Intermediate Code Generation

Part III

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**

Short-Circuit Code

- IF $B \rightarrow B_1 \,||\, B_2$ and B_1 is **true** then B is **true**
  - We can omit evaluation of B_2
- IF $B \rightarrow B_1 \,\&\&\, B_2$ 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

Might be translated into

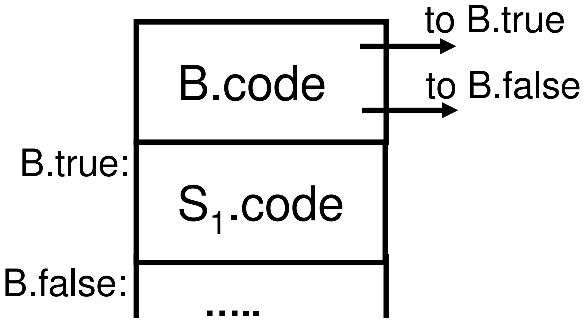
```
          if x < 100 goto $L_2$
          ifFalse x >200 goto $L_1$
          ifFalse x!= y goto $L_1$
   $L_2$ : x=0
   $L_1$ :
```

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

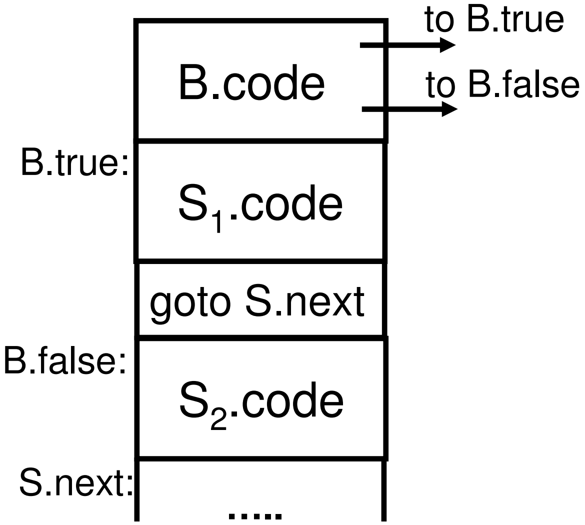
Flow-of-Control Statements

$$S \rightarrow \text{if } (B)\ S_1$$
$$S \rightarrow \text{if } (B)\ S_1 \text{ else } S_2$$
$$S \rightarrow \text{while } (B)\ S_1$$

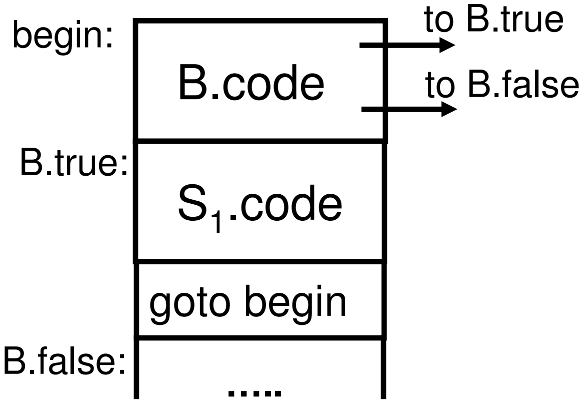
- B and S has synthesized attribute *code*
- Within B.code jumps are based on value of B



If (B) S_1



If (B) S_1 else S_2



while (B) S_1

Syntax directed definition for flow-of-control statements

$P \rightarrow S$	S.next = newLabel()
	P.code = S.code || 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 || label(B.true) || 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 || label(B.true) || S_1.code
	|| gen('goto' S.next) || label (B.false) || 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) || 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

Generating three-address code for booleans

<table>
<tr><td>B → B_1 || B_2</td><td>B_1.true = B.true
B_1.false = newLabel()
B_2.true=B.true
B_2.false=B.false
B.code= B_1.code || label (B_1.false) || B_2.code</td></tr>
<tr><td>B → B_1 &amp;&amp; B_2</td><td>B_1.true = newLabel()
B_1.false = B.false
B_2.true=B.true
B_2.false=B.false
B.code= B_1.code || label (B_1.true) || B_2.code</td></tr>
<tr><td>B → ! B_1</td><td>B_1.true = B.false
B_1.false = B.true
B.code= B_1.code</td></tr>
<tr><td>B → E_1 <b>rel</b> E_2</td><td>B.code= E_1.code || E_2.code
|| gen (‘if’ E_1.addr <b>rel</b>.op E_2.addr ‘goto’ B.true)
|| gen (‘goto’ B.false)</td></tr>
<tr><td>B → <b>true</b></td><td>B.code = gen (‘goto’ B.true)</td></tr>
<tr><td>B → <b>false</b></td><td>B.code = gen (‘goto’ B.false)</td></tr>
</table>

Example

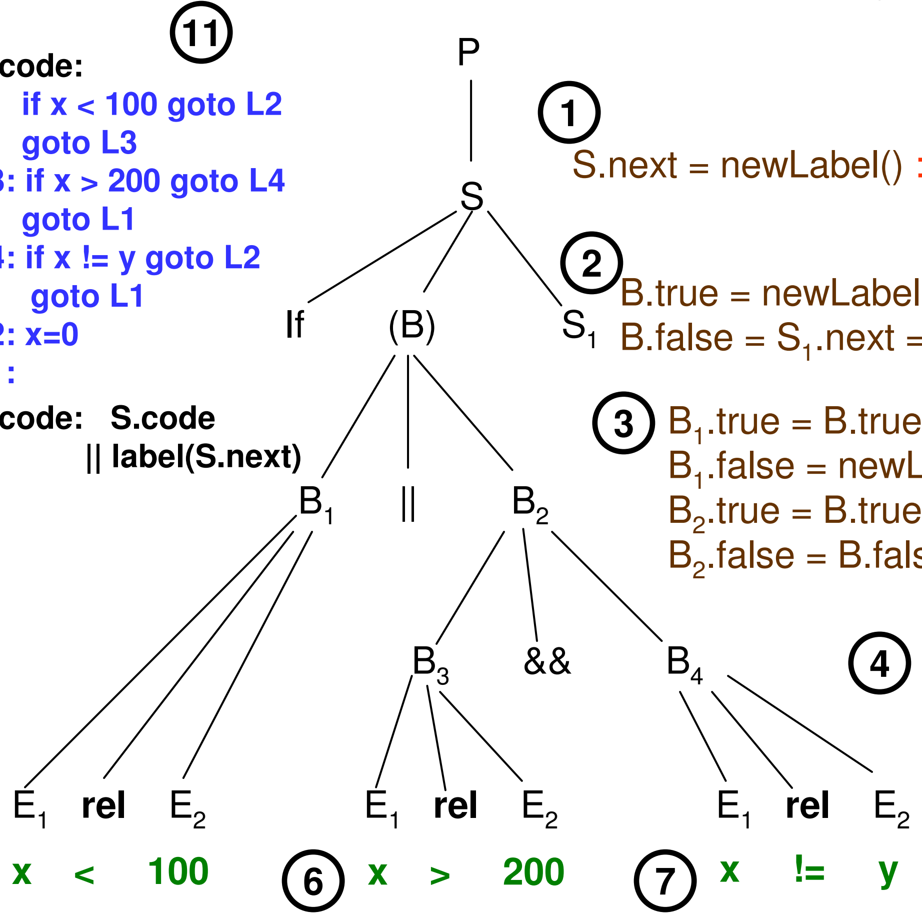
```
if (x < 100 || x > 200 && x!= y ) x=0;
```

is translated to

```
     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

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



1

S.next = newLabel() : L1

2

B.true = newLabel() : L2
B.false = S_1.next = S.next : L1

3

B_1.true = B.true : L2
B_1.false = newLabel() : L3
B_2.true = B.true : L2
B_2.false = B.false : L1

4

B_3.true = newLabel() : L4
B_3.false = B_2.false : L1
B_4.true = B_2.true : L2
B_4.false = B_2.false : L1

5

B_1.code:

```
if x < 100 goto L2
goto L3
```

6

```
If x > 200 goto L4
goto L1
```

7

B_4.code:

```
if x != y goto L2
goto L1
```

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)

8

B_2.code:

```
    if x > 200 goto L4
    goto L1
L4: if x != y goto L2
    goto L1
```

B.code: B_1.code
|| label(B_1.true)
|| B_2.code

9

B.code:

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

B.code: B_1.code
|| label(B_1.false)
|| B_2.code

10

S.code:

```
    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
```

S.code: B.code
|| label(B.true)
|| S_1.code

11

P.code:

```
    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:
```

P.code: S.code
|| label(S.next)

Backpatching

- Easiest way to implement the translations is to use two passes
- In one pass we may not know the target label for a jump statement
- *Backpatching* allows one pass code generation
- 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

    Synthesized attributes of nonterminal B
  - nextlist: list of jumps to the instruction immediately following the code for S
- These lists can be implemented as a synthesized attribute
- Assume instructions are generated into an instruction arrays

Functions for backpatching

- makelist(i): creates a new list containing only i, an index into the array of instructions. Returns a pointer to the newly created array

- merge(p_1, p_2): concatenates the lists pointed by p_1 and p_2 and returns a pointer to the concatenated list

- backpatch(p,i): inserts i as the target label for each instructions on the list pointed by p

Backpatching for Boolean Expression

$B \rightarrow B_1 \mathbin{\Vert} M\ B_2$	{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;}

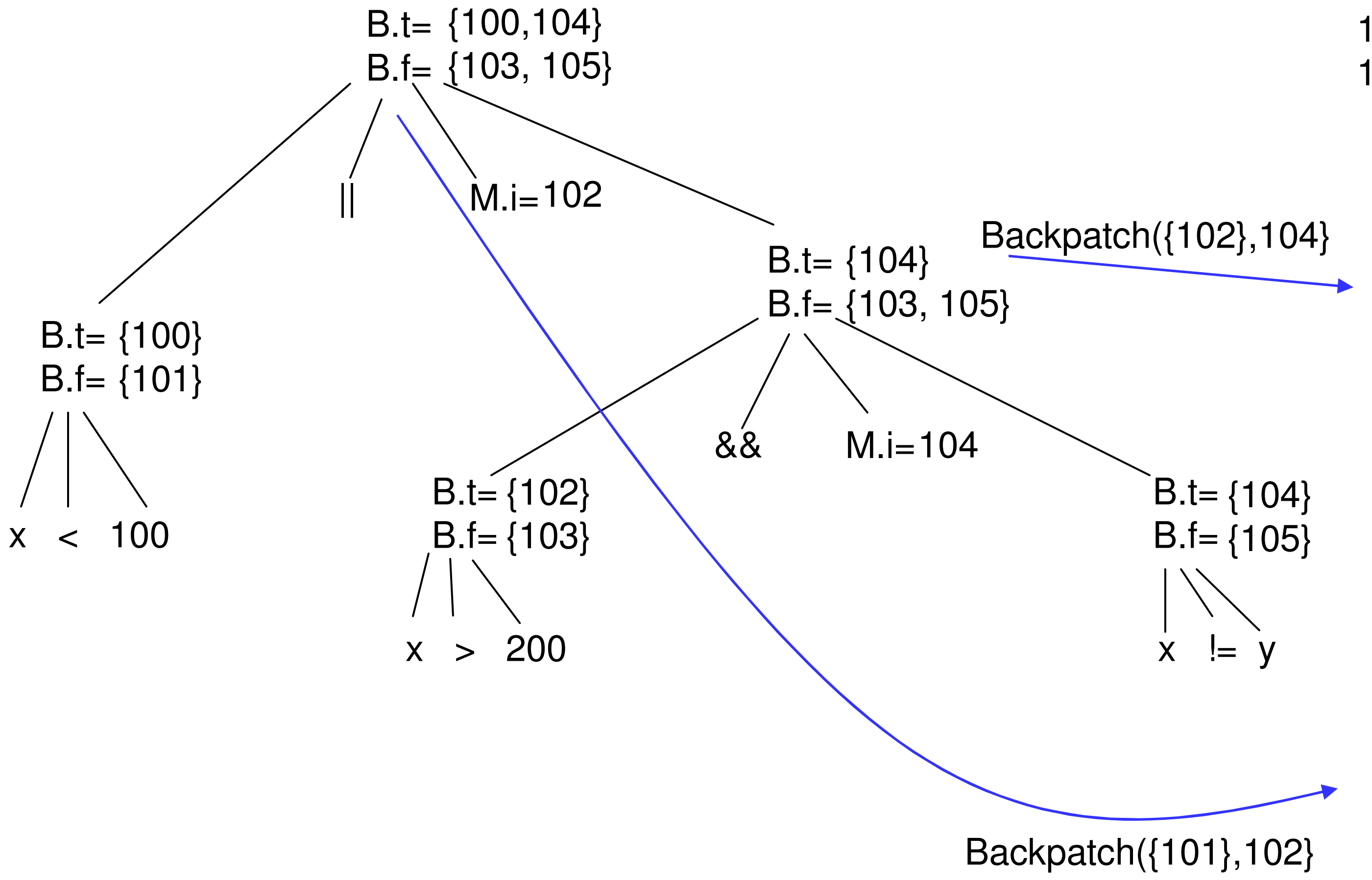
$B \rightarrow E_1$ **rel** E_2	{B.truelist = makelist(nextinstr);
	B.falselist = makelist(nextinstr+1);
	emit(‘if’ E_1.addr **rel**.op E_2.addr ‘goto _’)
	emit(‘goto _’)}

$B \rightarrow$ **true**	{B.truelist = makelist(nextinstr); emit (‘goto _’);}
$B \rightarrow$ **false**	{B.false = makelist(nextinstr); emit (‘goto _’);}
$M \rightarrow \varepsilon$	{ M.instr = nextinstr;}

Backpatching: Example

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
```



B.t= {100,104}
B.f= {103, 105}

||

M.i=102

B.t= {104}
B.f= {103, 105}

Backpatch({102},104)

B.t= {100}
B.f= {101}

x < 100

B.t= {102}
B.f= {103}

x > 200

&&

M.i=104

B.t= {104}
B.f= {105}

x != y

Backpatch({101},102)

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

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

Backpatching for flow of control statements

S → if (B) M S_1	{backpatch(B.truelist, M.instr);
	S.nextlist = merge(B.falselist,S_1.nextlist);}

S → 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 → 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 → { L }	{S.nextlist = L.nextlist;}
S → A;	{S.nextlist = **null;**}
M → ε	{M.instr=nextinstr;}
N → ε	{N.nextlist = makelist(nextinstr); emit ('goto _');}

L → L_1 M S	{backpatch(L1.nextlist, M.instr);
	L.nextlist = S.nextlist;}

L → S	{ L.nextlist = S.nextlist;}