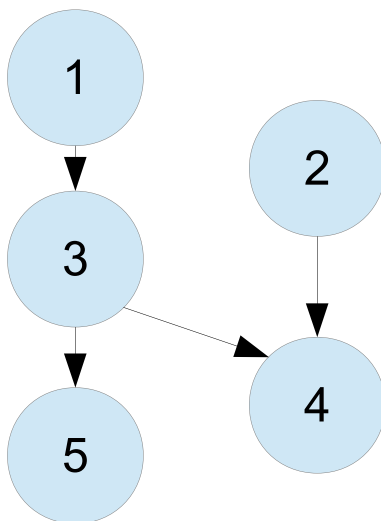
Code gen: selection and iteration

- Recap: our code can be broken into blocks
- Each block has a unique entry point at the top of the block (no mid-block labels)
- Each block has a unique exit point at the end of the block (no mid-block branches)
- Assuming we number or label each block, we could identify each block by its entry/exit point
- Control-flow graphs can model the possible block sequences using nodes for blocks and directed edges for control transfers

Instructions within a block

- instructions within a block can often be reordered safely
- directed graphs can represent the partial ordering

1) x=1
2) y=2
3) z=x+1
4) y=y+z
5) x=x+1



Possible orders
12345
12354
13245
13254
13524
21345
21354

(We'll consider the instruction scheduling problem in the optimization section)

Selection: if/else

- largely covered in our earlier boolean ops section
- if (cond) BLOCKA else BLOCKB
- implementation might prioritize one block over the other
  - is one block much more likely to run than the other?
  - is one block longer or slower than the other?
  - does one block contain more nested-ifs than the other?

Selection: switch/case

- Could implement simply as if a cascading if/else sequence
- switch (x)
  - case A: blockA
  - case B: blockB
  - case C: blockC
  - ...
  - default: blockX

```
if X==A then blockA
else
  if X==B then blockB
  else
    if X==C then blockC
    else
      ...
         else blockX
```

Optimize for large sets of cases

- given large set of values, rewrite as a table of values and block labels
- Insert code to perform binary search on the values, then jump to the correct label
- O(logN) execution time instead of O(N) , at the expense of generating more complex code

Optimize for sequential integers

- case values are often an ascending range of integer values or character codes (e.g. cases 3,4,5,6,...,17 or 'a','b',...'k')
- implement like an array of labels to the code blocks
- use the case value to compute the correct array position offset = (x - base index)*storagesize
- look at value of x, compute offset, jump to right spot in “array” and get label of the desired block

Iteration: loops

- details depend heavily on available test/branch operations
- initially we might generate non-optimal assembly constructs for a loop, with the goal of making later optimizations more easily applicable
- tends to show up as having a test at loop entry then another very similar test for loop continuation, will re-visit when we get to optimizations

for x=m to n by i do blockA

```
    load m,Rx
    load n,Rn
    load i,Ri
    compareGT Rx,Rn,Rc1 // in case m > n
    branch Rc1,Exit,Entry
Entry: blockA code
    add Ri,Rx,Rx
    compareGT Rx,Rn,Rc2
    branch Rc2,Exit,Entry
Exit:
```

while (m < n) do blockA

```
    load m,Rm
    load n,Rn
    compareGE rm,rn,Rc1
    branch Rc1,Exit,Entry
Entry: blockA code
    compareGE rm,rn,Rc2
    branch Rc2,Exit,Entry
Exit:
```

repeat blockA until (m < n)

```
    load m,Rm
    load n,Rn
Entry: blockA code
    compareGE Rm,Rn,Rc
    branch Rc,Exit,Entry
Exit:
```

Break, continue, goto

- Each can be a single unconditional jump
  - break jumps to Exit
  - goto jumps to specified location
  - continue jumps to Entry