Last Time: Control Model Templates

• Last time we showed how to generate code for conditions (Boolean expressions) using the control model, resulting in a condition descriptor \((c, t, f)\) for the result.

• Recall that the general not operation

\[
c_1 \text{ not}
\]

has the simple template:

\[
c_1 \quad \text{code for } c_1 \\
\text{not} \quad (c_1, t_1, f_1) \\
\text{not} \quad (-c_1, f_1, t_1)
\]

• Thus, the not operator requires no code to be emitted.

• We now show how to use these results in generating code for control flow in statements such as if and while
If Statements

• While the Operand Stack and condition descriptors are sufficient to keep track of the labels associated with branch conditions, we need another mechanism to keep track of the control flow associated with statements.

• For this we use the **Fix Stack**, which is a stack of forward and backward targets of statement branches in the code.

• Elements of the Fix Stack are either **forward branches**, which means we have already generated a branch and we need to remember the label to emit when we figure out where it is to go (e.g., branch to **else** part of an **if** statement).

• Or they are **backward branches**, which means we have already generated a label and we need to remember it to emit the branch back to it when we figure out where that is (e.g., branch back to the beginning of a **loop**).

• The **Fix Stack** serves both purposes.
If Statements

• The Fix Stack keeps track of labels that we don't know the target of yet (forward fixes) and labels that we will need to be targets of branches we will emit later (backward fixes)

• Example (forward fix):

<table>
<thead>
<tr>
<th>Emitted Code</th>
<th>Operand Stack</th>
<th>Fix Stack</th>
</tr>
</thead>
<tbody>
<tr>
<td>if c then</td>
<td>(c, t, f)</td>
<td></td>
</tr>
<tr>
<td>code for c</td>
<td></td>
<td></td>
</tr>
<tr>
<td>b - c</td>
<td></td>
<td></td>
</tr>
<tr>
<td>f</td>
<td></td>
<td></td>
</tr>
<tr>
<td>t:</td>
<td>(ε, f, ε)</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>S1</th>
<th>code for S1</th>
<th>forward Lₙ</th>
</tr>
</thead>
<tbody>
<tr>
<td>br Lₙ</td>
<td>(ε, f, ε)</td>
<td></td>
</tr>
<tr>
<td>else</td>
<td>f:</td>
<td>forward Lₙ</td>
</tr>
<tr>
<td>S₂</td>
<td>code for S₂</td>
<td>forward Lₙ</td>
</tr>
<tr>
<td>end if</td>
<td>Ln:</td>
<td></td>
</tr>
</tbody>
</table>
Constant Conditions

- Constant conditions, those whose result is known at compile time, can be *folded* much like constant expressions.

- These are modelled using *always* and *never* branch conditions.

- If the condition is known at compile time, then statements masked by the condition need not be generated - for example, if the *then* part of an *if* statement can't be reached because we know at compile time that the condition of the *if* is always false, then we just don't generate any code for it.

- This is a special case of the constant folding local optimization called *statement folding* - Java for example guarantees that the compiler implements statement folding.

- Sometimes the individual portions of the condition may not be constant, but the expression as a whole may be constant.

\[
a < 5 \text{ and } a > 7
\]
Loop Templates

- The Turing-style general **loop / exit** is handled like **if**, except that:
  - the fix label is emitted before the loop (the branch is backward)
  - the branch condition is not inverted

<table>
<thead>
<tr>
<th>Emitted Code</th>
<th>Operand Stack</th>
<th>Fix Stack</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>loop</strong></td>
<td>$L_n:$</td>
<td>backward $L_n$</td>
</tr>
<tr>
<td>$S_1$</td>
<td>code for $S_1$</td>
<td></td>
</tr>
<tr>
<td>exit when $c$</td>
<td>code for $c$</td>
<td>$(c, t, f)$</td>
</tr>
<tr>
<td></td>
<td>$bc$ $L_m$</td>
<td></td>
</tr>
<tr>
<td>f:</td>
<td></td>
<td>$(\varepsilon, t+L_m, \varepsilon)$</td>
</tr>
<tr>
<td></td>
<td></td>
<td>backward $L_n$</td>
</tr>
<tr>
<td>$S_2$</td>
<td>code for $S_2$</td>
<td></td>
</tr>
<tr>
<td>end loop</td>
<td>$br L_n$</td>
<td>$(\varepsilon, t+L_m, \varepsilon)$</td>
</tr>
<tr>
<td></td>
<td>$t+L_m:$</td>
<td></td>
</tr>
</tbody>
</table>
Loop Templates

- All other kinds of loops are simply special cases of the general one, so we don't need any other loop templates

```plaintext
while c
    S
end while

repeat
    S
until c

for i := 1 to 10
    S
end for

loop
    exit when not c
    S
end loop

loop
    exit when c
    S
end loop

var i := 1
loop
    exit when i > 10
    S
    i := i + 1
end loop
```
Case Statements (Pascal, Turing, Modula, ... )

- Case statements are handled in two different ways, depending on the value of the labels - if the values are sparse, that is, there are a wide range of different values, then the case statement is treated as a large set of nested if then else statements

```
case x of
  label 35:
    if x = 35 then
      S1
  label 65536:
    elsif x = 65536 then
      S2
  label 1122341341:
    elsif x = 1122341341 then
      S3
  otherwise:
    else
      S4
end case

end if
```

- More often the values are close together, and the case statement can be implemented as a jump table (e.g. all cases in PT)
Case Statements (cont'd)

case e of

val(e) → reg

cmp reg, 3
blt L3
cmp reg, 7
bgt L3
jmp @reg .caseTable-3*4

label 3:
S1

label 5,6,7:
S2

otherwise:
S3

end case

caseTable:
L1,L3,L2,L2,L2
Lend:
Switch Statements (Java, C, C++)

```java
switch (e) {
    case 3:
        S1
        break;
    case 5,6,7:
        S2
        break;
    default:
        S3
}

val(e) → reg

cmp reg, 3
blt L3
cmp reg, 7
bgt L3
jmp @reg .caseTable-3*4

L1: code for S1
    br Lend
L2: code for S2
    br Lend
L3: code for S3
    br Lend

caseTable: L1,L3,L2,L2,L2
Lend:
```
Call Statements

• Unlike **if**, **loop** and **case** statements, the target address of a call statement is often unknown even after the entire file is compiled, because the called procedure, method or function may be in another compilation or library.

• In this case we rely on a separate pass called the **linker** to solve the problem, either at link or load time (**static linking**) or when the procedure is actually called at run time (**dynamic linking**).

• In both cases, the problem is solved in essentially the same way, using an **external symbol dictionary** (ESD) in the compiled object code file.
Call Statement Linking

• The **ESD** is a table that the code generator (or assembler) writes out preceding the actual code in the object code file.

• There are two kinds of entries in the table: *definitions* and *references*.

• These correspond to procedures defined in this compilation that others may need to link to, and procedures referenced in this compilation that it needs to link to.

```assembly
prog.o

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>‘P’ 030 def</td>
<td></td>
</tr>
<tr>
<td></td>
<td>‘Q’ 082 ref</td>
<td></td>
</tr>
<tr>
<td></td>
<td>‘R’ 180 def</td>
<td></td>
</tr>
</tbody>
</table>

ESD

prog.s

```

... .globl P
P: ...

... .globl Q
call Q

... .globl R
R: ...
...
Static Linking

- Either at *link time* (when executable file is made) or at *load time* (when execution begins), code segments are *relocated* and ESD's are *merged* to resolve references.

<table>
<thead>
<tr>
<th>prog.o</th>
<th>sub.o</th>
<th>main.o</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>'P' 030 def</td>
<td>'Q' 000 def</td>
<td>'main' 000 def</td>
</tr>
<tr>
<td>'Q' 082 ref</td>
<td>'R' 064 ref</td>
<td>'P' 020 ref</td>
</tr>
<tr>
<td>'R' 180 def</td>
<td></td>
<td></td>
</tr>
<tr>
<td>000</td>
<td>000</td>
<td>000</td>
</tr>
<tr>
<td>P:</td>
<td>Q:</td>
<td>main:</td>
</tr>
<tr>
<td>.globl Q</td>
<td>.globl R</td>
<td>.globl P</td>
</tr>
<tr>
<td>call 000</td>
<td>call 000</td>
<td>call 000</td>
</tr>
<tr>
<td>030</td>
<td>064</td>
<td>020</td>
</tr>
<tr>
<td>082</td>
<td></td>
<td></td>
</tr>
<tr>
<td>180</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Static Linking

<table>
<thead>
<tr>
<th></th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>‘P’</td>
<td>030 def</td>
</tr>
<tr>
<td>‘Q’</td>
<td>082 ref</td>
</tr>
<tr>
<td>‘R’</td>
<td>180 def</td>
</tr>
<tr>
<td>‘Q’</td>
<td>200 def</td>
</tr>
<tr>
<td>‘R’</td>
<td>264 ref</td>
</tr>
<tr>
<td>‘main’</td>
<td>320 def</td>
</tr>
<tr>
<td>‘P’</td>
<td>340 ref</td>
</tr>
</tbody>
</table>

- prog.o
- sub.o
- main.o

```
000
P:
.globl Q
call 200

082
Q:
.globl R
call 180

180
R:

200
Q:
.globl P
call 030

264
main:

320
main: 
.globl P
call 030

340
```

a.out

CISC 458 Winter 2015 © 2015 J.R. Cordy Lecture 30
Dynamic Linking

• Dynamic linking is similar, except that relocation and merged ESD are done at run time, loading object files as they are needed, and allowing for different object files to be linked in depending on environment (Java)
Summary

• Once expression and condition code templates have been designed, statement templates simply use their results to generate control code as conditional and unconditional branches

• Forward and backward branches are managed by a Fix Stack, which keeps track of both labels we have generated forward branches to that still need to be emitted, and labels we have emitted that we have yet to generate backward branches to

• Case and switch statements use a special template that treats the alternatives as an array of labels we need to subscript to find the appropriate alternative for a given case tag value

• Call statements are handled by generating an external symbol dictionary of calls to routines to be linked in. A special linker phase resolves these class between compilation units

• Next:
  Direct execution of virtual machine code using a micro-programmed VM interpreter