Control Flow

Boolean Expressions

We will consider expressions defined by the following grammar:

$E \rightarrow E$ **or** E | E **and** E | **not** E | (E) | **id relop id** | **true** | **false**

Where **relop** is: `<`, `<=`, `=`, `!=`, `>`, or `>=`

Evaluation is typically left to right.

The following expression: **`a or b and not c`**

Translates to three address code as follows:

t_1 := not c
t_2 := b and t_1
t_3 := a or t_1

Three-address code for Booleans

Production	Semantic Rules
$E \rightarrow E_1$ **or** E_2	$E_1.true := E.true;$ $E_1.false := newlabel;$ $E_2.true := E.true;$ $E_2.false := E.false;$ $E.code := E_1.code \parallel gen(E_1.false\ ':') \parallel E_2.code$
$E \rightarrow E_1$ **and** E_2	$E_1.true := newlabel;$ $E_1.false := E.false;$ $E_2.true := E.true;$ $E_2.false := E.false;$ $E.code := E_1.code \parallel gen(E_1.true\ ':') \parallel E_2.code$
$E \rightarrow$ **not** E_1	$E_1.true := E.false;$ $E_1.false := E.true;$ $E.code := E_1.code$
$E \rightarrow (E_1)$	$E_1.true := E.true;$ $E_1.false := E.false;$ $E.code := E_1.code$
$E \rightarrow$ **id**$_1$ **relop** **id**$_2$	$E.code :=$ gen('if' **id**$_1$.place **relop**.op **id**$_2$.place 'goto' E.true) $\parallel$ gen('goto' E.false)
$E \rightarrow$ **true**	$E.code :=$ gen('goto' E.true)
$E \rightarrow$ **false**	$E.code :=$ gen('goto' E.false)

Fig. 8.24. Syntax-directed definition to produce three-address code for booleans.

Example

Consider: `a < b or c < d and e < f`

Then:

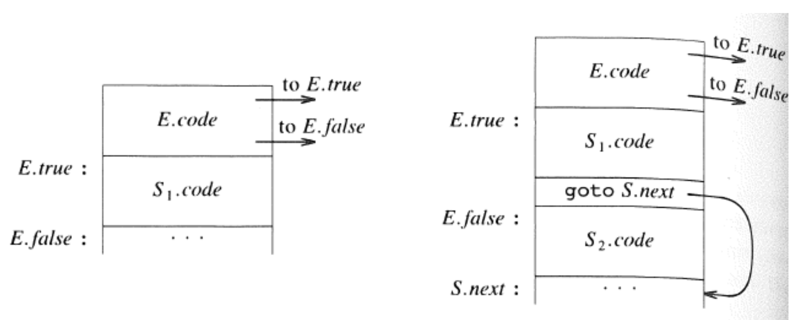
```
     if a < b goto Ltrue
     goto L1
L1:  if c < d goto L2
     goto Lfalse
L2:  if e < f goto Ltrue
     goto Lfalse
```

Flow-of-control Statements

Consider:

$$\begin{aligned} S \rightarrow\ & \textbf{if } E \textbf{ then } S_1 \\ \mid\ & \textbf{if } E \textbf{ then } S_1 \textbf{ else } S_2 \\ \mid\ & \textbf{while } E \textbf{ do } S_1 \end{aligned}$$

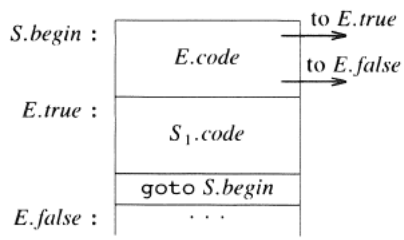
Code for If-Then/Else



(a) if-then

(b) if-then-else

Code for While



(c) while-do

Translation Scheme for Control Flow Expressions

PRODUCTION	SEMANTIC RULES
$S \rightarrow$ **if** E **then** S_1	$E.true$:= $newlabel$; $E.false$:= $S.next$; $S_1.next$:= $S.next$; $S.code$:= $E.code$ ‖ $gen(E.true$ ':') ‖ $S_1.code$
$S \rightarrow$ **if** E **then** S_1 **else** S_2	$E.true$:= $newlabel$; $E.false$:= $newlabel$; $S_1.next$:= $S.next$; $S_2.next$:= $S.next$; $S.code$:= $E.code$ ‖ $gen(E.true$ ':') ‖ $S_1.code$ ‖ gen('goto' $S.next$) ‖ $gen(E.false$ ':') ‖ $S_2.code$
$S \rightarrow$ **while** E **do** S_1	$S.begin$:= $newlabel$; $E.true$:= $newlabel$; $E.false$:= $S.next$; $S_1.next$:= $S.begin$; $S.code$:= $gen(S.begin$ ':') ‖ $E.code$ ‖ $gen(E.true$ ':') ‖ $S_1.code$ ‖ gen('goto' $S.begin$)

Fig. 8.23. Syntax-directed definition for flow-of-control statements.

Example

Consider:

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

Then:

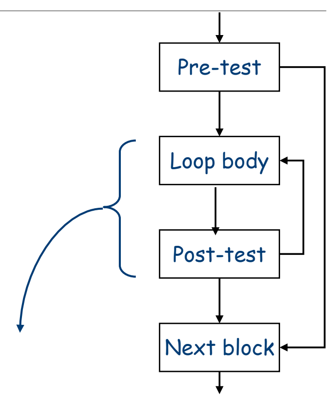
```
L1:  if a < b goto L2
     goto Lnext
L2:  if c < d goto L3
     goto L4
L3:  t1 := y + z
     x := t1
     goto L1
L4:  t2 := y - z
     x := t2
     goto L1
Lnext:
```

Control Flow

Loops

- Evaluate condition before loop (if needed)
- Evaluate condition after loop
- Branch back to the top (if needed)

Merges test with last block of loop body



while, for, do, and until all fit this basic model

Break statements

Many modern programming languages include a break

Exits from the innermost control-flow statement
- Out of the innermost loop
- Out of a case statement

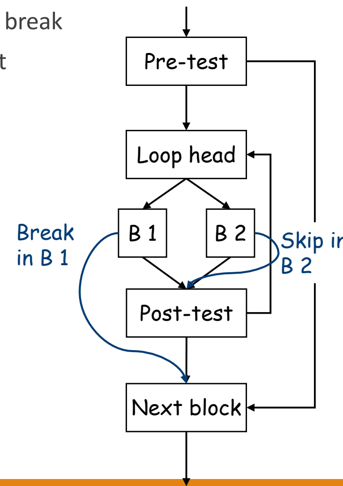
Translates into a jump

Targets statement outside control-flow construct

Creates multiple-exit construct

Skip in loop goes to next iteration

Only make sense if loop has > 1 block



Code Shape Case Statement

- Implement as cascaded if-then-else statements
- Implement it as a jump table
- Implement it as a binary search
- Compiler must choose best implementation strategy

Cascaded If-Then-Else Statements

```
switch (e1)  {
  case 0:   block0;
            break;
  case 1:   block1;
            break;
  case 3:   block3;
            break;
  default:  blockd;
            break;
}
```

```
t1 ← e1
if (t1 = 0)
   then block0
   else if (t1 = 1)
      then block1
      else if (t1 = 2)
         then block2
         else if (t1 = 3)
            then block3
            else blockd
```

- Cost depends on where your case actually occurs
- O(number of cases)

Jump Table

```
switch (e1)  {
  case 0:   block0
            break;
  case 1:   block1
            break;
  case 2:   block2
            break;
  ...
  case 9:   block9
            break;
  default:  blockd
            break;
}
```

Label
LB_0
LB_1
LB_2
LB_3
LB_4
LB_5
LB_6
LB_7
LB_8
LB_9

```
t1 ← e1
if (0 > t1 or t1 > 9)
   then jump to LBd
   else
      t2 ←@Table + t1 x 4
      t3 ← memory(t2)
      jump to t3
```

- Lookup address in a table and jump to it
- Uniform (constant) cost

Binary Search

```
switch (e1)  {
  case 0:  block0
           break;
  case 15: block15
           break;
  case 23: block23
           break;
  ...
  case 99: block99
           break;
  default: blockd
           break;
}
```

Value	Label
0	LB_0
15	LB_{15}
23	LB_{23}
37	LB_{37}
41	LB_{41}
50	LB_{50}
68	LB_{68}
72	LB_{72}
83	LB_{83}
99	LB_{99}

```
t1 ← e1
down ← 0   // lower bound
up ← 10    // upper bound + 1
while (down + 1 < up)  {
   middle ← (up + down) ÷ 2
   if (Value [middle] ≤ t1)
      then down ← middle
      else up ← middle
}
if (Value [down] = t1
   then jump to Label[down]
   else jump to LBd
```

- Need a dense set of conditions to search
- Uniform (log n) cost

Backpatching

Main problem with generating code for Boolean expressions and flow-of-control statements in a single pass is that we may not know all the labels.

Solution: generate branching statements with targets left unspecified.

Each such statement will be placed on a list whose labels will be filled in when it is known.

The subsequent filling of goto labels is called *backpatching*.

Translation Scheme for Boolean Expressions

$E \rightarrow E_1$ **or** E_2	{ *E.place* := *newtemp*; *emit*(*E.place* ':=' E_1.*place* 'or' E_2.*place*) }
$E \rightarrow E_1$ **and** E_2	{ *E.place* := *newtemp*; *emit*(*E.place* ':=' E_1.*place* 'and' E_2.*place*) }
$E \rightarrow$ **not** E_1	{ *E.place* := *newtemp*; *emit*(*E.place* ':=' 'not' E_1.*place*) }
$E \rightarrow (E_1)$	{ *E.place* := E_1.*place* }
$E \rightarrow$ **id**$_1$ **relop id**$_2$	{ *E.place* := *newtemp*; *emit*('if' **id**$_1$.*place* **relop**.*op* **id**$_2$.*place* 'goto' *nextstat* + 3); *emit*(*E.place* ':=' '0'); *emit*('**goto**' *nextstat* + 2); *emit*(*E.place* ':=' '1') }
$E \rightarrow$ **true**	{ *E.place* := *newtemp*; *emit*(*E.place* ':=' '1') }
$E \rightarrow$ **false**	{ *E.place* := *newtemp*; *emit*(*E.place* ':=' '0') }

Fig. 8.20. Translation scheme using a numerical representation for booleans.

Example

Consider: `a < b or c < d and e < f`

The code produced is:

```
100:  if a < b goto 103        107:  t2 := 1
101:  t1 := 0                  108:  if e < f goto 111
102:  goto 104                 109:  t3 := 0
103:  t1 := 1                  110:  goto 112
104:  if c < d goto 107        111:  t3 := 1
105:  t2 := 0                  112:  t4 := t2 and t3
106:  goto 108                 113:  t5 := t1 or t4
```

Fig. 8.21. Translation of `a<b or c<d and e<f`.

1. *makelist*(i) creates a new list containing only i, an index into the array of quadruples; *makelist* returns a pointer to the list it has made.
2. *merge*(p_1, p_2) concatenates the lists pointed to by p_1 and p_2, and returns a pointer to the concatenated list.
3. *backpatch*(p, i) inserts i as the target label for each of the statements on the list pointed to by p.

Modify grammar by inserting a non-terminal *M* in strategic places:

$$
\begin{array}{lll}
(1) & E \rightarrow & E_1 \textbf{ or } M\ E_2 \\
(2) & & \mid\ E_1 \textbf{ and } M\ E_2 \\
(3) & & \mid\ \textbf{not } E_1 \\
(4) & & \mid\ (\ E_1\ ) \\
(5) & & \mid\ \textbf{id}_1 \textbf{ relop } \textbf{id}_2 \\
(6) & & \mid\ \textbf{true} \\
(7) & & \mid\ \textbf{false} \\
(8) & M \rightarrow & \epsilon
\end{array}
$$

With production M -> ε, we associate the following semantic action:

$$\{ M.quad := nextquad \}$$

The variable "nextquad" holds the index of the next quadruple to follow.

Translation Scheme

(1)	$E \rightarrow E_1$ **or** $M\ E_2$	{ *backpatch* (E_1.*falselist*, *M.quad*); *E.truelist* := *merge*(E_1.*truelist*, E_2.*truelist*); *E.falselist* := E_2.*falselist* }
(2)	$E \rightarrow E_1$ **and** $M\ E_2$	{ *backpatch* (E_1.*truelist*, *M.quad*); *E.truelist* := E_2.*truelist*; *E.falselist* := *merge*(E_1.*falselist*, E_2.*falselist*) }
(3)	$E \rightarrow$ **not** E_1	{ *E.truelist* := E_1.*falselist*; *E.falselist* := E_1.*truelist* }
(4)	$E \rightarrow (\ E_1\ )$	{ *E.truelist* := E_1.*truelist*; *E.falselist* := E_1.*falselist* }
(5)	$E \rightarrow$ **id**$_1$ **relop id**$_2$	{ *E.truelist* := *makelist*(*nextquad*); *E.falselist* := *makelist*(*nextquad* + 1); *emit*('if' **id**$_1$.*place* **relop**.*op* **id**$_2$.*place* 'goto _'); *emit*('goto _') }
(6)	$E \rightarrow$ **true**	{ *E.truelist* := *makelist*(*nextquad*); *emit*('goto _') }
(7)	$E \rightarrow$ **false**	{ *E.falselist* := *makelist*(*nextquad*); *emit*('goto _') }
(8)	$M \rightarrow \epsilon$	{ *M.quad* := *nextquad* }