CSE 4102

Intermediate Code Generation
Lecture 12

Compilers principles, techniques, & tools-ULLMAN
Chapter 06

Control Flow

- Translation of conditional statements is tied to translation of Boolean expressions.
- Boolean expressions are used to
  - Alter the flow of control, e.g. if (E) S
  - Compute logical values
- Evaluated in analogy to arithmetic expressions
- Intended use of Boolean expression is determined from its syntactic context
  - Expression follows the keyword **if**
    - Alter the flow of control
  - Expression on the right side of an assignment
    - Denote a logical value

Boolean Expression

- Boolean operators
  - '&&' (AND) , '||' (OR) , '!' (NOT)
- Relational expressions
  - E_1 **rel** E_2
    - E_1 and E_2 are arithmetic expressions
    - **rel**.op : <, <=, =, !=, >, >=
- Grammar for Boolean Expression

B→ B || B
| B && B
| !B
| (B)
| E **rel** E
| **true**
| **false**

- We assume that | | and && are left associative
  - | | has the lowest precedence
  - then &&
  - then !

Short-Circuit Code

- Boolean expressions are typically used in the flow of control statements, such as:
  - if, while and for statements, the effect of such boolean expression can be represented by the position of the program after the expression is evaluated.
- Jump code can be directly generated without evaluating the expressions explicitly.

Short-Circuit Code

- IF B → B_1 || B_2 and B_1 is **true** then B is **true**
  - We can omit evaluation of B_2
- IF B → B_1 && B_2 and B_1 is **false** then B is **false**
  - We can omit evaluation of B_2

- Semantic definitions of language determines whether all parts of a Boolean expression must be evaluated

Short-Circuit Code

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

- ```
        if x < 100 goto L2
        ifFalse x > 200 goto L1
        ifFalse x != y goto L1
  L2:   x = 0
  L1:
  ```

NOTE: Here all the Boolean operators &&, ||, ! are translated into jumps
  ```

Short-Circuit Code

```
if (x<y)
      z=x;
else
     z=y;
z=z*z
```

```
  t0 = x < y
  if t0 goto L0
  z = x
  goto L1
L0:
  z=y
L1:
  z = z*z
```

Short-Circuit Code

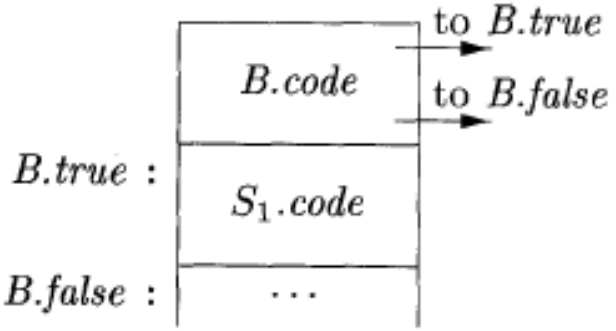
```
while (x < y) {
x = x * 2;
}
y = x;
```

```
L0:
  t0 = x < y
  IfF t0 goto L1
  x = x * 2
  Goto L0
L1:
```

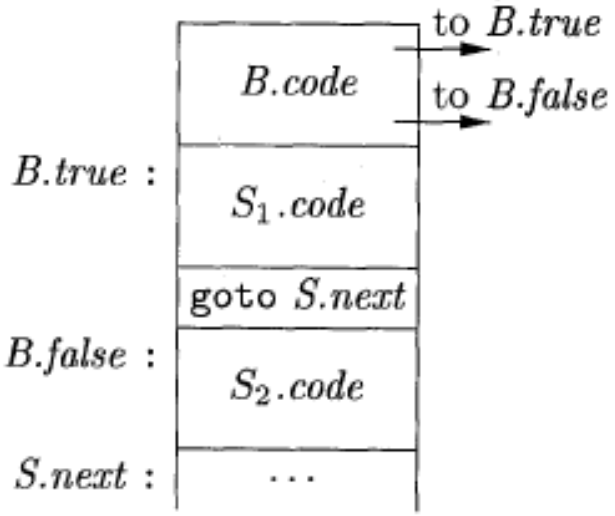
Flow-of-Control Statements

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

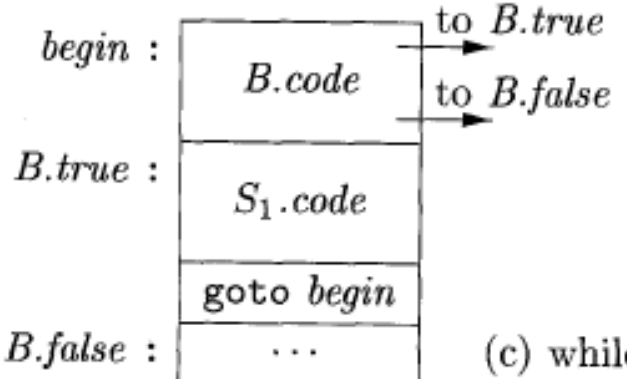
- B and S has synthesized attribute *code*
- Within B.code jumps are based on value of B



(a) if



(b) if-else



(c) while

Syntax directed definition for flow-of-control statements

PRODUCTION	SEMANTIC RULES
$P \rightarrow S$	$S.next = newlabel()$ $P.code = S.code \mid\mid label(S.next)$
$S \rightarrow$ **assign**	$S.code =$ **assign**$.code$
$S \rightarrow$ **if** (B) S_1	$B.true = newlabel()$ $B.false = S_1.next = S.next$ $S.code = B.code \mid\mid label(B.true) \mid\mid S_1.code$
$S \rightarrow$ **if** (B) S_1 **else** S_2	$B.true = newlabel()$ $B.false = newlabel()$ $S_1.next = S_2.next = S.next$ $S.code = B.code$ $\mid\mid label(B.true) \mid\mid S_1.code$ $\mid\mid gen('\texttt{goto}'\ S.next)$ $\mid\mid label(B.false) \mid\mid S_2.code$
$S \rightarrow$ **while** (B) S_1	$begin = newlabel()$ $B.true = newlabel()$ $B.false = S.next$ $S_1.next = begin$ $S.code = label(begin) \mid\mid B.code$ $\mid\mid label(B.true) \mid\mid S_1.code$ $\mid\mid gen('\texttt{goto}'\ begin)$
$S \rightarrow S_1\ S_2$	$S_1.next = newlabel()$ $S_2.next = S.next$ $S.code = S_1.code \mid\mid label(S_1.next) \mid\mid S_2.code$

Generating three-address code for booleans

PRODUCTION	SEMANTIC RULES
$B \rightarrow B_1 \mid\mid B_2$	$B_1.true = B.true$ $B_1.false = newlabel()$ $B_2.true = B.true$ $B_2.false = B.false$ $B.code = B_1.code \mid\mid label(B_1.false) \mid\mid B_2.code$
$B \rightarrow B_1 \;\&\&\; B_2$	$B_1.true = newlabel()$ $B_1.false = B.false$ $B_2.true = B.true$ $B_2.false = B.false$ $B.code = B_1.code \mid\mid label(B_1.true) \mid\mid B_2.code$
$B \rightarrow ! B_1$	$B_1.true = B.false$ $B_1.false = B.true$ $B.code = B_1.code$
$B \rightarrow E_1$ **rel** E_2	$B.code = E_1.code \mid\mid E_2.code$ $\mid\mid gen('\texttt{if}'\ E_1.addr\ \textbf{rel}.op\ E_2.addr\ '\texttt{goto}'\ B.true)$ $\mid\mid gen('\texttt{goto}'\ B.false)$
$B \rightarrow$ **true**	$B.code = gen('\texttt{goto}'\ B.true)$
$B \rightarrow$ **false**	$B.code = gen('\texttt{goto}'\ B.false)$

Example

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

$S \rightarrow$ if E then S_1	E.true := newlabel; E.false := S.next; S_1.next := S.next; S.code := E.code || gen(E.true ':') || S_1.code
$E \rightarrow E_1$ or E_2	E_1.true := E.true; E_1.false := newlabel; E_2.true := E.true; E_2.false := E.false; E.code := E_1.code || gen(E_1.false ':') || E_2.code
$E \rightarrow E_1$ and E_2	E_1.true := newlabel; E_1.false := E.false; E_2.true := E.true; E_2.false := E.false; E.code := E_1.code || gen(E_1.true ':') || E_2.code
$E \rightarrow id_1$ relop id_2	E.code := gen('if' id.place relop.op id_2.place 'goto' E.true) || gen('goto' E.false)

```
        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 :
```

Example

```
while a < b do
  if c < d then
    x := y + z
  else
    x := y - z
```

```
L1:     if a < b goto L2
        goto Lnext
L2:     if c < d goto L3
        goto L4
L3:     t1 := y + z
        x:= t1
        goto L1
L4:     t2 := y – z
        x := t2
        goto L1
Lnext:
```

$S \rightarrow$ while E do S_1

```
S.begin := newlabel;
E.true := newlabel;
E.false := S.next;
S1.next := S.begin;
S.code := gen(S.begin ':') || E.code ||
          gen(E.true ':') || S1.code ||
          gen('goto' S.begin)
```

$S \rightarrow$ if E then S_1 else S_2

```
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
```

$E \rightarrow id_1$ relop id_2

```
E.code := gen('if' id.place
  relop.op id2.place 'goto'
  E.true) ||
  gen('goto' E.false)
```

Avoiding Redundant Gotos

```
        if x > 200 goto L4
        goto L1
L4
```

```
ifFalse x > 200 goto L1
```

Avoiding Redundant Gotos

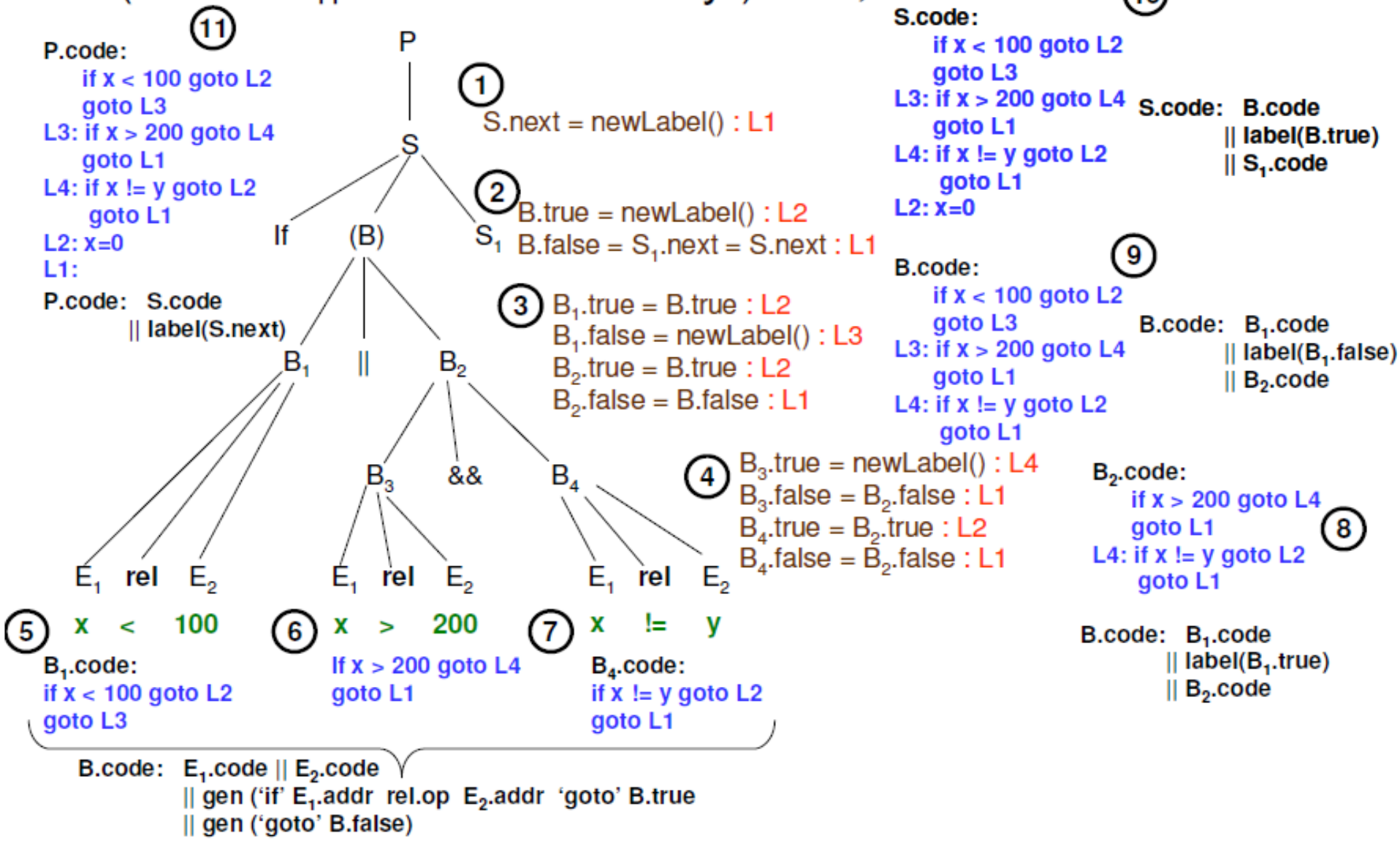
```
if( x < 100 || x > 200 && x != y ) x = 0;
```

```
      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:
```

```
      if x < 100 goto L2
      ifFalse x > 200 goto L1
      ifFalse x != y goto L1
L2    x   0
L1
```

Example

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



- A key problem when generating code for *boolean expressions and flow-of-control statements* is that of matching a jump instruction with the target of the jump.
- For example, if (B) S
  - when B is false, to the instruction following the code for S
    - B must be translated before S is examined.
  - what then is the target of the goto that jumps over the code for S?
  - we addressed this problem by passing labels as inherited attributes to where the relevant jump instructions were generated.
  - but a separate pass is then needed to bind labels to addresses.

Backpatching

- *Backpatching* can be used to generate code for Boolean expressions and flow of-control statements in one pass.

- Generate branching statements with the targets of the jumps temporarily unspecified
- Put each of these statements into a list which is then filled in when the proper label is determined

Backpatching

```
108: t0 = true
109: if t0 goto 111
110: goto _
111: ...
122: goto 108
123: ...
```

- backpatch({110}, 123)

Keep track of incomplete jump instructions

Backpatch when information is available

Backpatching

- We maintain a list of statements that need patching by future statements
- Three lists are maintained:
  - truelist: for targets when evaluation is true
  - falselist: for targets when evaluation is false
  - nextlist: list of jumps to the instruction immediately following the code for S

Synthesized attributes of nonterminal B

- These lists can be implemented as a synthesized attribute
- Assume instructions are generated into an instruction arrays

Back-patching

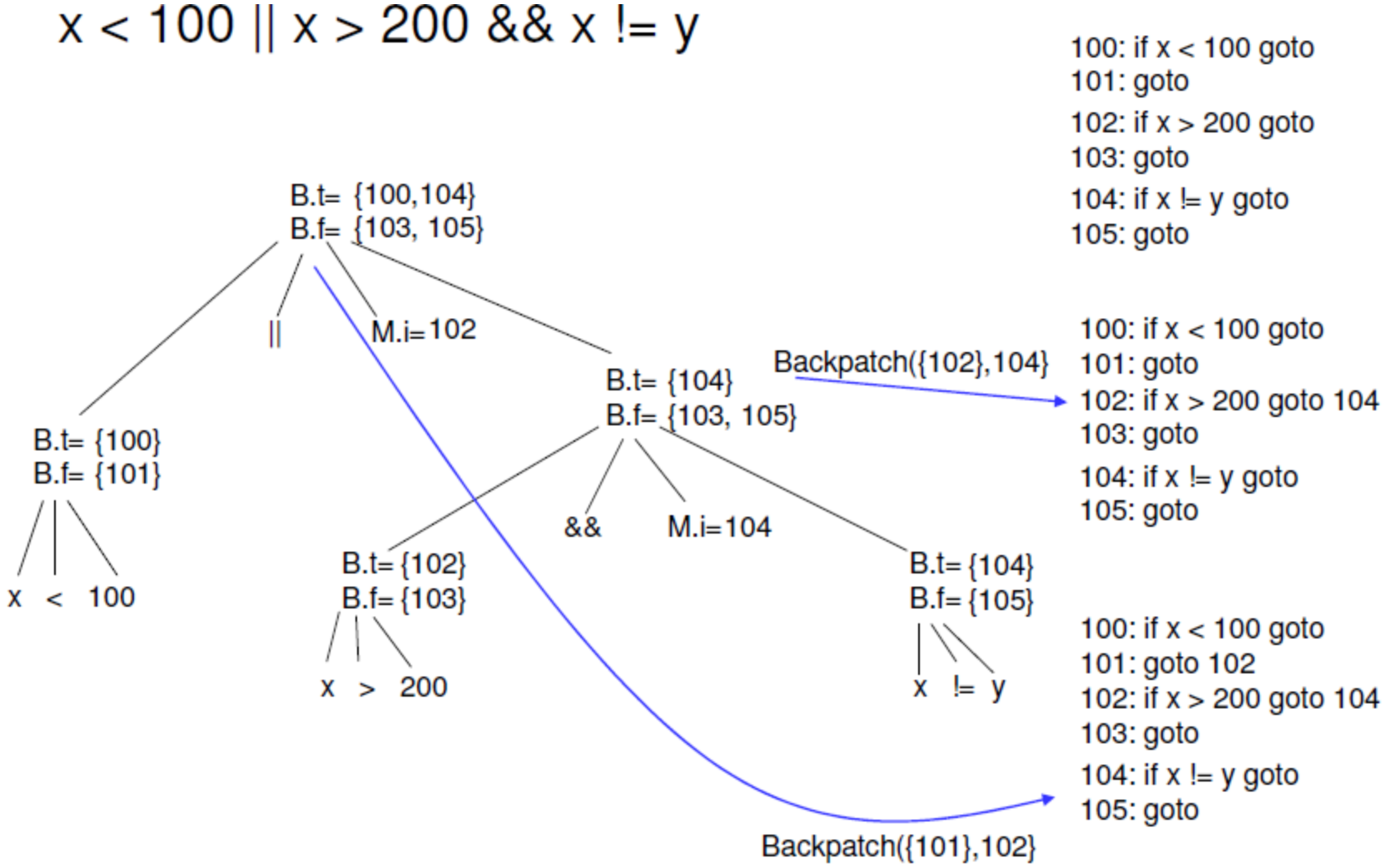
- For non-terminal B we use two attributes B.truelist and B.falselist together with following functions:
  - makelist(i): create a new list containing only i, an index into the array of instructions
  - Merge(p1,p2): concatenates the lists pointed by p1 and p2 and returns a pointer to the concatenated list
  - Backpatch(p,i): inserts i as the target label for each of the instruction on the list pointed to by p

Backpatching for Boolean Expressions

$B \rightarrow B_1 \mid\mid M\ B_2$	{backpatch(B_1.falselist, M.instr); B.truelist = merge(B_1.truelist,B_2.truelist); B.falselist=B_2.falselist;}
$B \rightarrow B_1$ && $M\ B_2$	{backpatch(B_1.truelist, M.instr); B.truelist = B_2.truelist; B.falselist= merge(B_1.falselist,B_2.falselist);}
$B \rightarrow !\ B_1$	{B.truelist = B_1.falselist; B.falselist= B_1.truelist;}
$B \rightarrow (\ B_1\ )$	{B.truelist = B_1.truelist; B.falselist= B_1.falselist;}
$B \rightarrow E_1$ **rel** E_2	{B.truelist = makelist(nextinstr); B.falselist = makelist(nextinstr+1); emit('if' E_1.addr **rel**.op E_2.addr 'goto _') emit('goto _')}
B → **true**	{B.truelist = makelist(nextinstr); emit ('goto _');}
B → **false**	{B.false = makelist(nextinstr); emit ('goto _');}
M → ε	{ M.instr = nextinstr;}

Backpatching for Boolean Expressions

- Annotated parse tree for x < 100 || x > 200 && x ! = y



Back-patching for flow of control statements

$S \rightarrow$ if (B) M S_1	{backpatch(B.truelist, M.instr); S.nextlist = merge(B.falselist,S_1.nextlist);}
$S \rightarrow$ 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);}
$S \rightarrow$ 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; emit ('goto' M_1.instr);}
$S \rightarrow$ { L }	{S.nextlist = L.nextlist;}
$S \rightarrow$ A;	{S.nextlist = **null**;}
$M \rightarrow \varepsilon$	{M.instr=nextinstr;}
$N \rightarrow \varepsilon$	{N.nextlist = makelist(nextinstr); emit ('goto _');}
$L \rightarrow L_1$ M S	{backpatch(L1.nextlist, M.instr); L.nextlist = S.nextlist;}
$L \rightarrow S$	{ L.nextlist = S.nextlist;}

Translation of a switch-statement

```
switch ( E ) {
    case V1: S1
    case V2: S2
        ...
    case Vn-1: Sn-1
    default: Sn
}
```

```
        code to evaluate E into t
        goto test
L1:     code for S1
        goto next
L2:     code for S2
        goto next
        ...
Ln-1:   code for Sn-1
        goto next
Ln:     code for Sn
        goto next
test:   if t = V1 goto L1
        if t = V2 goto L2
        ...
        if t = Vn-1 goto Ln-1
        goto Ln
next:
```

```
        code to evaluate E into t
        if t != V1 goto L1
        code for S1
        goto next
L1:     if t != V2 goto L2
        code for S2
        goto next
L2:
        ...
Ln-2:   if t != Vn-1 goto Ln-1
        code for Sn-1
        goto next
Ln-1:   code for Sn
next:
```

Back-patching Practice

```
if(m>2 && m<50 && m%3==0){
        a=a+2;
        b++;
        m/=3
}
else{
       a=a+5;
       b--;
}
```

$B \rightarrow B_1$ && $M\ B_2$

{backpatch(B_1.truelist, M.instr);
B.truelist = B_2.truelist;
B.falselist= merge(B_1.falselist,B_2.falselist);}

$S \rightarrow$ 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);}

Any Question?