$E_1.place$ = newtemp;

$E_2.true$ = newlabel

$E_2.false$ = newlabel

$E.code$ = $E_1.code$ || $E_2.code$ || gen($E_2.true$ ':'

$E.place$ ':=' $E_1.place$ + 1) ||

gen ('goto' nextstat + 1) ||

gen ($E_2.false$ ':' $E.place$ ':=' $E_1.place$)

elseif ...

Case Statements:

```
Switch E
  begin
    case V1 : S1
    case V2 : S2
      ...
    case Vn-1 : Sn-1
    default : Sn
  end
```

code to evaluate E into t
if t ≠ V_1 goto L_1
code for S_1
goto next
L_1: if t ≠ V_2 goto L_2
code for S_2
goto next
L_2:
...
L_{n-2}: if t ≠ V_{n-1} goto L_{n-1}
code for S_{n-1}
goto next
L_{n-1} : code for S_n
next :

code to eval E into t
goto test
L_1: code for S_1
goto next
L_2: code for S_2
goto next
...
L_{n-1}: code for S_{n-1}
goto next
L_n: code for S_n
goto next
test: if t = V_1 goto L_1
if t = V_2 goto L_2
...
if t = V_{n-1} goto L_{n-1}
goto L_n
next:

Back Patching

(a||b) && (c < d) && (x > 5)

- while generating 'goto' statements in TAC, we don't know the label to be used for forward jumps
- leave the label unspecified and fill it later when we know what it will be.
- printing code into a file → filling in labels become difficult

→ Store TAC in an array and buffer the code, printing it when you reach points where the code can be emitted.

→ We maintain a list of TAC statements that need to be completed with the same label.

(i) makelist(i) → index into the array of quadruples

(ii) merge(L_1, L_2)

(iii) backpatch(L, label) – inserts label as the target for statements in L

(iv) nextquad – index of the next quadruple to be generated.

$E \rightarrow E_1$ or $M\ E_2$

backpatch (E_1.falselist , M.quad)

E.truelist = merge (E_1.truelist, E_2.truelist)

E.falselist = E_2.falselist

$E \rightarrow E_1$ and $M\ E_2$

backpatch (E_1.truelist , M.quad)

E.truelist = E_2.truelist

E.falselist = merge (E_1.falselist , E_2.falselist)

$E \rightarrow$ not E_1

E.truelist = E_1.falselist

E.falselist = E_1.truelist

$E \rightarrow (E_1)$

E.truelist = E_1.truelist

E.falselist = E_1.falselist

$E \rightarrow id_1$ relop id_2

E.truelist = makelist (nextquad)

E.falselist = makelist (nextquad + 1)

gen ('if' id_1 relop id_2 'goto _')

gen ('goto _')

E → true

E.truelist = makelist (nextquad)

gen ('goto _')

E → false

E.falselist = makelist (nextquad)

gen ('goto _')

M → ∈

M.quad = nextquad

Semantic rule for flow-of-control statements using backpatching

$S \rightarrow$ if E then M_1 S_1 N else M_2 S_2

- backpatch (E.truelist , M_1.quad)
  backpatch (E.falselist , M_2.quad)
  S.nextlist = merge (S_1.nextlist ,
  merge (N.nextlist , S_2.nextlist))

$S \rightarrow$ if E then M S_1

- backpatch (E.truelist , M.quad)
  S.nextlist = merge (E.falselist , S_1.nextlist)

$S \rightarrow$ while M_1 E do M_2 E

- backpatch (S_1.nextlist , M_1.quad)
  backpatch (E.truelist , M_2.quad)
  S.nextlist = E.falselist
  gen ('goto _' M_1.quad)

$N \rightarrow \epsilon$

- N.nextlist = makelist (nextquad)
  gen ('goto _')

$M \rightarrow \epsilon$ M.quad = nextquad

Nextlist → list of quadruples generated by statements that must be backpatched

Case statements :

- Sort values into a binary tree
- Hash table

Procedure calls :

S → call id (E list)
{ for each item p on queue do
emit ('param' p);
emit ('call' id.place); }

E list → E list, E {append E.place to the end of queue}

E list → E {initialize queue to contain only E.place}

Calling Sequences:

procedure call:

* Space alloc for activation rec. of the callee
* arg. eval of the called procedure, Env pointers to accen data in enclosing blo
* Save state of caller, ret. addren
* Jump to Callee

return from procedure:

* Store result in a known place
* restore activation rec. of the caller
* Jump to caller's ret. addren.