If statements

Processing control structures

```plaintext
if <bool_expr_1> then
    <stmt_list_1>
elseif <bool_expr_2> then
    <stmt_list_2>
... else
    <stmt_list_3>
endif
```

Notes on code generation

- The <op> in j<op> is dependent on the type of comparison you are doing in <bool_expr>
- When you generate JUMP instructions, you should also generate the appropriate LABELs
- But you may not put the LABEL into the code immediately
  - e.g., the OUT label (when should you create this? When should you put this in code?)
- Instead, pass LABEL around to routine which does need to generate it
- Remember: labels have to be unique!

Generating code for ifs

```
if <bool_expr_1> then
    <stmt_list_1>
elseif <bool_expr_2> then
    <stmt_list_2>
else
    <stmt_list_3>
endif
```

Directly generating binary code

- Recall difference between assembly code and machine code
  - Assembly code must be processed by assembler, machine code directly executable
  - One job of assembler: decide actual addresses to jump to instead of labels
- So what happens if we generate binary directly?
  - Need to insert JMP instructions before knowing where the label will be
- Solution: backpatching
  - Store offset of JMP instruction in semantic record
  - When label is created, access JMP instruction and “patch up” jump target
While loops

while <bool_expr> do
  <stmt_list>
end

Generating code for while loops

while <bool_expr> do
  <stmt_list>
end

LOOP:
  <bool_expr> j<!op> OUT
  <stmt_list>
  jmp LOOP
OUT:

For loops

for (<init_stmt>; <bool_expr>; <incr_stmt>)
  <stmt_list>
end

Generating code: for loops

for (<init_stmt>; <bool_expr>; <incr_stmt>)
  <stmt_list>
end

<init_stmt>
LOOP:
  <bool_expr> j<!op> OUT
  <stmt_list>
  INCR: <incr_stmt> jmp LOOP
OUT:

continue and break statements

for (<init_stmt>; <bool_expr>; <incr_stmt>)
  <stmt_list>
end

<init_stmt>
LOOP:
  <bool_expr> j<!op> OUT
  <stmt_list>
  INCR: <incr_stmt> jmp LOOP
OUT:

• Continue statements: skip past rest of block, perform incr_stmt and restart loop
• Break statements: jump out of loop (do not execute incr_stmt)
• Caveats:
  • Code for stmt_list is generated earlier—where do we jump?
  • Keep track of “loop depth” as you descend through AST

continue and break statements

• Continue statements: skip past rest of block, perform incr_stmt and restart loop
• Break statements: jump out of loop (do not execute incr_stmt)
• Caveats:
  • Code for stmt_list is generated earlier—where do we jump?
  • Keep track of “loop depth” as you descend through AST
Switch/case statements

```
switch (<expr>)
    case <const_list>: <stmt_list>
    case <const_list>: <stmt_list>
    ...
    default: <stmt_list>
end
```

Deciding where to jump

- Problem: do not know which label to jump to until switch expression is evaluated
- Use a jump table: an array indexed by case values, contains address to jump to
  - If table is not full (i.e., some possible values are skipped), can point to a default clause
  - If default clause does not exist, this can point to error code
- Problems
  - If table is sparse, wastes a lot of space
  - If many choices, table will be very large

Jump table example

Consider the code:

```
Case x is
(0010) When 0: stmts0
(0017) When 1: stmts1
(0192) When 2: stmts2
(0198) When 3 stmts3
(1000) When 5 stmts;
(1050) Else stmts;
```

Jump table has 6 entries:

```
<table>
<thead>
<tr>
<th>Case x</th>
<th>Jump</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>JUMP 0010</td>
</tr>
<tr>
<td>1</td>
<td>JUMP 0017</td>
</tr>
<tr>
<td>2</td>
<td>JUMP 0192</td>
</tr>
<tr>
<td>3</td>
<td>JUMP 0198</td>
</tr>
<tr>
<td>4</td>
<td>JUMP 1050</td>
</tr>
<tr>
<td>986</td>
<td>JUMP 1050</td>
</tr>
<tr>
<td>987</td>
<td>JUMP 1000</td>
</tr>
</tbody>
</table>
```

Table only has 983 unnecessary rows.

Perform a binary search on the table. If the entry is found, then jump to that offset. If the entry isn’t found, jump to others clause. $O(\log n)$ time, $n$ is the size of the table, for each jump.
Linear search example

Consider the code:

Is offset of local

Code start from the
Jump instruction

Case x is

(0010) When 0: stmts
(0017) When 1: stmts
(0192) When 2: stmts
(1050) When others stmts;

If there are a small number of
choices, then do an in-line linear
search. A straightforward way to do
this is generate code analogous to an
IF THEN ELSE.

If (x == 0) then stmts1;
Elseif (x == 1) then stmts2;
Elseif (x == 2) then stmts3;
Else stmts4;

O(n) time, n is the size of the table, for each jump.

Dealing with jump tables

switch (<expr>)

case <const_list>: <stmt_list>
case <const_list>: <stmt_list>
...
default: <stmt_list>
end

<expr>
<code for jump table>
LABEL0:
LABEL1:
...
DEFAULT:
OUT:

• Generate labels, code, then build jump table
• Put jump table after generated code
• Why do we need the OUT label?
• In case of break statements