(15) x = a[t6]
(16) t7 = 4*i
(17) t8 = 4*j
(18) t9 = a[t8]
(19) a[t7] = t9
(20) t10 = 4*j
(21) a[t10] = x
(22) goto (5) // no condition is checked because while(1), i.e. loop will continue.
(23) t11 = 4*i
(24) x = a[t11]
(25) t12 = 4*i
(26) t13 = 4*n
(27) t14 = a[t13]
(28) a[t12] = t14
(29) t15 = 4*n
(30) a[t15] = x

After writing 3 address code, we create a flow chart. For this we use Basic block containing a set of statements.

→ Here we need to decide 'leader' for each block.

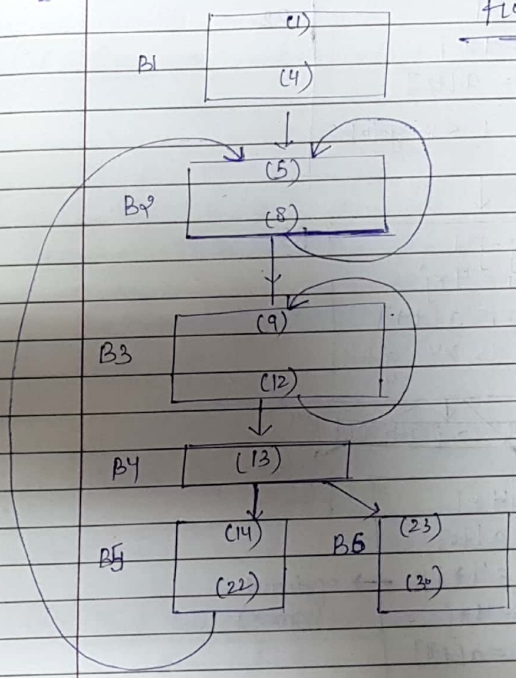
first statement in a block.

→ starting statement

→ LEADER : → target statement of conditional & unconditional statement.

→ statement next to the conditional & unconditional jump statement.

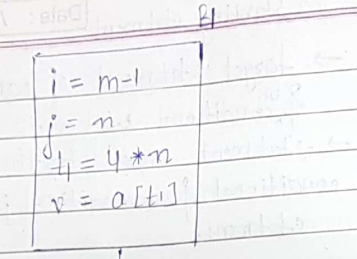
FLOW GRAPH.

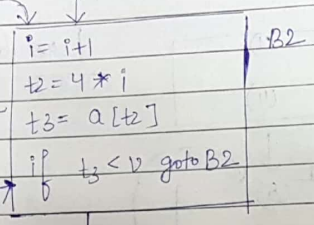


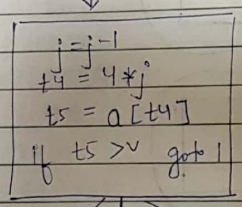
Optimisation → Local optimisation (within a block)

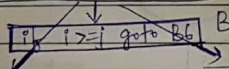
→ Global optimisation (between diff blocks)

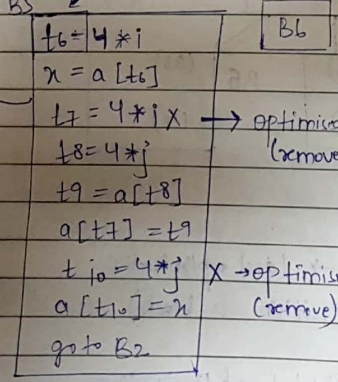
↳ if a statement is not changing in b/w









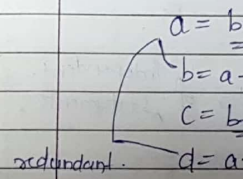


optimised (remove)

X → optimised (remove)

1. Common Subexpression elimination
2. Dead Code elimination
3. Renaming temporary variables
4. Interchange of statements.
5. Algebraic transformation

1. Common Subexpression elimination



not subexp (∵ b is changing in b/w)

redundant. as a & d are not changing.

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

2. Dead Code elimination.

```
debug = false
if (debug)
{
}
```

this code will never get executed as, debug = false in beginning only.

↓

remove dead code

Page No.
Date : / /

③. Renaming temporary variables.

$t = a + b$

↓

$u = a + b$

④. Interchange of statements

$t_1 = a + b$

$t_2 = c + q$

We can change order of independent statements.

⑤. Algebraic transformation

$x = x + 0$

$x = x * 1$

} can be removed.

$p = a^2 \rightarrow p = a * a$

time consumed is less as compared to power operation.

⑥. Copy Propagation

$f = g$

$h = f$

→ Here we try to remove occurrence of f variable.

(common subExp elimination) ↓ applying local optimisation on B5

On applying optimisation on (B5).

$\Rightarrow$

$t_6 = 4 * i$	→ X (t2)
$x = a[t_6]$	
$t_8 = 4 * j$	→ X (t4)
$t_9 = a[t_8]$	
$a[t_6] = t_9$	
$a[t_8] = x$	
goto B2	(common subExp elim^n)

↓ applying global optimisation

$x = a[t_2]$
$t_9 = a[t_4]$
$a[t_2] = t_9$
$a[t_4] = x$
goto B2

↓ global (common SubExp Elim^n)

~~$x = t_3$~~
~~$t_9 = t_5$~~
$a[t_2] = t_9$
$a[t_4] = x$
goto B2

(Copy propogation)

↓

~~$x = t_3$~~	X	
~~$t_9 = t_5$~~	→	$a[t_2] = t_5$
$a[t_2] = t_5$	(Dead code elim^n)	$a[t_4] = t_3$
$a[t_4] = t_3$		goto B2
goto B2		

Now, B6 block,

Page No.
Date : / /

$t_{11} = 4*i$ X
$n = a[t_{11}]$
$t_{12} = 4*j$ X
$t_{13} = 4*n$ X
$t_{14} = a[t_{13}]$
$a[t_{12}] = t_{14}$
$t_{15} = 4*n$ X
$a[t_{15}] = n$

$n = a[t_2]$
$t_{14} = a[t_1]$ → (can't be replaced because in while it may happen, t1 becomes equal to some other temp variable & that index value may get changed.
$a[t_2] = t_{14}$
$a[t_{15}] = n$

~~$n = a[t_2]$~~
$n = t_3$
$t_{14} = a[t_1]$
$a[t_2] = t_{14}$
$a[t_{15}] = n$

$n = t_3$
$t_{14} = a[t_1]$
$a[t_2] = t_{14}$
$a[t_{15}] = t_3$

→
$t_{14} = a[t_1]$
$a[t_2] = t_{14}$
$a[t_{15}] = t_3$

Loop Optimisation Techniques

Page No.
Date : / /

→ Code motion
→ Induction-variable elimination
→ Reduction in strength

① Code motion.
↳ loop invariant codes are shifted outside the loop

```
limit = 10
while ( i <= limit-2)
{
}
```

reduced complexity eg.

```
limit = 10
t = limit - 2
j = 2
while ( i <= t)
{
}
```

② Reduction in strength → can be in loop or elsewhere also.

eg. $p * p = p + p$

Computational cost reduces.

③ Induction-variable elimination

Page No.
Date : / /

B5
j = j - 1
t4 = 4 * j
t5 = a[t4]
if t5 > v goto B3

→ induction variables.
t4 is changing by change in (j) and every time it is reducing by diff of (4).

B4
if i >= j goto B6

j = j - 1 → can't be removed ∵ it is being compared in next block. (B4)
t4 = t4 - 4
∴ needs to be updated

alone, it is not correct.
∵ t4 is not initialised.

other blocks

B1
i = m - 1
j = n
t1 = 4 * n
v = a[t1]

B2
i = i + 1
t2 = t2 * 4
t3 = a[t2]
if t3 < v goto B2

induction.

(B7)

Final code

B1
i = m - 1
j = n
t1 = 4 * n
v = a[t1]
t2 = 4 * i
t4 = 4 * j

B2
t2 = t2 + 4
t3 = a[t2]
if t3 < v goto B2

B3
t4 = t4 - 4
t5 = a[t4]
if t5 > v goto B3

if t2 >= t4 goto B6