If statements

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

Generating code for ifs

```plaintext
<code for bool_expr_1> \text{j<op> ELSE_1} \\
<code for stmt_list_1> \text{jmp OUT} \\
ELSE_1: \text{<code for bool_expr_2>} \text{j<op> ELSE} \\
<code for stmt_list_2> \text{jmp OUT} \\
ELSE: \text{<code for stmt_list_3> OUT}:
```

Notes on code generation

- The \text{j<op>} in \text{j<op>} is dependent on the type of comparison you are doing in \text{<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!

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
Processing Loops

While loops

while <bool_expr> do
  <stmt_list>
end

stmt_list
bool_expr
while_stmt
cond
block

Generating code for while loops

while <bool_expr> do
  <stmt_list>
end
LOOP:
  <bool_expr>
  j<!op> OUT
  <stmt_list>
  jmp LOOP
OUT:

• Note that the jump op is the negation of the expression op
• “Jump if boolean expression is false”
• Unconditional jump at end of loop
• Re-evaluate expression each time
• Question: what would code for “do-while” loop look like?

For loops

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

for_stmt
bool_expr
incr_expr
stmt_list
init_stmt
cond
next_stmt
body

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:

• Execute init_stmt first
• Jump out of loop if bool_expr is false
• Execute incr_stmt after block, jump back to top of loop
• Question: Why do we have the INCR label?

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
Switch/case statements

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

• Generated code should evaluate <expr> and make sure that some case matches the result.
• Question: how to decide where to jump?

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: ([xxxx] is address of code)

<table>
<thead>
<tr>
<th>Case x is</th>
<th>stmts</th>
</tr>
</thead>
<tbody>
<tr>
<td>(0010)</td>
<td>When 0:</td>
</tr>
<tr>
<td>(0017)</td>
<td>When 1:</td>
</tr>
<tr>
<td>(0192)</td>
<td>When 2:</td>
</tr>
<tr>
<td>(0198)</td>
<td>When 3:</td>
</tr>
<tr>
<td>(1000)</td>
<td>When 5:</td>
</tr>
<tr>
<td>(1050)</td>
<td>Else stmts</td>
</tr>
</tbody>
</table>

Jump table has 6 entries:

| 0 | JUMP 0010 |
| 1 | JUMP 0017 |
| 2 | JUMP 0192 |
| 3 | JUMP 0198 |
| 4 | JUMP 1050 |
| 5 | JUMP 1000 |

Table only has 983 unnecessary rows. Doesn’t appear to be the right thing to do! NOTE: table size is proportional to range of choice clauses, not number of clauses!

Do a binary search

Consider the code: ([xxxx] is address of code)

<table>
<thead>
<tr>
<th>Case x is</th>
<th>stmts</th>
</tr>
</thead>
<tbody>
<tr>
<td>(0010)</td>
<td>When 0:</td>
</tr>
<tr>
<td>(0017)</td>
<td>When 1:</td>
</tr>
<tr>
<td>(0192)</td>
<td>When 2:</td>
</tr>
<tr>
<td>(0198)</td>
<td>When 3:</td>
</tr>
<tr>
<td>(1000)</td>
<td>When 5:</td>
</tr>
<tr>
<td>(1050)</td>
<td>When others</td>
</tr>
</tbody>
</table>

Jump table has 6 entries:

| 0 | JUMP 0010 |
| 1 | JUMP 0017 |
| 2 | JUMP 0192 |
| 3 | JUMP 0198 |
| 4 | JUMP 1050 |
| 987 | JUMP 1000 |

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:

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.

Case x is
(0010) When 0: stmts1
(0017) When 1: stmts2
(0192) When 2: stmts3
(1050) When others stmts4;

\[ O(n) \text{ time, } n \text{ is the size of the table, for each jump.} \]

Dealing with jump tables

\[
\begin{align*}
\text{switch (\texttt{<expr>})} \\
\text{case \texttt{<const_list>}: \texttt{<stmt_list>} } \\
\text{case \texttt{<const_list>}: \texttt{<stmt_list>} } \\
\vdots \\
\text{default: \texttt{<stmt_list>} } \\
\end{align*}
\]

- 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