Semantic actions for control structures

Statement lists

- So far we have discussed generating code for one assignment statement
- Generating code for multiple statements is easy

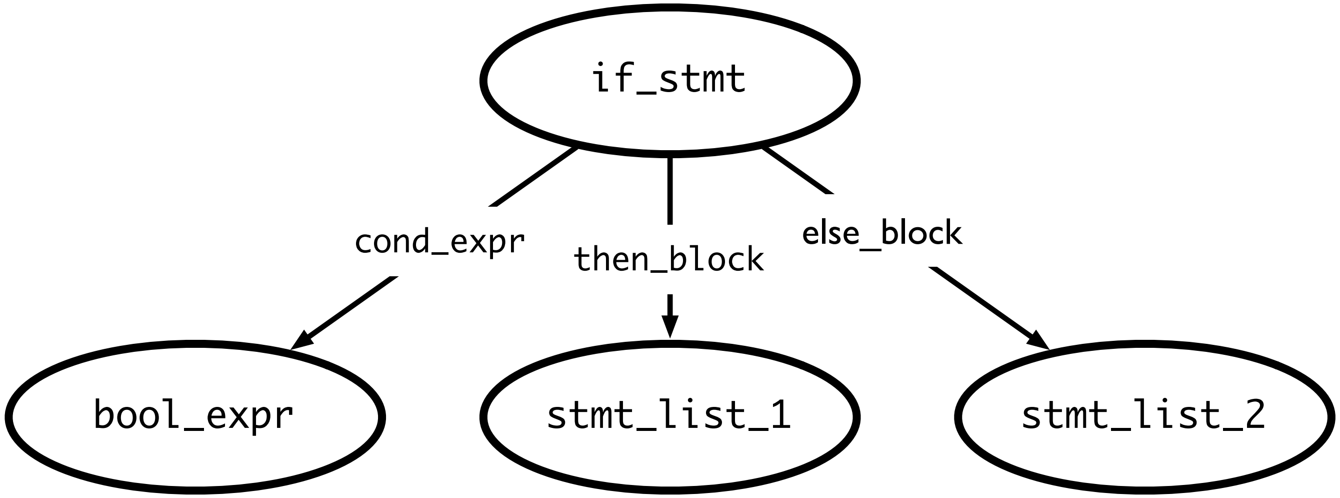
```
stmt_list → stmt stmt_list | λ
```

- Keep appending (or prepending) the code generated by a single statement to the code generated by the rest of the statement list
- What if statement is not an assignment?

If statements

```
if <bool_expr_1>
  <stmt_list_1>
else
  <stmt_list_2>
endif
```

If statements



Generating code for ifs

```
if <bool_expr_1>
  <stmt_list_1>
else
  <stmt_list_2>
endif
```

```
  <code for bool_expr_1>
  j<!op> ELSE_1
  <code for stmt_list_1>
  jmp OUT_1
ELSE_1:
  <code for stmt_list_2>
OUT_1:
```

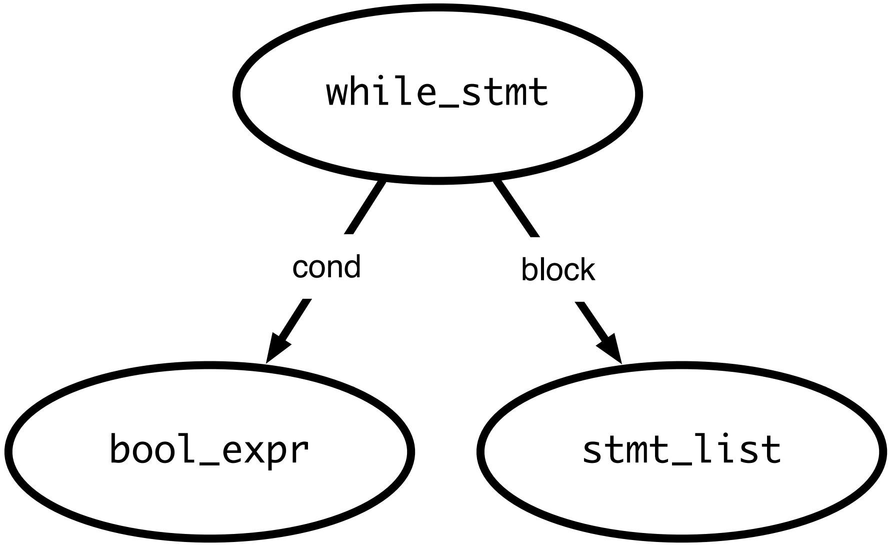
Notes on code generation

- 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, generate the labels when you first process the if statement (i.e., before you process the children) so that it's available when necessary
- Remember: labels have to be unique!

Processing Loops

While loops

```
while <bool_expr>
  <stmt_list>
endwhile
```



Generating code for while loops

```
while <bool_expr>
  <stmt_list>
endwhile;
```

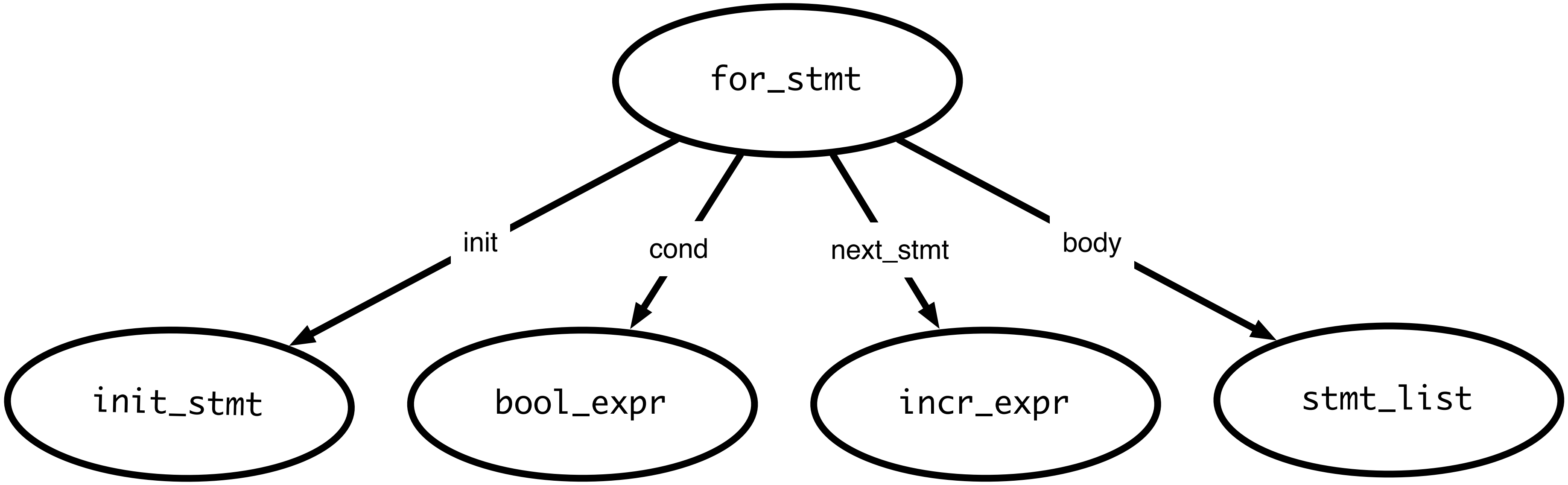
↓

```
LOOP:
  <bool expr>
  j<!op> OUT
  <stmt_list>
  jmp LOOP
OUT:
```

- Re-evaluate expression each time
- Question: what would code for "repeat until" loop look like? For "do while"?

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:
```

- 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

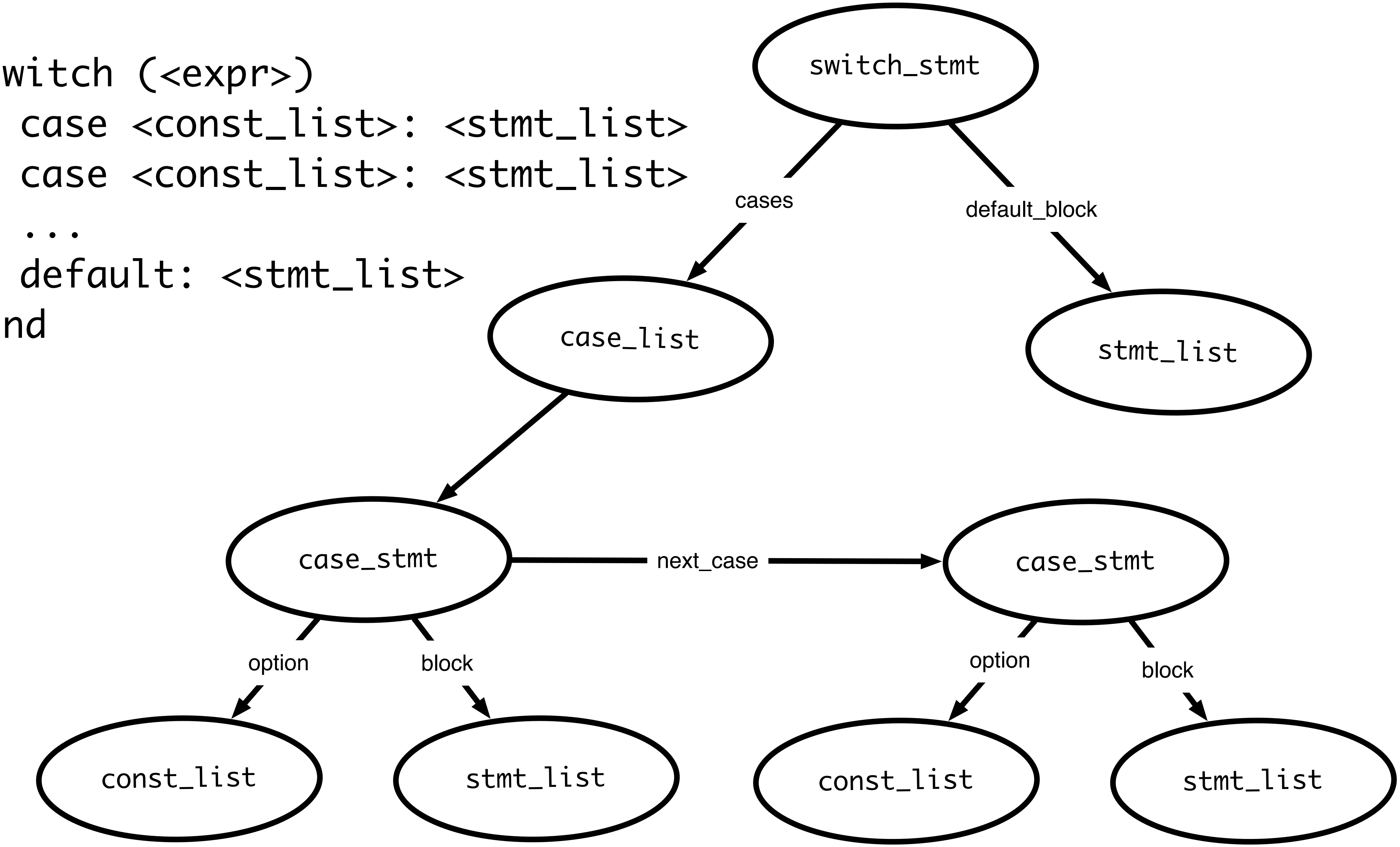
```
for (<init_stmt>;<bool_expr>;<incr_stmt>)
  <stmt_list>
end
```

```
  <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 statements

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



Switch 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)

```
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;
```

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 one
Unnecessary row
(for choice 4)

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

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

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)

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.

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