

## If statements

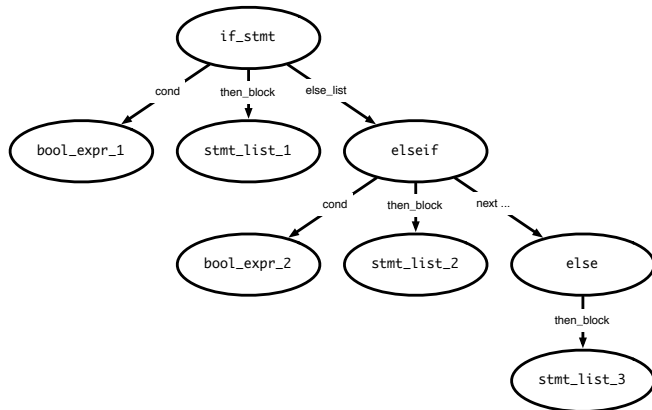
# Processing control structures

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

Wednesday, February 23, 2011

Wednesday, February 23, 2011

## If statements



Wednesday, February 23, 2011

## 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
```

```
<code for bool_expr_1>
j<op> ELSE_1
<code for stmt_list_1>
jmp OUT
ELSE_1:
<code for bool_expr_2>
j<op> ELSE
<code for stmt_list_2>
jmp OUT
ELSE:
  <code for stmt_list_3>
OUT:
```

Wednesday, February 23, 2011

## 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!

Wednesday, February 23, 2011

## Directly generating binary code

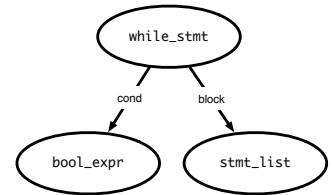
- Recall difference between assembly code and machine code
- Assembly code must be processed by assembler, machine code directly executable
- One job of assembler: decide actual addresses to jump to instead of labels
- So what happens if we generate binary directly?
  - Need to insert JMP instructions 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

Wednesday, February 23, 2011

# Processing Loops

## While loops

```
while <bool_expr> do
  <stmt_list>
end
```



Wednesday, February 23, 2011

Wednesday, February 23, 2011

## Generating code for while loops

```
while <bool_expr> do
  <stmt_list>
end
```

```

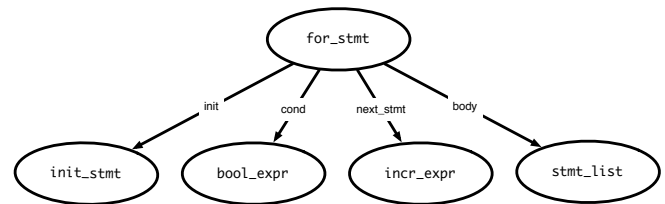
↓
LOOP:
  <bool_expr>
  j<!op> OUT
  <stmt_list>
  jmp LOOP
OUT:
```

- 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?

Wednesday, February 23, 2011

## For loops

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



Wednesday, February 23, 2011

## 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?

Wednesday, February 23, 2011

## continue and break statements

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

```

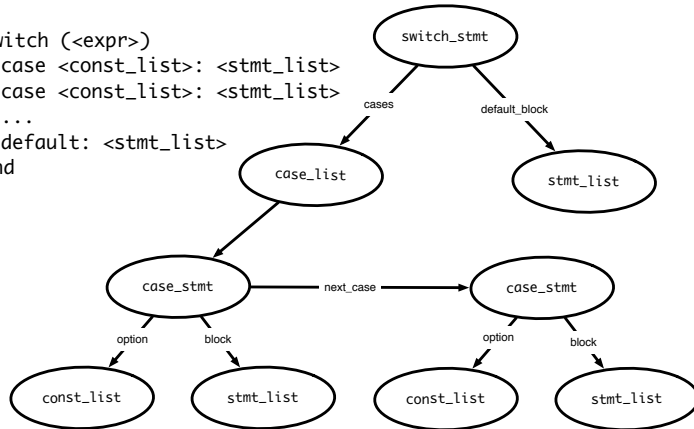
↓
<init_stmt>
LOOP:
  <bool_expr>
  j<!op> OUT
  <stmt_list>
INCR:
  <incr_stmt>
  jmp LOOP
OUT:
```

- 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

Wednesday, February 23, 2011

## Switch/case statements

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



Wednesday, February 23, 2011

## 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?

Wednesday, February 23, 2011

## 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

Wednesday, February 23, 2011

## Jump table example

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

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

Table only has one  
Unnecessary row  
(for choice 4)

Jump table has 6 entries:

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

Wednesday, February 23, 2011

## Jump table example

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

Case x is  
(0010) When 0: stmts0  
(0017) When 1: stmts1  
(0192) When 2: stmts2  
(0198) When 3: stmts3  
(1000) When 987: stmts4  
(1050) When others: stmts5

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!**

Jump table has 6 entries:

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

Wednesday, February 23, 2011

## Do a binary search

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

Case x is  
(0010) When 0: stmts0  
(0017) When 1: stmts1  
(0192) When 2: stmts2  
(0198) When 3: stmts3  
(1000) When 987: stmts4  
(1050) When others: stmts5

Jump table has 6 entries:

0	JUMP 0010
1	JUMP 0017
2	JUMP 0192
3	JUMP 0198
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.

Wednesday, February 23, 2011

## Linear search example

Consider the code:  
(xxxx) 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:  
    <stmt_list>  
  LABEL1:  
    <stmt_list>  
  ...  
  DEFAULT:  
    <stmt_list>  
  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