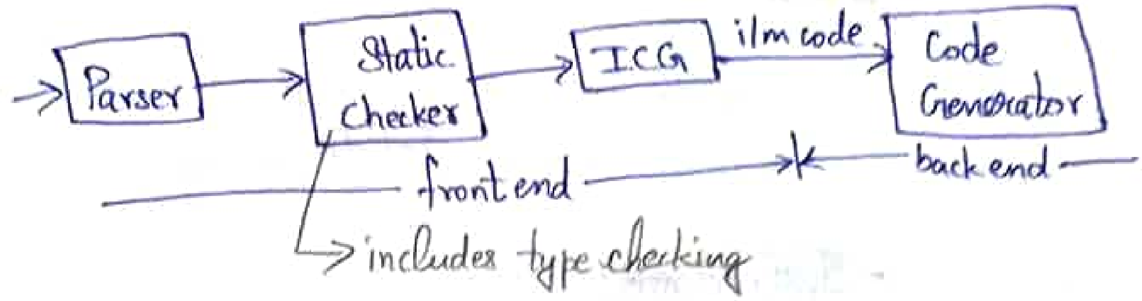
10/11/2022

Module08: Intermediate Code Generation



Intermediate representations: High-level / low-level
↳ Syntax Trees

Src Pgm → HighLvl Ilm rep. → . . . → Low Lvl Ilm rep. → Target Code

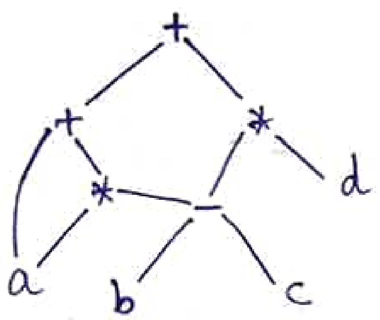
Directed Acyclic Graph (DAG):

- identifies common subexpressions of expression
- efficient syntax tree.
- Ex: DAG for $a + a * (b-c) + (b-c) * d$

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

Postfix: a a b c - * + b c - d * +

d c b - * c b - a * + a +



Three-Address Code:

→ Atmost one operator on RHS of instruction

→ 3-addr. code for prev. DAG:

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

→ Built from 2 concepts : address and instructions

* Address:
  - name
  - constant
  - compiler generated temporary

* Instructions :
  - Assignment: $x = y \text{ op } z$
    $x = \text{op } y$
  - Copy: $x = y$
  - Unconditional jump: goto L
  - Conditional jumps: if x goto L
    if False goto L
    if x relop y goto L
  - Procedure calls: param x_1
    param x_2
    ⋮
    param x_n
    call p, n // y = call p, n
    (return y)
  - Indexed copy: $x = y[i]$
    $x[i] = y$
  - Address & Pointer assignments: $x = \&y$
    $x = *y$
    $*x = y$
    sets rval(x) to lval(y)
    sets rval(x) to the contents at rval(y)
    ⇓
    y: pointer
    sets rvalue of obj. pointed by x to rvalue of y

Quadraples:
→ Called as quad.
→ 4 fields:
  - op
  - arg1
  - arg2
  - result

→ Edge cases:

- x = minus y (or) $x = y$ do not use arg2
- param operator neither use arg2 nor result
- Jumps put target label in results.

→ Ex: $a = b * -c + b * -c;$

3-addr. code:

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

Quadraples

	op	arg1	arg2	result
0	minus	c		t_1
1	*	b	t_1	t_2
2	minus	c		t_3
3	*	b	t_3	t_4
4	+	t_2	t_4	t_5
5	=	t_5		a

Triples:

→ 3 fields:
- op
- arg1
- arg2

→ Result is referred by position

→ Ex: $a = b * -c + b * -c;$

~~3-addr. code~~

→ Moving an inst requires changing all ref. to that result.

Triples

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

Indirect Triples:

→ An optimizing compiler can move an instruction by reordering inst. list without affecting triplets themselves.

→ Ex: $a = b * -c + b * -c$

instruction

35	(0)
36	(1)
37	(2)
38	(3)
39	(4)
40	(5)

indirect triples

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

Static Single-Assignment Form:

→ Ilm rep. that facilitates code optimizations.

→ All assignments in SSA are to variables with distinct names

→ Ex:

3-addr.code	SSA
$p = a + b$	$p_1 = a + b$
$q = p - c$	$q_1 = p_1 - c$
$p = q * d$	$p_2 = q_1 * d$
$p = e - p$	$p_3 = e - p_2$
$q = p + q$	$q_3 = p_3 + q_1$

→ How to handle 2 control paths for a variable value?

Ex: if (flag) $x = -1$; else $x = 1$;
$y = x * a$;

Here, SSA uses a notational convention called the ϕ-function:

$$\text{if}(flag)\ x_1 = -1;\ \text{else}\ x_2 = 1;$$
$$x_3 = \phi(x_1, x_2);$$
$$y_1 = x_3 * a;$$

Types and Declaration:

- Type checking: - uses logical rules to reason about the behaviour of a pgm at run time.
  - ensures that the type of an operand matches with the type expected by an operator.

- Translation Applications: - From type of name, compiler determine storage needed for name at runtime
  - Calculate addr. by array ref., explicit type conversions, correct version of arithmetic op.

Type Expressions (TE):

- Either a basic type or is formed applying type constructor operator.
- Basic Type: boolean, char, integer, float, void
- Type name
- Array type const. to a number and TE
- Record: DS with named fields ⇒ Applying record type const. to fieldnames & types.
- $s \to t$: function from type s to type t.
- $s \times t$ is also TE (if s and t are TE's)
- TE may contain variables whose values are TE.

Type Equivalence:

- If 2 TE are equal, then return a certain type; else error.
- Ambiguity: - when names are given to TE, and those names are used in subseq. TE.
  - Whether a name in a TE stands for itself or it is an abbr. for another TE.

- TE - represented by Graph, 2 types are structurally equivalent if & only if (one of the following is true):
  * They are same basic type
  * Formed by applying same const. to structurally equiv. types
  * One is a type name that denotes the other.

Storage Layout (for Local Names):

→ Width of a type - no. of storage units needed for obj. of that type.

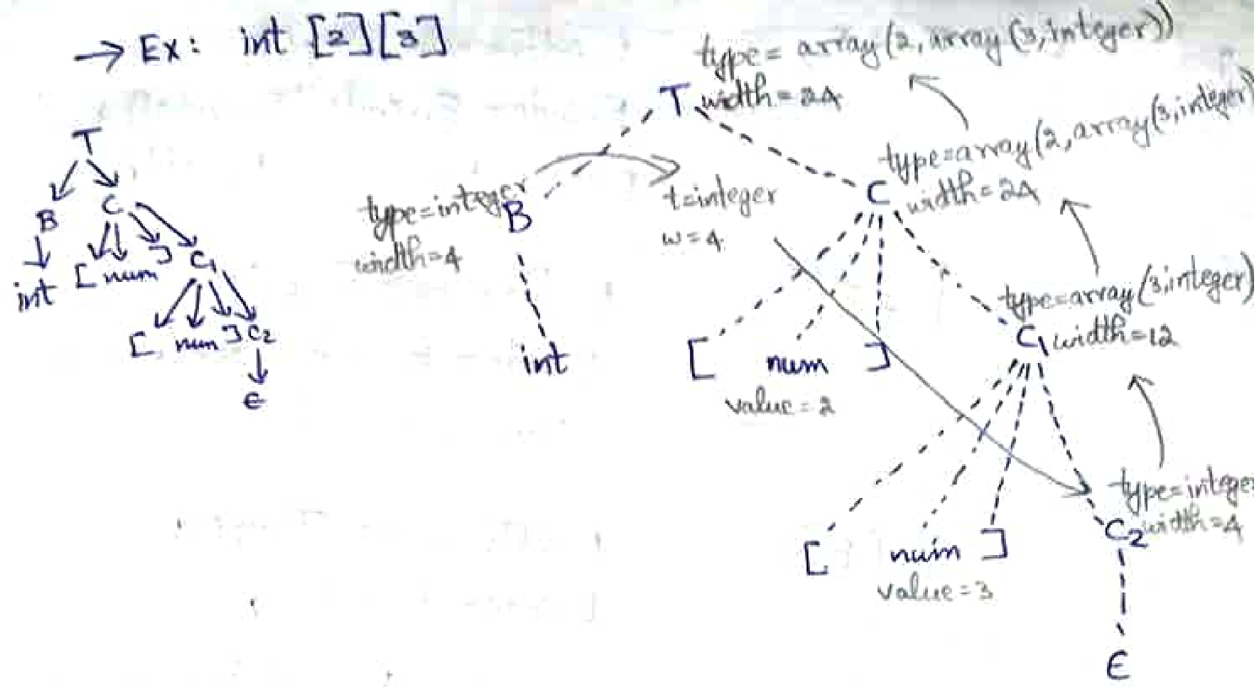
→ In SDT, syn. attr. type & width - for each NT, 2 variables t & w to pass info.

→ In SDD, t & w - inherited attr.

→ Grammar for type & width:

T → B	{ t = B.type ; w = B.width; }
C	{ T.type = C.type ; T.width = C.width; }
B → int	{ B.type = integer; B.width = 4; }
B → float	{ B.type = float; B.width = 8; }
C → ε	{ C.type = t ; C.width = w; }
C → [num] C_1	{ C.type = array(num.value, C_1.type); C.width = num.val × C_1.width; }

→ Ex: int [2][3]



Sequence of Declarations:

P → { offset = 0; } D

D → T id; { top.put (id.lexeme, T.type, offset);
offset = offset + T.width; }

D1

D → ϵ

Fields in Records and Classes:

T → record '{' { Env.push(top); top = new Env();
Stack.push(offset); offset = 0; }

D '}' { T.type = record (top); T.width = offset;
top = Env.pop(); offset = Stack.pop(); }

Translation of Expressions:

(i) Using SDD:

Production	Semantic Rules
S → id = E;	S.code = E.code || gen(top.get(id.lexeme) '=' E.addr)

$E \rightarrow E_1 + E_2$ E.addr= new Temp();
E.code = E_1.code || E_2.code || gen(
E.addr '=' E_1.addr '+' E_2.addr)

| $-E_1$ E.addr= new Temp();
E.code = E_1.code || gen(E.addr '='
'minus' E_1.addr)

| (E_1) E.addr = new Temp();
E.code = E_1.code

| id   E.addr= top.get(id.lexeme);
E.code = ''

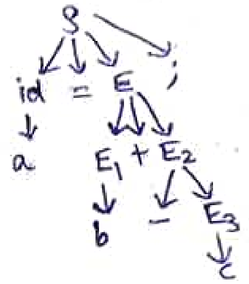
<u>Ex:</u> a=b-c

3-addr. code:

t_1 = minus c

t_2 = b+t_1

a=t_2



(ii) <u>Using SDT:</u>

- Generate 3-addr. code incrementally to avoid long string manipulations.
- Grammar:

$S \rightarrow$ id = E; {gen(top.get(id.lexeme) '=' E.addr);}

$E \rightarrow E_1 + E_2$; { E.addr= new Temp();
gen(E.addr '=' E_1.addr + E_2.addr);}

| $-E_1$ { E.addr = new Temp();
gen(E.addr '=' 'minus' E_1.addr);}

| (E_1) {E.addr = E_1.addr;}

| id {E.addr = top.get(id.lexeme);}

Addressing array elements:

→ 1D array: i^{th} element location = base + $i \times$ width

array base address

width of an element in array

→ 2D array: $A[i_1][i_2]$ location = base + $i_1 \times w_1 + i_2 \times w_2$

{row-major}

width of a row

width of an ele. in a row

→ k-D array: base + $i_1 \times w_1 + i_2 \times w_2 + \ldots + i_k \times w_k$, for $1 \le j \le k$. (w_j)

→ Location based on #elements n_j along j^{th} dimension

- for 2D: base + $i_1 \times w_1 + i_2 \times w_2$

  But, $w_1 = n_2 \times w$; $w_2 = w$

  $\therefore$ 2D : base + $(i_1 \times n_2 + i_2)\, w$

  declared $A[n_1][n_2]$

- for k-D: base + $((\ldots((i_1 \times n_2 + i_2) \times n_3 + i_3) \ldots) \times n_k + i_k) \times w$

→ With low, low+1, ..., high

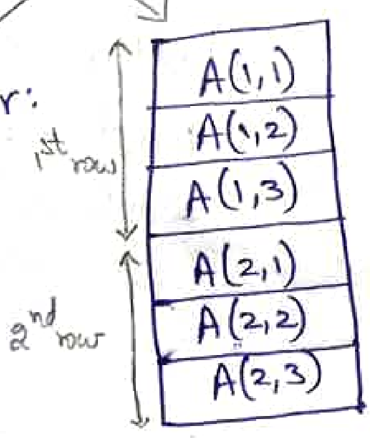
- 1D : base + $(i - \text{low}) \times$ width

  = $i \times$ width + c   // c = base − low × width

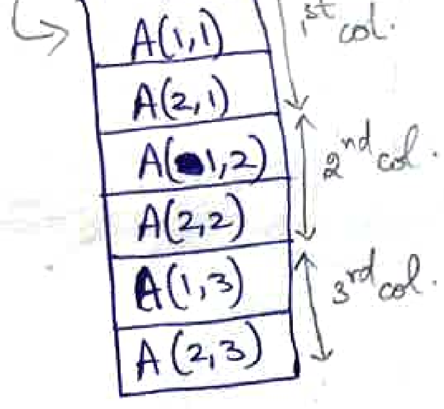
  // c = base, if low = 0

→ Row-major:

$A[2][3]$;



→ Column major.



Translation of Array references:

- Grammar:

$S \rightarrow$ id = E ; { gen (top.get(id.lexeme) '=' E.addr; }
L = E ; { gen(L.array.base '[' L.addr ']' '=' E.addr); }

$E \rightarrow E_1 + E_2$ { E.addr = newTemp();
gen (E.addr '=' E_1.addr '+' E_2.addr); }

| id { E.addr = top.get(id.lexeme); }

| L { E.addr = new Temp();
gen (E.addr '=' L.array.base '['
L.addr ']'); }

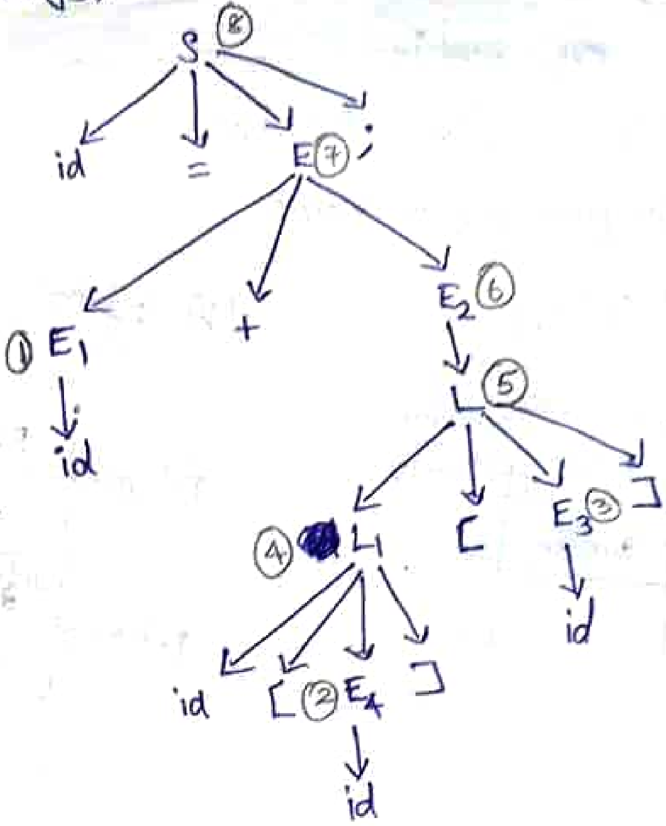
$L \rightarrow$ id [E] { L.array = top.get (id.lexeme);
L.type = L.array.type.elem;
L.addr = new Temp();
gen (L.addr '=' E.addr * L.type.width); }

| L_1 [E] { L.array = L_1.array;
L.type = L_1.type.elem;
t = new Temp();
L.addr = new Temp();
gen(t '=' E.addr '*' L.type.width);
gen(L.addr '=' L.addr '+' t); }

Intuition:

L.addr $\rightarrow$ $i_j \times w_j$
L.array $\rightarrow$ ptr to ST
L.type $\rightarrow$ type of subarr. gen. by L

Ex: $d = c + a[i][j];$



1. $E_1.addr = top.get(id.lexeme) \Rightarrow E_1.addr = c$

2. $E_4.addr = top.get(id.lexeme) \Rightarrow E_4.addr = i$,

3. $E_5.addr = top.get(id.lexeme) \Rightarrow E_5.addr = j$

4. $L_1.array = top.get(id.lexeme) \Rightarrow L_1.array = a$
   $L_1.type = L_1.array.type.element \Rightarrow L_1.type = array(3, integer)$
   $L_1.addr = t_1$
   $t_1 = E_4.addr * L_1.type.width \Rightarrow \boxed{t_1 = i * 12}$

5. $L.array = L_1.array \Rightarrow L.array = a$
   $L.type = L_1.type.elem \Rightarrow L.type = integer$
   $L.addr = t_3$
   $t_2 = E_3.addr * L.type.width \Rightarrow \boxed{t_2 = j * 4}$
   $t_3 = L.addr + t_2 \Rightarrow \boxed{t_3 = t_1 + t_2}$

6. $E_2.addr = t_4$
   $t_4 = L.array.base[L.addr] \Rightarrow \boxed{t_4 = a[t_3]}$

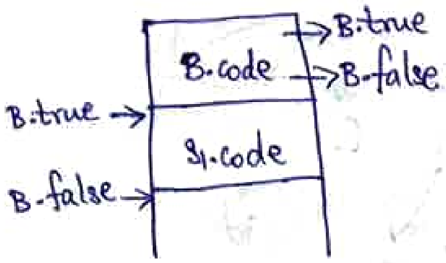
7. $E.addr = t_5$
   $t_5 = E_1.addr + E_2.addr \Rightarrow \boxed{t_5 = c + t_4}$

8. $\boxed{d = t_5}$

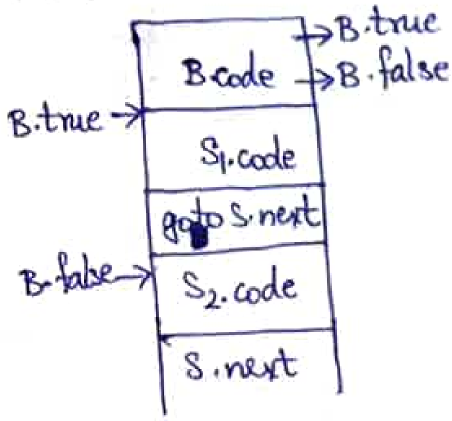
Control Flow:

→ Boolean exp. used to:
- alter the flow of control
- compute logical values

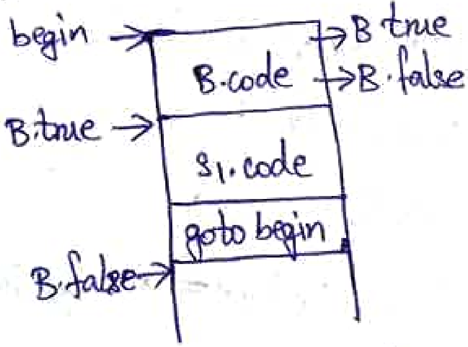
→ (a) if (B) S_1;



(b) if (B) S_1; else S_2;



(c) while (B) S_1;



→ SDD Grammar for Control-Flow Statements:

Production	Semantic Rules
$P \to S$	S.next = new label(); P.code = S.code || label(S.next);
$S \to$ assign	S.code = assign.code;
$S \to$ if(B) S_1	B.true = newlabel(); B.false = S_1.next = S.next; S.code = B.code || label(B.true) || S_1.code

$S \rightarrow$ if (B) S_1 else S_2

B.true = new label();
B.false = new label();
S_1.next = S_2.next = S.next;
S.code = B.code || label(B.true) || S_1.code || gen('goto' S.next) || label(B.false) || S_2.code;

$S \rightarrow$ while (B) S_1

begin = new label();
B.true = new label();
B.false = S.next;
S_1.next = begin;
S.code = label (begin) || B.code || label(B.true) || S_1.code || gen('goto' begin);

$S \rightarrow S_1\ S_2$

S_1.next = newlabel();
S_2.next = S.next;
S.code = S_1.code || label(S_1.next) || S_2.code;

→ SDD Grammar for Boolean Expressions:

Production	Semantic Rules
$B \rightarrow B_1 \parallel B_2$	B_1.true = B.true; B_1.false = new label(); B_2.true = B.true; B_2.false = B.false; B.code = B_1.code || label (B_1.false) || B_2.code;
$B \rightarrow B_1$ && B_2	B_1.false = B.false; B_1.true = new label(); B_2.true = B.true;

$B_2.false = B.false;$

$B.code = B_1.code \,||\, label(B_1.true) \,||\, B_2.code$

$B \rightarrow \,!B_1$

$B_1.true = B.false$

$B_1.false = B.true$

$B.code = B_1.code$

$B \rightarrow E_1 \text{ rel } E_2$

$B.code = E_1.code \,||\, E_2.code \,||$ gen('if' $E_1.addr$ rel-op $E_2.addr$ 'goto' $B.true$) || gen('goto' $B.false$);

$B \rightarrow true$ 　　 $B.code = gen('goto' \; B.true)$

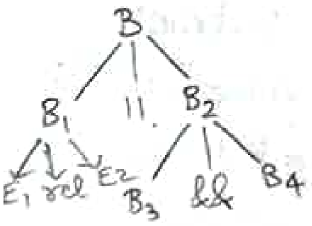
$B \rightarrow false$ 　　 $B.code = gen('goto' \; B.false)$

<u>Ex:</u> Generate 3-addr code for:

if (x<100 || x>200 && x!=y) x=0;

3-addr:

```
    if x<100 goto L2
    goto L3
L3: if x>200 goto L4
    goto L1
L4: if x!=y goto L2
    goto L1
L2: x=0;
L1:
```



Alternatively: (using iffalse)

```
    if x<100 goto L2
    iffalse x>200 goto L1
    iffalse x!=y goto L1
L2: x=0;
L1:
```

Backpatching:

→ Problem: if (B) S;

↳ contains a jump on false to S.next

↳ in 1 Pass, B is translated before S

↳ How to determine S.next?

↳ Solution: inherited attributes

↳ requires more than 1 pass.

→ Alternate solution: Backpatching - list of jumps are passed as syn. attr.

- target of jump is unspecified when created
- each jump is put on a list of jumps
- all jumps on a list have same target label.

One Pass Code Generation using Backpatching:

→ syn. attr.: B.truelist, B.falselist } list of jump inst. on which we must insert label to which control goes if B is T/F.

S.nextlist: list of jumps to inst. imm. after S

→ 3 fns:

* makelist (i): create new list
* merge (p1, p2): concatenates list p1 & p2
* backpatch (p, i): insert i as target label to each inst. pointed by p.

→ Boolean Exp. Grammar

1. $B \rightarrow B_1 \,||\, M\, B_2$ { backpatch (B_1.falselist, M.instr);
   B.truelist = merge (B_1.truelist,
   B_2.truelist);
   B.falselist = B_2.falselist; }

2. $B \rightarrow B_1 \,\&\&\, M B_2$ { backpatch (B_1.truelist, M.instr);
   B.truelist = B_2.truelist;
   B.falselist = merge(B_1.falselist,
   B_2.falselist); }

3. $B \rightarrow\ !B_1$ { B.truelist = B_1.falselist;
   B.falselist = B_1.truelist; }

4. $B \rightarrow (B)$ { B.truelist = B_1.truelist;
   B.falselist = B_1.falselist; }

5. $B \rightarrow E_1$ rel E_2 { B.truelist = makelist(nextinstr);
   B.falselist = makelist(nextinstr+1);
   gen('if' E_1.addr rel.op E_2.addr
   'goto' '_'); ||
   gen('goto _'); }

6. $B \rightarrow$ true { B.truelist = makelist(nextinstr);
   gen('goto _'); }

7. $B \rightarrow$ false { B.falselist = makelist(nextinstr);
   gen('goto _'); }

8. $M \rightarrow \epsilon$ { M.instr = nextinstr; }

→ Control Statements Grammar:

1. $S \rightarrow if(B)\ M\ S_1$ { backpatch (B.truelist, M.instr);
   S.nextlist = merge (B.falselist, S_1.nextlist); }

2. $S \rightarrow if\ (B)\ M_1 S_1\ N\ else\ M_2\ S_2$
   { backpatch (B.truelist, M_1.instr);
   backpatch (B.falselist, M_2.instr);
   temp = merge(S_1.nextlist, N.nextlist);
   S.nextlist = merge(temp, S_2.nextlist); }

3. $S \rightarrow while\ M_1\ (B)\ M_2\ S_1$
   { backpatch (S_1.nextlist, M_1.instr);
   backpatch (B.truelist, M_2.instr);
   S.nextlist = B.falselist;
   gen ('goto' M_1.instr); }

4. $S \rightarrow \{ L \}$ { S.nextlist = L.nextlist; }

5. $S \rightarrow A;$ { S.nextlist = null; }

6. $M \rightarrow \epsilon;$ { M.instr = nextinstr; }

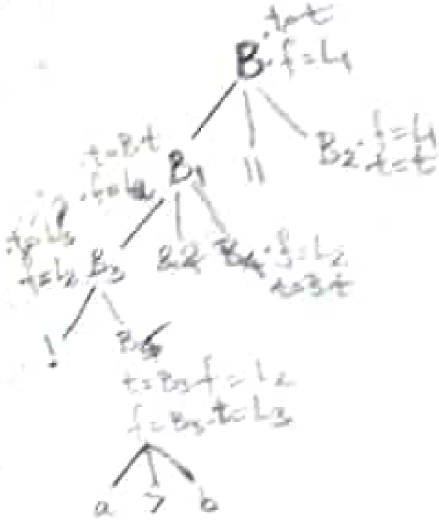
7. $N \rightarrow \epsilon;$ { N.nextlist = makelist (nextinstr);
   gen ('goto —'); }

8. $L \rightarrow L_1 M S$ { backpatch (L_1.nextlist, M.instr);
   L.nextlist = S.nextlist; }

9. $L \rightarrow S$ { L.nextlist = S.nextlist; }

Ex: Generate 3 addr code for the following code with & without backpatching:

```
a=c*d
if((~ (a>b)) && (c<=d) || (f!=g))
        a=-f*a
else
        a=b++
p=a+s
```



Ans: (i) without Backpatching:

a = c*d

100: if a>b goto L_2

101: goto L_3

102: L_3: if c<=d goto B.true

103: goto L_2

104: L_2: if f!=g goto B.true

105: goto L_1

106: B.true: t_1 = minus f

107: a = t_1 * a

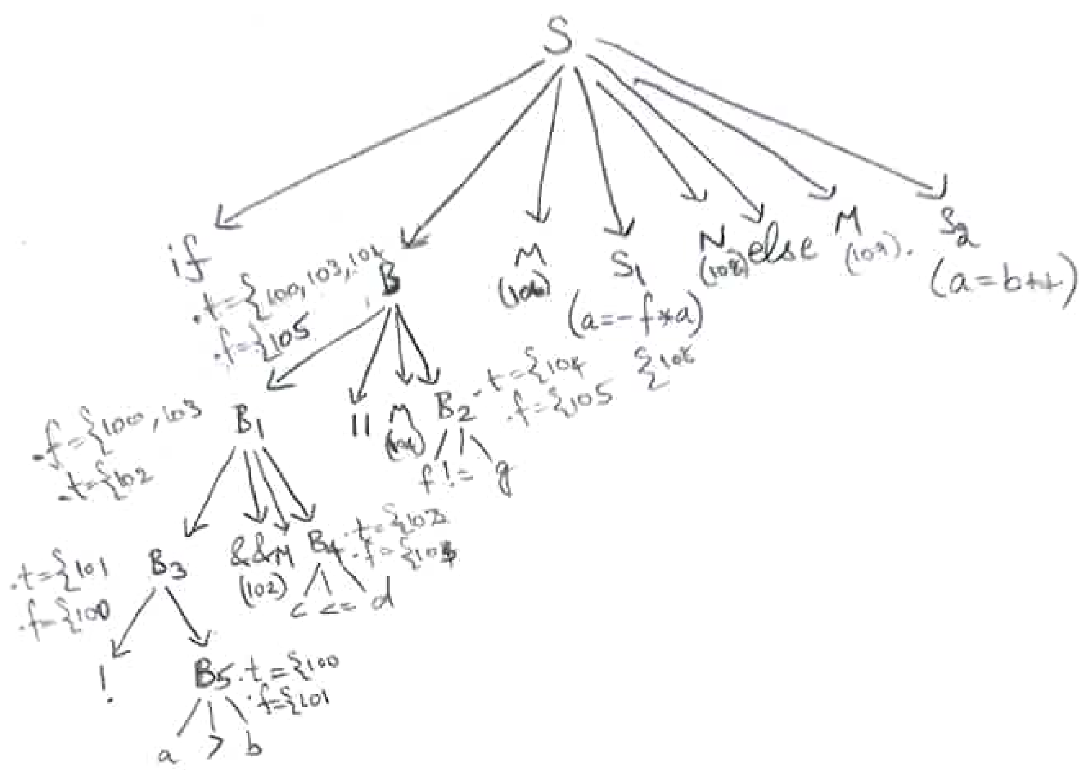
108: goto L_0

109: L_1: a = b++

110: L_0: p = a+s

111:

(ii) with backpatching:



99: a = c*d

100: if a>b goto 104

101: goto 102

102: if c <= d goto ___

103: goto 104

104: if f != g goto ___

105: goto 109

106: t_1 = minus f

107: a = t_1 * a

108: goto ___

109: a = b++

110: p = a+s