CS536

Control Flow

A Sahu

CSE, IIT Guwahati

Outline

- Control Flow: Flow Graph
- Short-Circuit Code
- Flow of Control Statement
- Back Patching
- Translation of Switch Statement
- Translation of function call

Control Flow

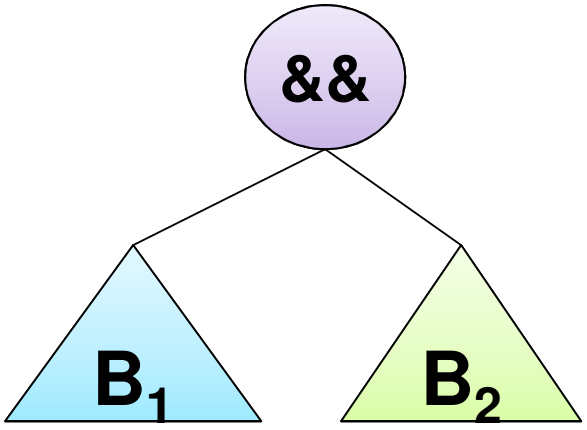
Boolean expressions are often used to:

- *Alter the flow of control.*
- *Compute logical values.*

Short-Circuit Code

- Given an expression B1 || B2
  - If we determine B1 is true, we can conclude B is true, without evaluating B2
- Given an expression B1 && B2
  - If we determine B1 is false, we can conclude B is false, without evaluating B2
- If the language permit portion of a Boolean expr to go unevaluated: to optimize
  - **Side effect: if B2 contain a function that change global variable, then unexpected answer be obtained**

Short Circuit Evaluation



codeGen_bool (B, *trueDst*, *falseDst*):

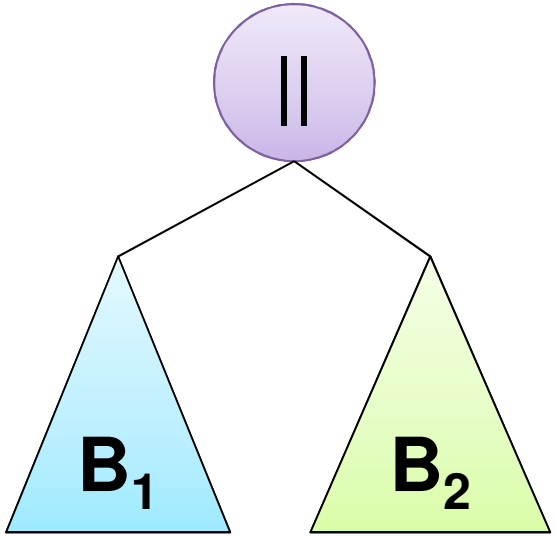
/* recursive case 1: B.nodetype == '&&' */

L_1 = *newlabel*( );

codeGen_bool(B_1, L_1, *falseDst*);

codeGen_bool(B_2, *trueDst*, *falseDst*);

B.code = B_1.code ⊕ L_1 ⊕ B_2.code;



codeGen_bool (B, *trueDst*, *falseDst*):

/* recursive case 2: B.nodetype == '||' */

L_1 = *newlabel*( );

codeGen_bool(B_1, *trueDst*, L_1);

codeGen_bool(B_2, *trueDst*, *falseDst*);

B.code = B_1.code ⊕ L_1 ⊕ B_2.code;

Backpatching

- 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.
- Backpatching uses
  - lists of jumps which are passed as synthesized attributes.
- Specifically, when a jump is generated, the target of the jump is temporarily left unspecified.
  - Each such jump is put on a list of jumps whose labels are to be filled in when the proper label can be determined.

Code Generation using Backpatching

- Generate instructions into an instruction array, and labels will be indices into this array. To manipulate lists of jumps, three functions are used:
  - **makelist(i)** creates a new list containing only i, an index into the array of instructions; makelist returns a pointer to the newly created list.
  - **merge(pl , p2)** concatenates the lists pointed to by pl and p2 , and returns a pointer to the concatenated list.
  - **backpatch(p, i)** inserts i as the target label for each of the instructions on the list pointed to by p.

Backpatching Example

If (a < b) then I := I+1 else j:= I+1

100: if a < b then goto ???
101: goto ???
102: t1 = I+1
103: I = t1
104: goto ???
105: t1 = I+1
106: j = t1
107:

Backpatching Example

If (a < b) then I := I+1 else j:= I+1

```
100:   if a < b then goto ???    //102
101:   goto ???                  //105
102:   t1 = I+1
103:   I = t1
104:   goto ???                  //107
105:   t1 = I+1
106:   j = t1
107:
```

Backpatching Example

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

100: If (x < 100) goto ???

101: goto ???

102: if x > 200 goto ???

103: goto ???

104: If x !=y goto ???

105: goto ???

106: x =0

Boolean Expression : Production

- Boolean expressions

  E->E1 or M E2 E->E1 and M E2

  E-> not E1 E-> (E1)

  E->id1 relop id2

  E->true E->false

  M->ϵ

- E has two attributes truelist and falselist to store the list of goto instructions with empty destinations.
  - Truelist: goto TRUELABEL
  - Falselist: goto FALSELABLE
- M.quad: the number for current instruction
- Makelist(quad): create a list.

Backpatching for Boolean Expressions

E->E1 or M E2 { backpatch(E1.falselist, M.qual);
E.truelist = merge(E1.truelist, E2.truelist);
E.falselist = E2.falselist;}

E->E1 and M E2 {backpatch(E1.truelist, M.qual);
E.truelist = E2.truelist;
E.falselist = merge(E1.truelist, E2.truelist);}

E-> not E1 {E.truelist = E1.falselist,
E.falselist = E1.truelist;}

E-> (E1) {E.truelist = E1.truelist;
E.falselist = E1.falselist;}

Backpatching for Boolean Expressions

E->id1 relop id2 {E.truelist = makelist(nextquad);

E.falselist = makelist(nextquad+1);

emit('if' id1.place relop.op id2.place 'goto **???**');

emit('goto **???**'); }

E->true {E.truelist = makelist(nextquad);

emit('goto **???**');}

E->false {E.falselist = makelist(nextquad);

emit('goto **???**');}

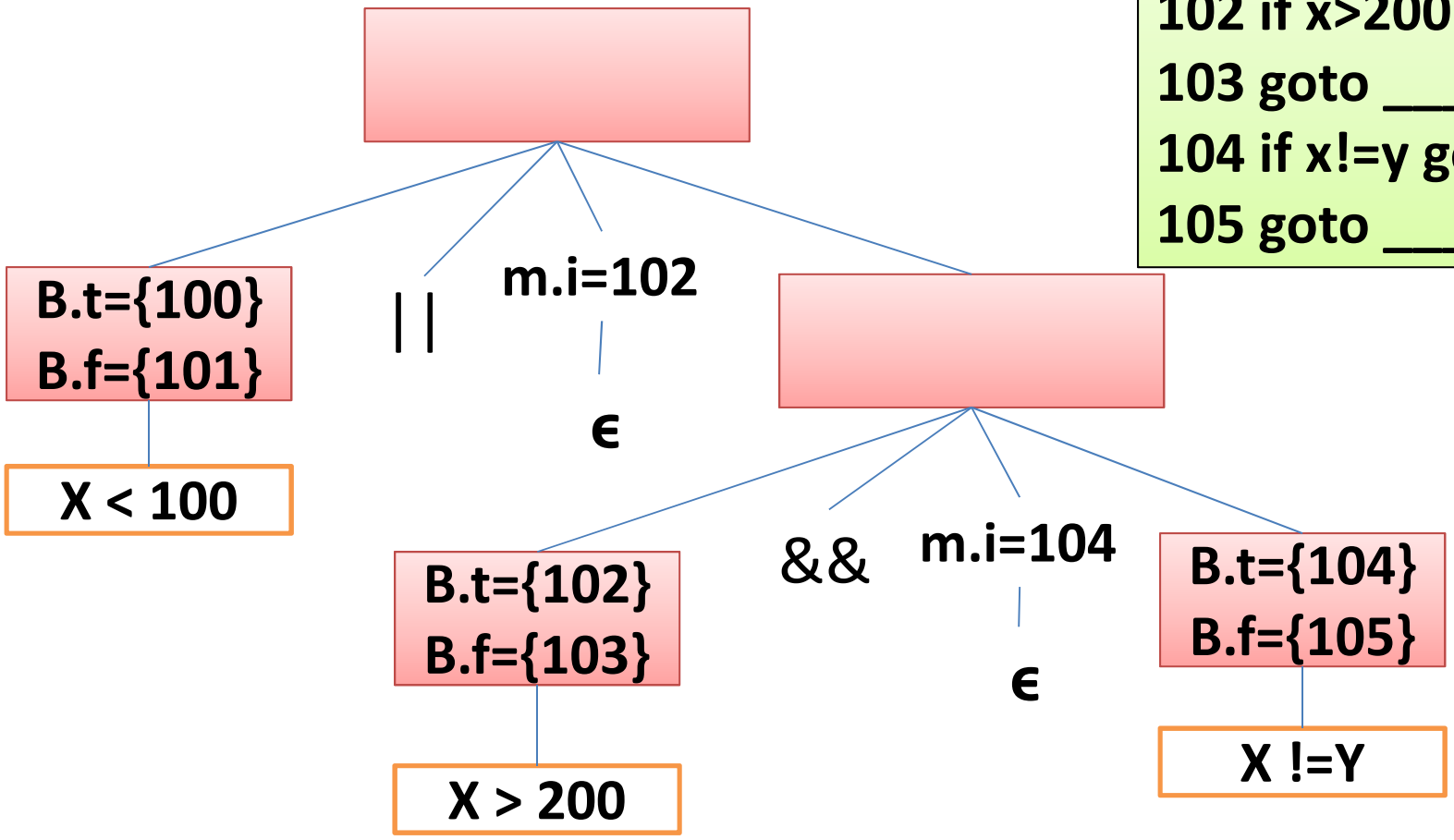
M-> ϵ {M.quad = nextquad;}

nextquad/nextinstr

Backpatching for Boolean Expressions

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

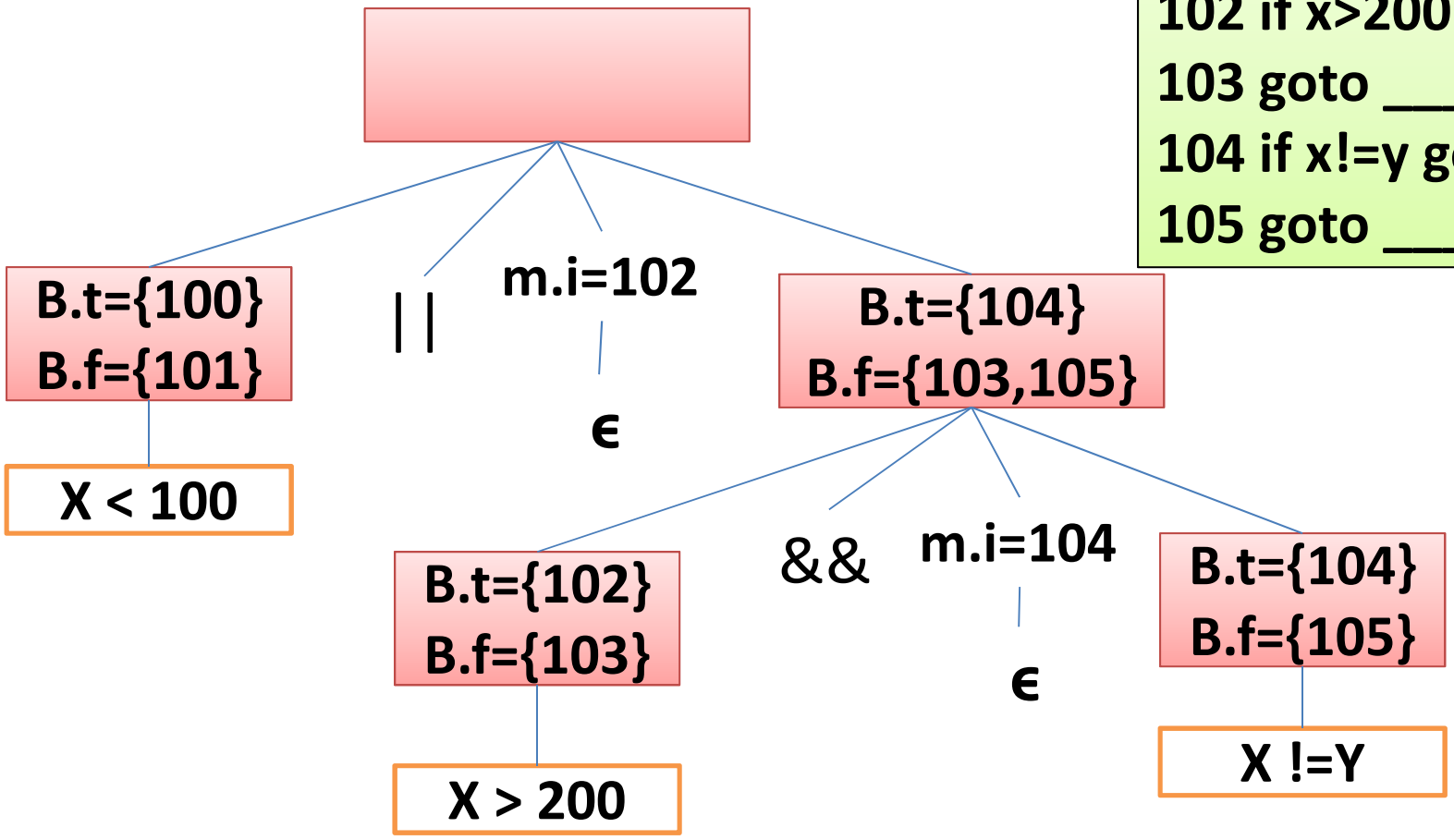
100:if x<100 goto __
101 goto ___
102 if x>200 goto ___
103 goto ___
104 if x!=y goto ___
105 goto ___



Backpatching for Boolean Expressions

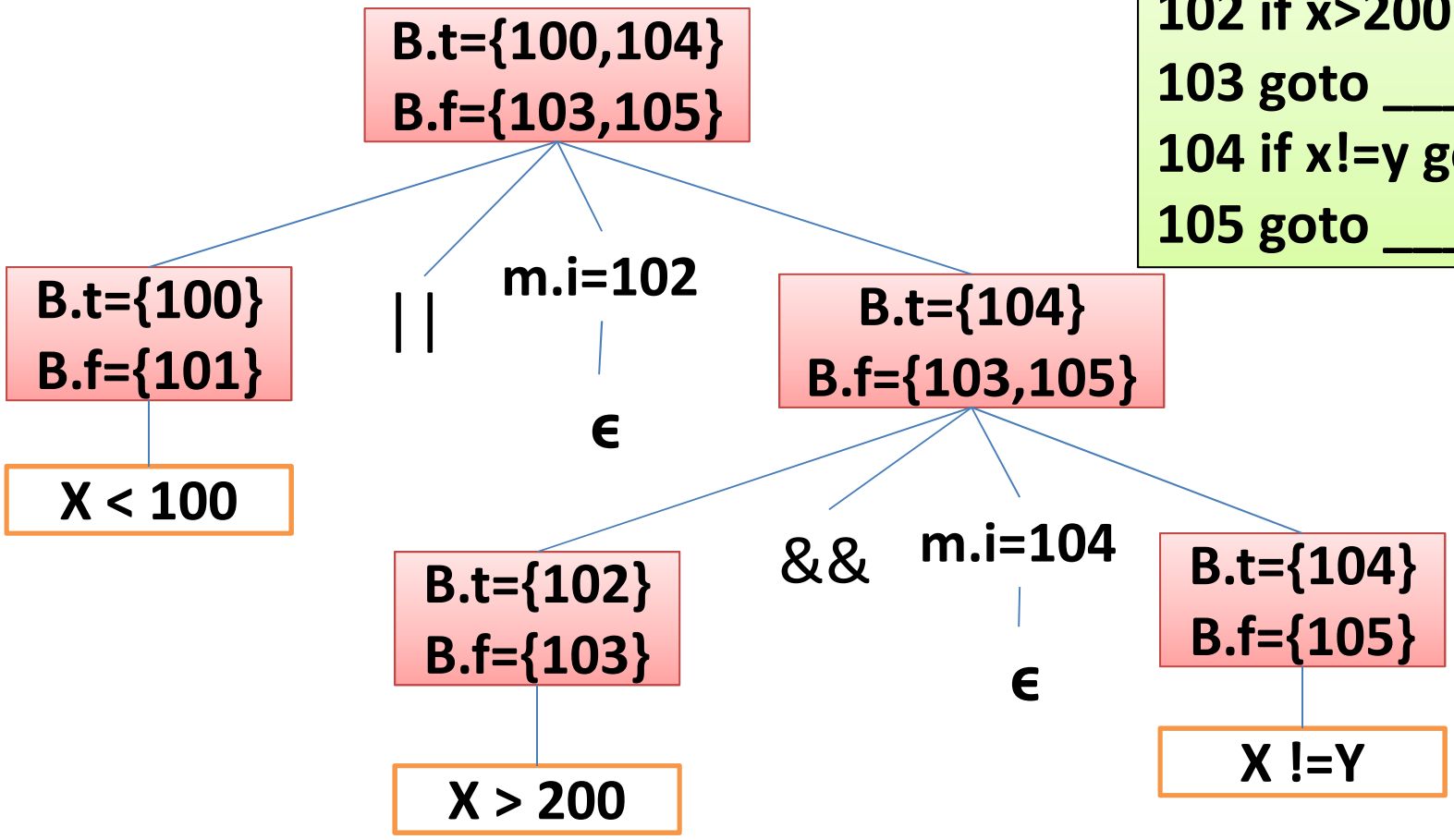
E->E1 and M E2 {backpatch(E1.truelist, M.qual);
E.truelist = E2.truelist;
E.falselist = merge(E1.truelist, E2.truelist);}

100:if x<100 goto __
101 goto ___
102 if x>200 goto ___
103 goto ___
104 if x!=y goto ___
105 goto ___



Backpatching for Boolean Expressions

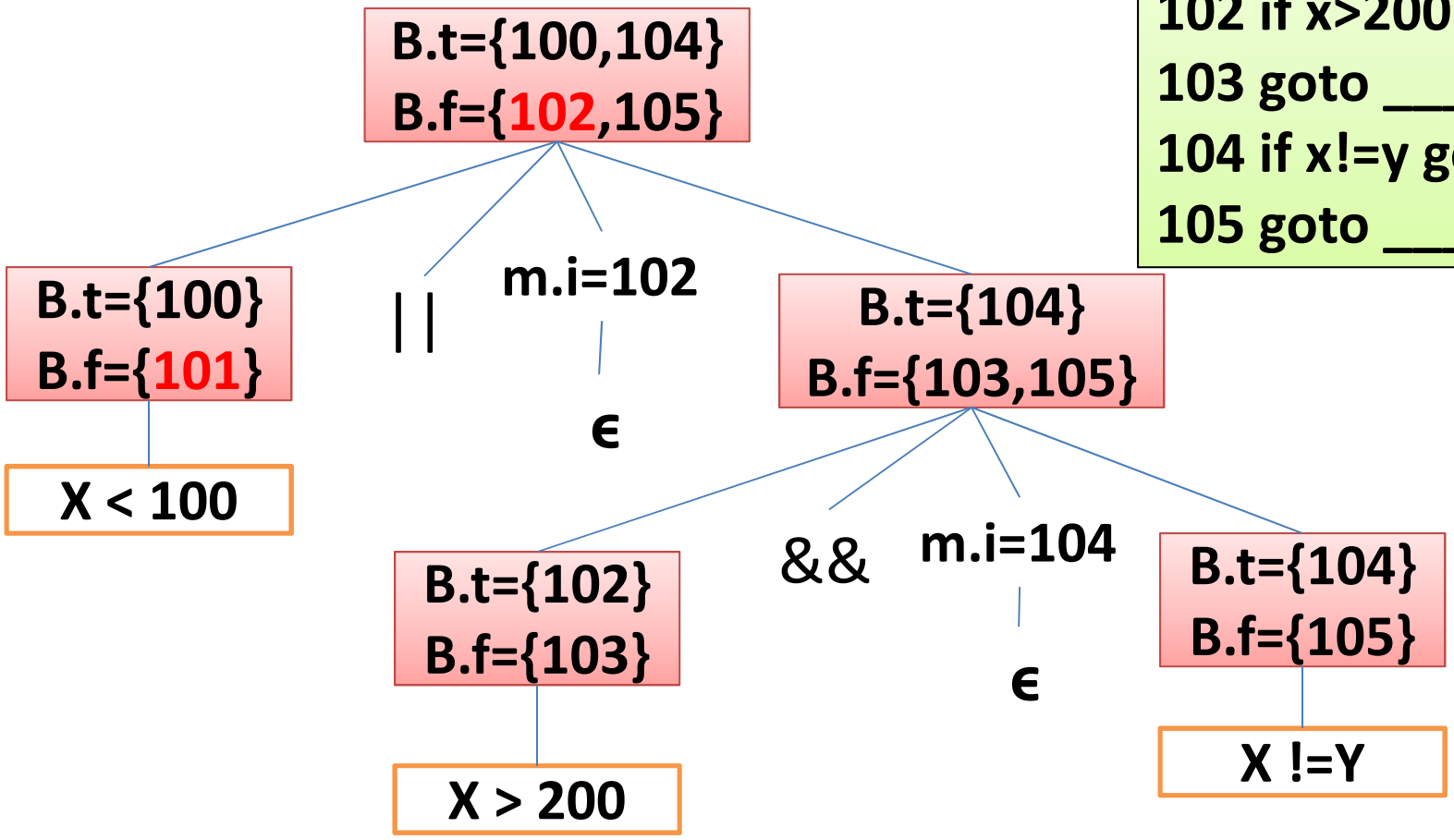
E->E1 or M E2 { backpatch(E1.falselist, M.qual);
E.truelist = merge(E1.truelist, E2.truelist);
E.falselist = E2.falselist;}



100:if x<100 goto __
101 goto ___
102 if x>200 goto ___
103 goto ___
104 if x!=y goto ___
105 goto ___

Backpatching for Boolean Expressions

E->E1 or M E2 { backpatch(E1.falselist, M.qual);
E.truelist = merge(E1.truelist, E2.truelist);
E.falselist = E2.falselist;}

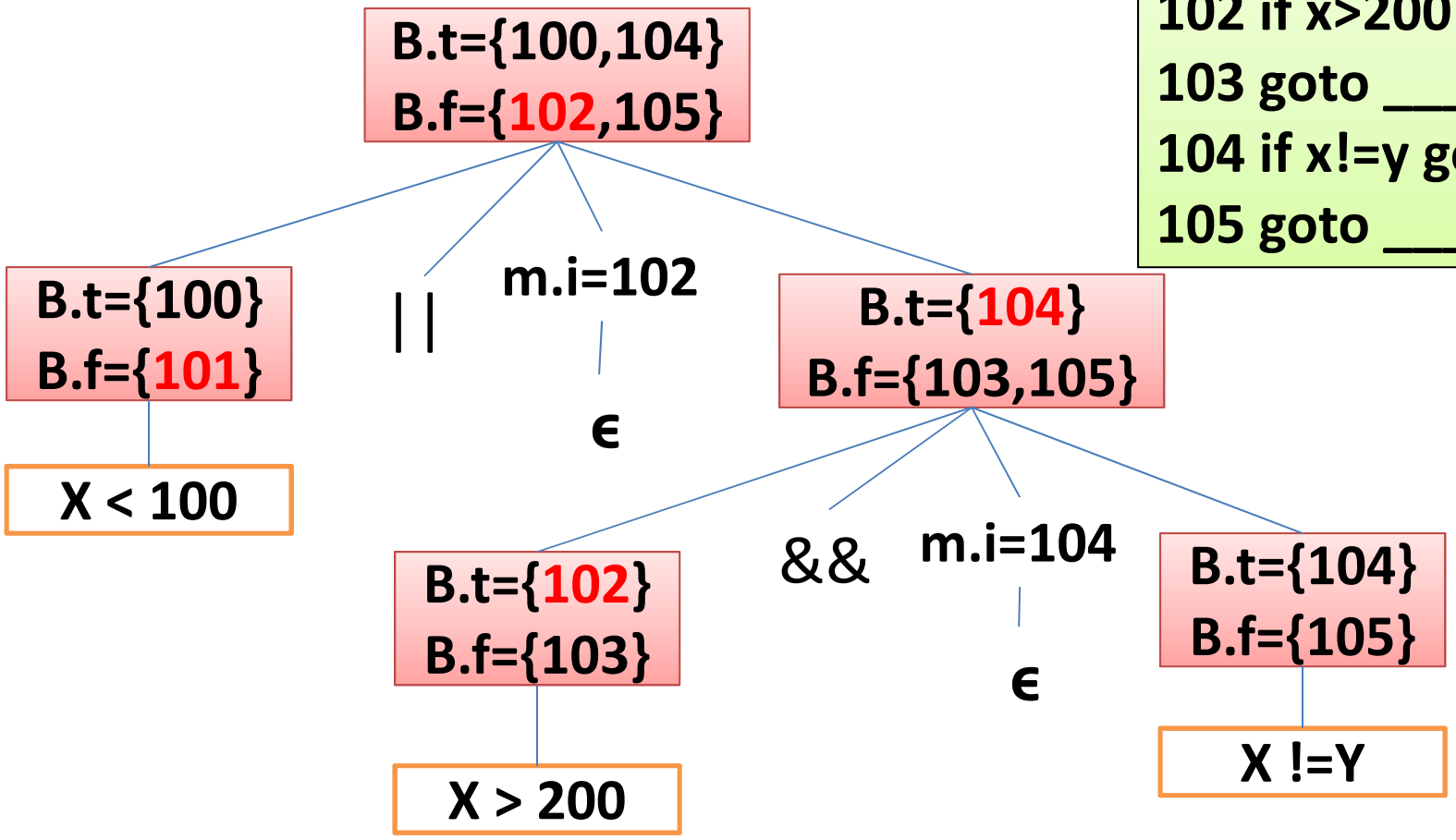


100:if x<100 goto __
101 goto 102
102 if x>200 goto ___
103 goto ___
104 if x!=y goto ___
105 goto ___

Backpatching for Boolean Expressions

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

100:if x<100 goto __
101 goto 102
102 if x>200 goto 104_
103 goto ___
104 if x!=y goto ___
105 goto ___



Flow of control statements

S -> if E then S1

S -> if E then S1 else S2

S -> while E do S1

S-> begin L end

S-> A

L->L; S

L->S

- Attributes: E.truelist, E.falselist
- S.nextlist: goto statements to the next instruction after this statement.

Flow of control statements

S -> if E then M S1 { backpatch(E.truelist, M.quad);

S.nextlist = merge(E.falselist, S1.nextlist);}

S-> if E then M1 S1 N else M2 S2 {

backpatch(E.truelist, M1.quad);

backpatch(E.falselist, M2.quad);

S.nextlist := merge(S1.nextlist, N.nextlist, S2.nextlist);}

S -> while M1 E do M2 S1 {

backpatch(E.truelist, M2.quad);

Backpatch(S1.nextlist, M1.quad);

S.nextlist = E.falselist;

Emit('goto ' M1.quad);}

Flow of control statements

S-> begin L end	{S.nextlist = L.nextlist}
S-> A	{S.nextlist = null;}
L->L1 M S	{backpatch(L1.nextlist, M.quad); L.nextlist = S.nextlist;}
L->S	{L.nextlist = S.nextlist;}
N-> ϵ	{N.nextlist = makelist(nextquad); emit('goto ???');}
M->ϵ	{M.quad = nextquad;}

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 :
```

Better 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
        if  t != V1 goto L1
        code for S1
        goto next
L1:     if  t != V2 goto L2
        code for S2
        goto next
L2:
Ln-1:   if t != Vn  goto Ln-1
        code for Sn-1
        goto next
Ln      : code for S,
next :
```

Function Calls

- **Caller:**
  - evaluate actual parameters, place them where the callee expects them:
    - param x, k // x is the k^{th} actual param of the call
  - save appropriate machine state (e.g., return address) and transfer control to the callee:
    - call p
- **Callee:**
  - allocate space for activation record, save callee-saved registers as needed, update stack/frame pointers:
    - enter p

Function Returns

- **Callee:**
  - restore callee-saved registers; place return value (if any) where caller can find it; update stack/frame pointers:
    - retval x;
    - leave p
  - transfer control back to caller:
    - return
- **Caller:**
  - save value returned by callee (if any) into x:
    - retrieve x

Function Call/Return: Example

Source:

```
x = f(0, y+1) + 1;
```

Intermediate Code: Caller:

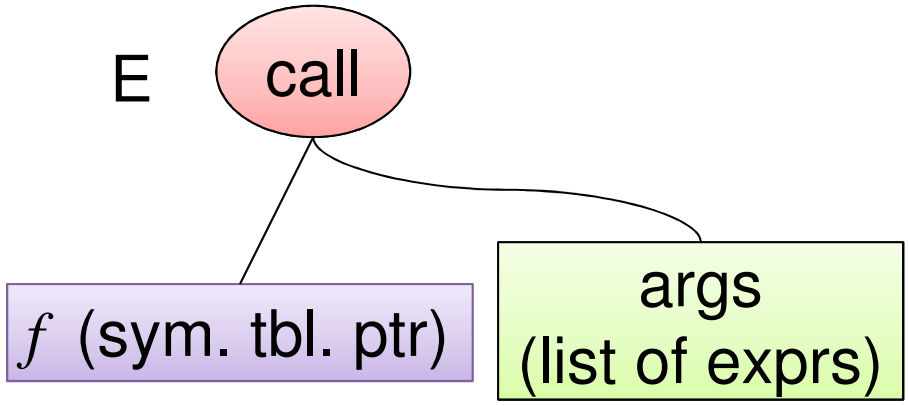
```
t1 = y+1
param t1, 2
param 0, 1
call f
retrieve t2
x = t2+1
```

Intermediate Code: Callee:

```
enter f    /* set up acti. record */
…          /* code for f's body */
retval t27  //return value of t27
leave f    //clean up acti.record
*/
return
```

Intermediate Code for Function Calls

non-void return type call:



Code Structure:

… evaluate actuals …

param x_k

…

param x_1

} R-to-L

call f

retrieve t0 /* t0 a temp var */

codeGen_expr(E):

/* E.nodetype = FUNCALL */

codeGen_expr_list(arguments);

E.place = newtemp(f.returnType);

E.code = …code to evaluate the arguments…

⊕ param x_k

…

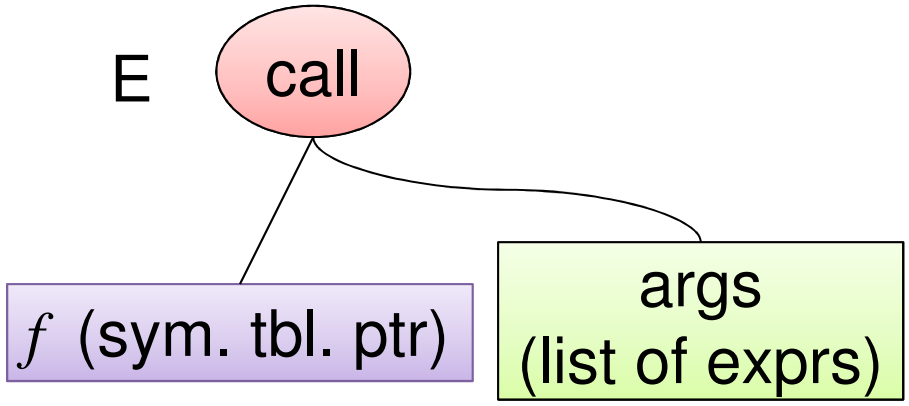
⊕ param x_1

⊕ call f, k

⊕ retrieve E.place;

Intermediate Code for Function Calls

void return type call:



Code Structure:

… evaluate actuals …

param x_k

…

param x_1

} R-to-L

call f

//retrieve t0 /* t0 a temp var */

codeGen_expr(E):

/* E.nodetype = FUNCALL */

codeGen_expr_list(arguments);

//E.place = newtemp(f.returnType);

E.code = …code to evaluate the arguments…

⊕ param x_k

…

⊕ param x_1

⊕ call f, k

// ⊕ retrieve E.place;

void return type ⇒ f has no return value ⇒ no need to allocate space for one, or to retrieve any return value.