NTNU – Trondheim
Norwegian University of
Science and Technology

Control Flow Graphs

Optimizations

- We wish to apply various program transformations to improve its performance without altering its meaning
- Transformations apply at either high or low IR levels

- Optimizations must be *safe*
  - That is, the optimized program must give the same results as the un-optimized program for **every possible execution**

Program meaning is implicit

- The information we require is not necessarily written plainly in the source code
- Consider:

```
x = y + 1
y = 2 * z
x = y + z
z = 1
z = x
```

- Are all these statements necessary?

Program meaning is implicit

- Some of the statements are *dead code*

```
x = y + 1    ← This assignment of x...
y = 2 * z    ← ...is not used in any intermediate statement...
x = y + z    ← ...until x is assigned again
z = 1        ← This assignment of z...
z = x        ← ...is immediately overwritten
```

- Noticing this, we can tell that

```
y = 2 * z
x = y + z
z = x
```

  is an equivalent program
- Control flow is linear here, so dead state is obvious
- It gets harder to tell when control flow gets complicated

Conditions complicate the matter

- Adding some control flow,

```
x = y + 1             ← is this statement still dead?
y = 2 * z
if ( c ) { x = y + z }
z = 1                 ← is this statement still dead?
z = x
```

- The first assignment of x may or may not be used:

```
x = y + 1
y = 2 * z
if ( c ) { x = y + z }
z = 1                 ← This assignment makes no difference
z = x
```

This assignment is relevant when c is false

NTNU – Trondheim
Norwegian University of
Science and Technology

Loops complicate the matter

- If we insert a loop...

```
while ( d ) {
  x = y + 1           ← is this statement still dead?
  y = 2 * z
  if ( c ) { x = y + z }
  z = 1               ← is this statement still dead?
}
z = x
```

...neither statement can be omitted

```
while ( d ) {
  x = y + 1
  y = 2 * z
  if ( c ) { x = y + z }
  z = 1
}
z = x
```

The assignment is relevant when there is another iteration of the loop

NTNU – Trondheim
Norwegian University of Science and Technology

Low-level code complicates the matter

- Control flow is more obvious from source code syntax than from its translation into jumps and labels:

```
L1:
ifFalse d jump L2
x = y + 1
y = 2 * z
ifFalse c jump L3
x = y + z
L3:
z = 1
jump L1
L2:
z = x
```

Data dependencies

Control flow

NTNU – Trondheim
Norwegian University of Science and Technology

What we need

- Methods to compute information that are
  - implicit in the program
  - static (so that it can be found at compile time)
  - valid for every possible dynamic situation (at run time)
- A data structure that can represent every possible control flow
  - Different branches taken (conditionals)
  - Branches taken different numbers of times (loops)
- Problem is similar to that of NFA:

  "What are all the possible paths I can take from here?"

Control Flow Graphs (CFGs)

- Program control flow can be captured in a directed graph, where statements make nodes and their sequencing follows the arcs
- Movement of data can be inferred by traversing a structure like this
  - By far the most common approach in present compilers

    (It is also possible to graph data movement and infer control, but let's stick to the control flow view)
- Multiple paths emerge since nodes can have multiple incoming/outgoing arcs

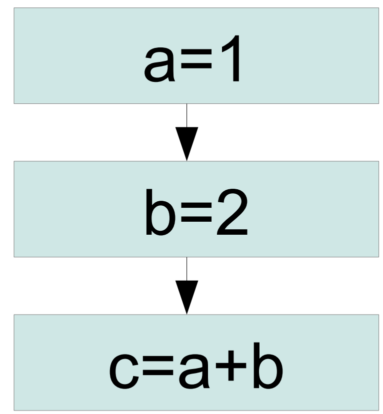
Linear sequences

- These are a bit boring:

  a = 1

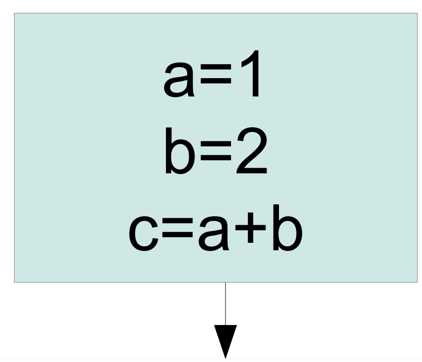
  b = 2

  c = a + b



- Therefore, we contract them to *basic blocks*

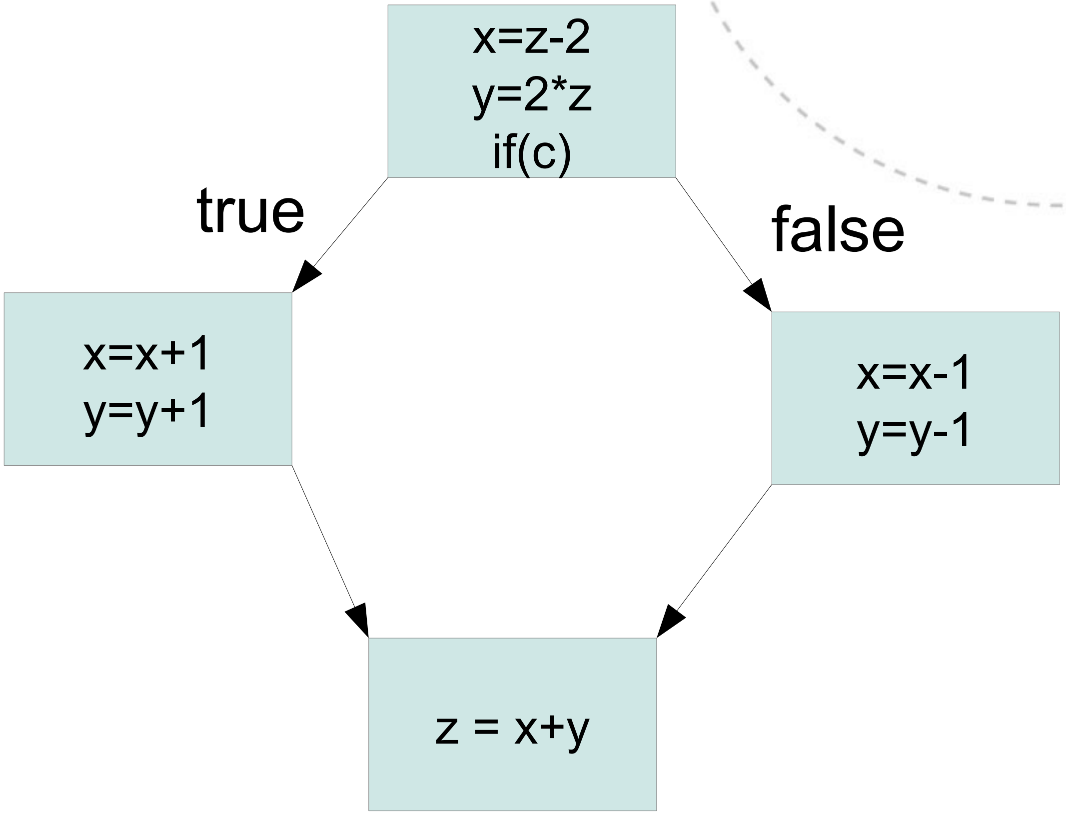
(but remember that there are separate statements inside...)



Branches end basic blocks

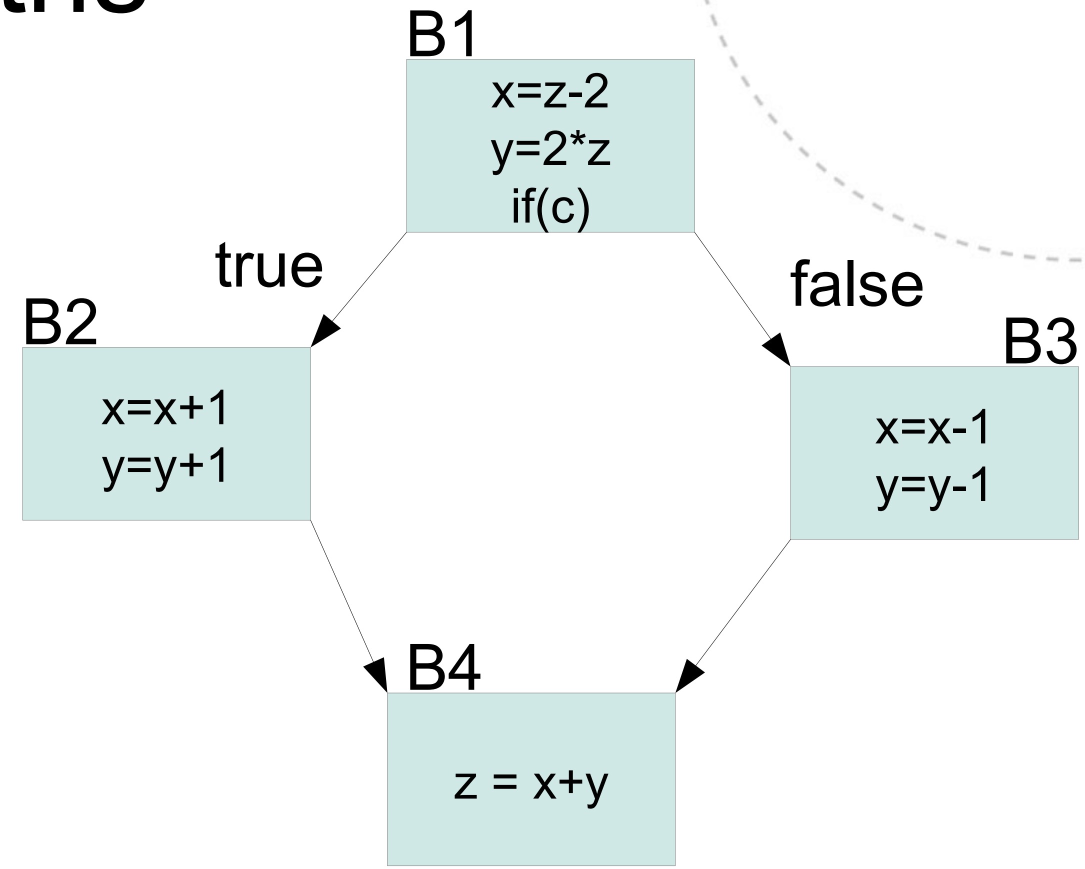
- Consider:

```
x = z-2
y = 2*z
if ( c ) {
    x = x+1
    y = y+1
}
else {
    x = x-1
    y = y-1
}
z = x + y
```

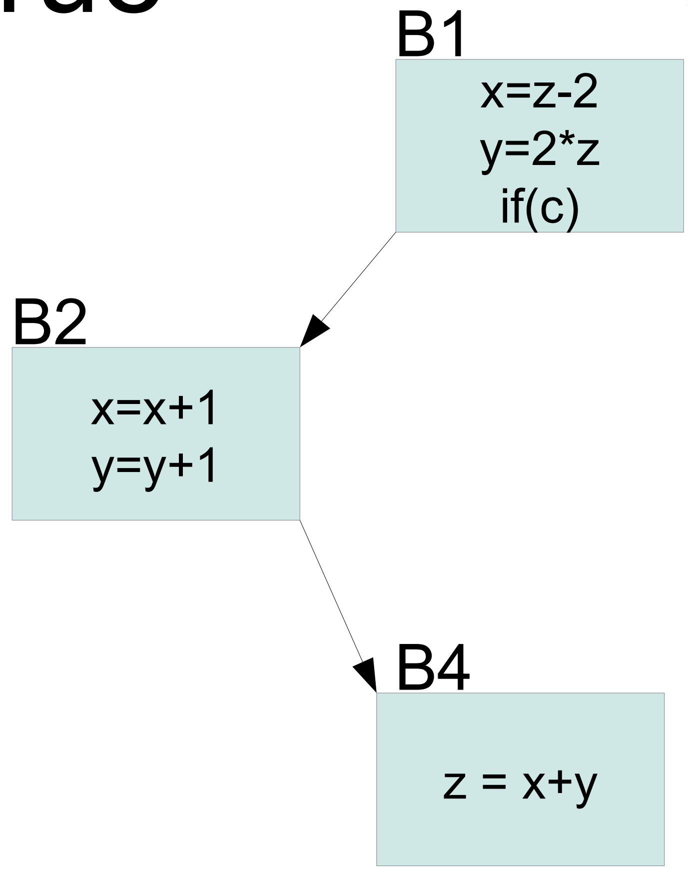


Multiple paths

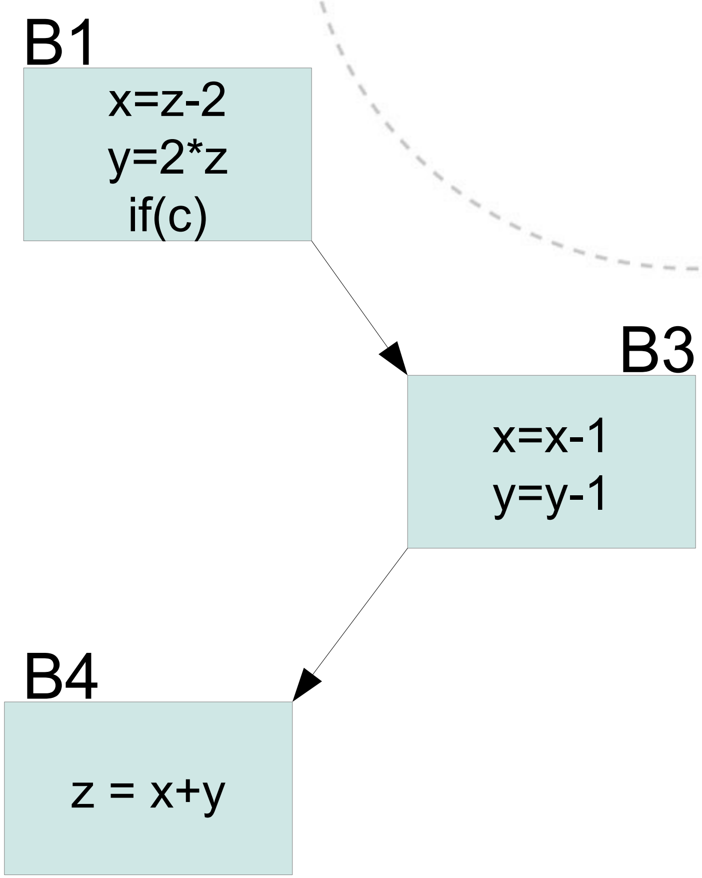
- Every possible execution is encoded in the CFG
- Each path corresponds to a run of the program



When c is true

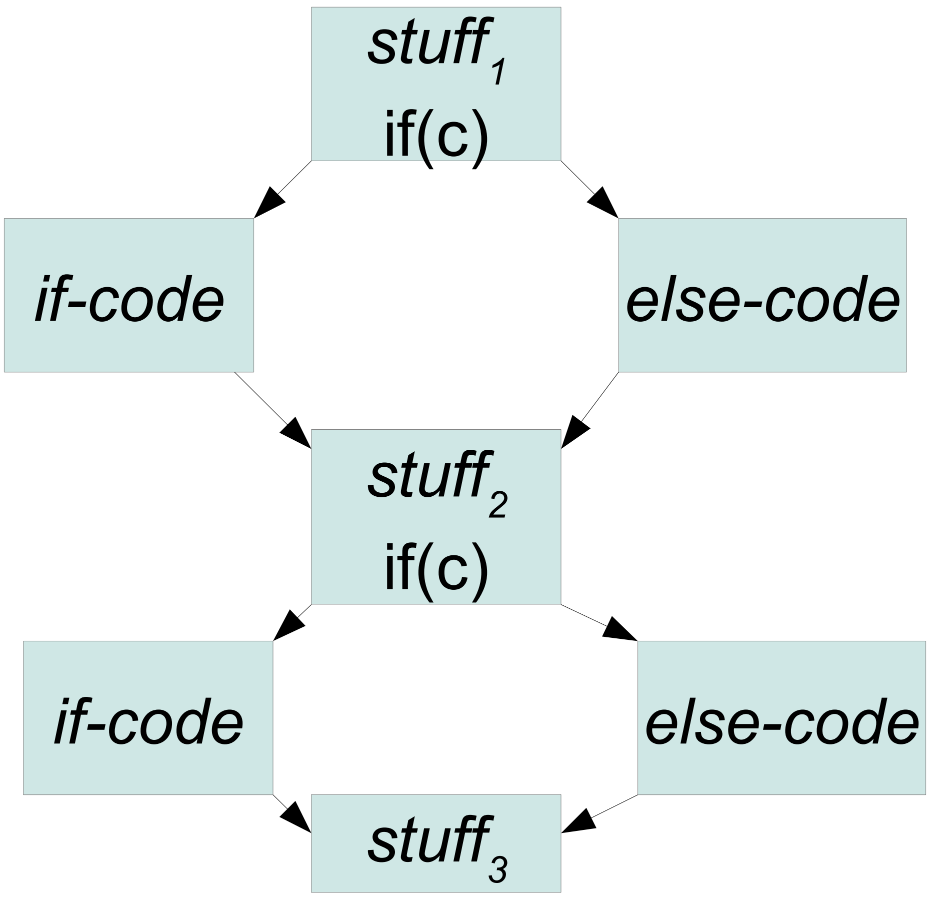


When c is false



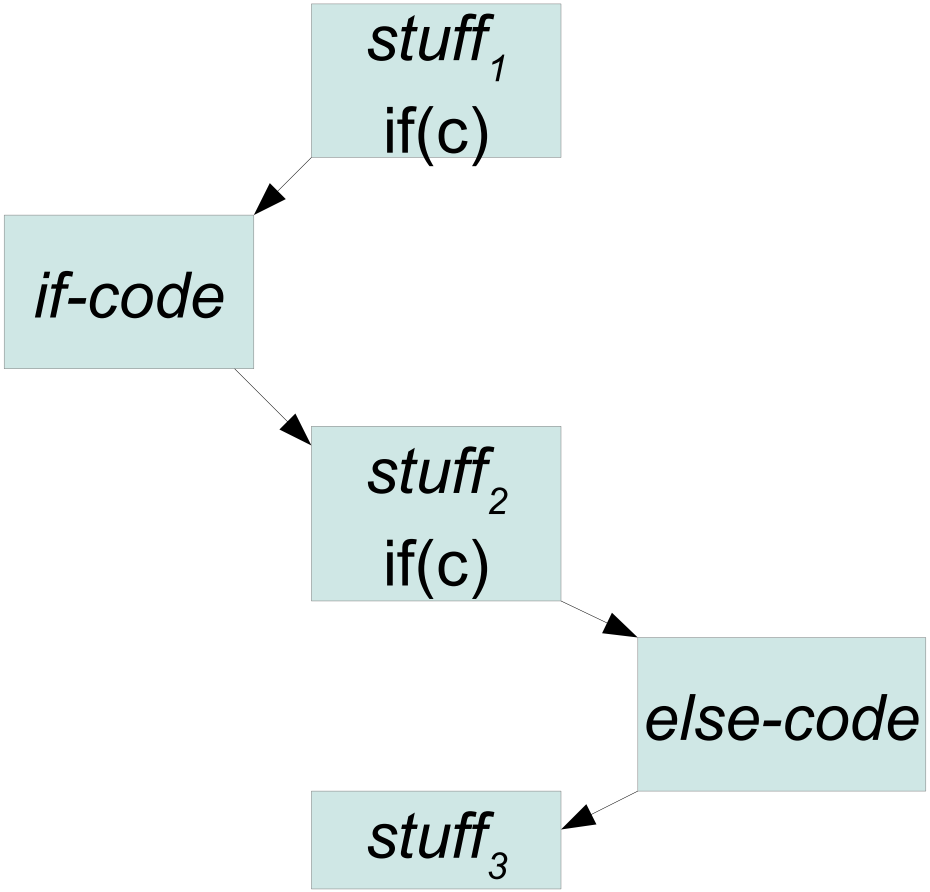
Infeasible executions

- Some paths may not correspond to any run



Infeasible executions

- Unless either branch modifies *c*, this path won't occur, even though the CFG contains it:



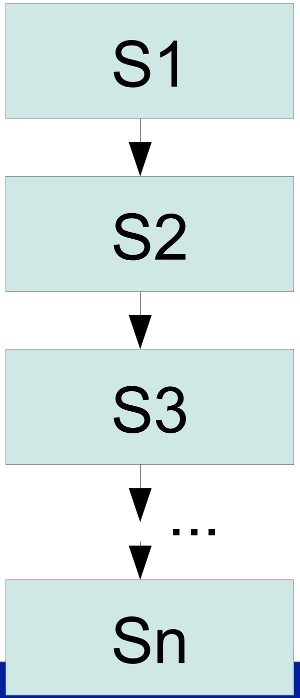
NTNU – Trondheim
Norwegian University of Science and Technology

Interpretation of arcs

- Without pruning infeasible paths (which may require run-time information), the analysis will remain conservative/safe as long as every actual path is also represented
- Outgoing arcs mean that their destination *may be* a successor to a basic block
- Incoming arcs mean that any of the source blocks *may be* a predecessor to a basic block

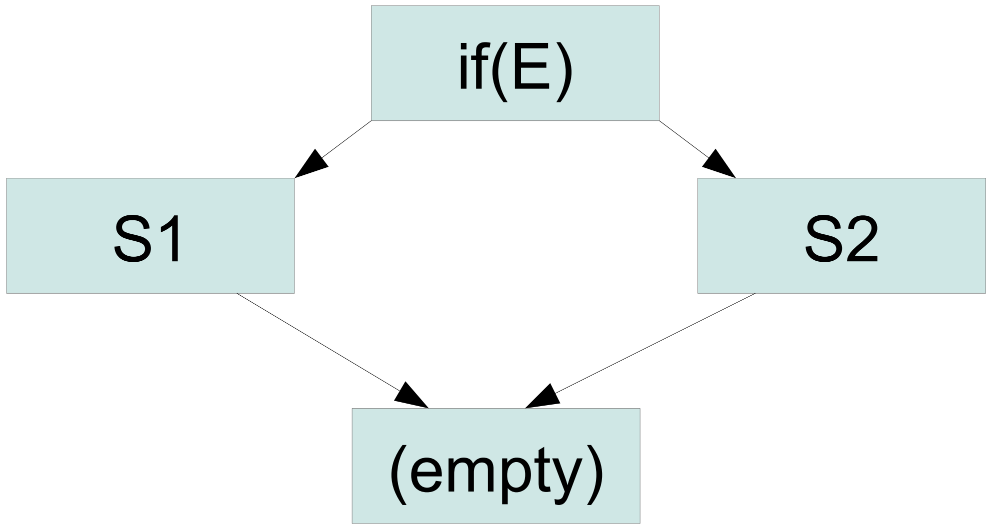
Recursive CFG construction

- At high level, CFGs can be built by a syntax directed scheme, like our TAC translation patterns:
- CFG (S1; S2; S3; … ; Sn) =



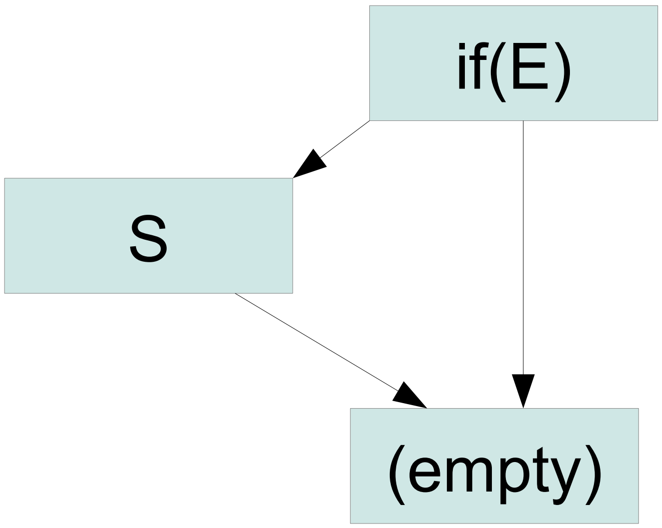
Recursive CFG construction: if-else

- CFG (if (E) S1 else S2) =



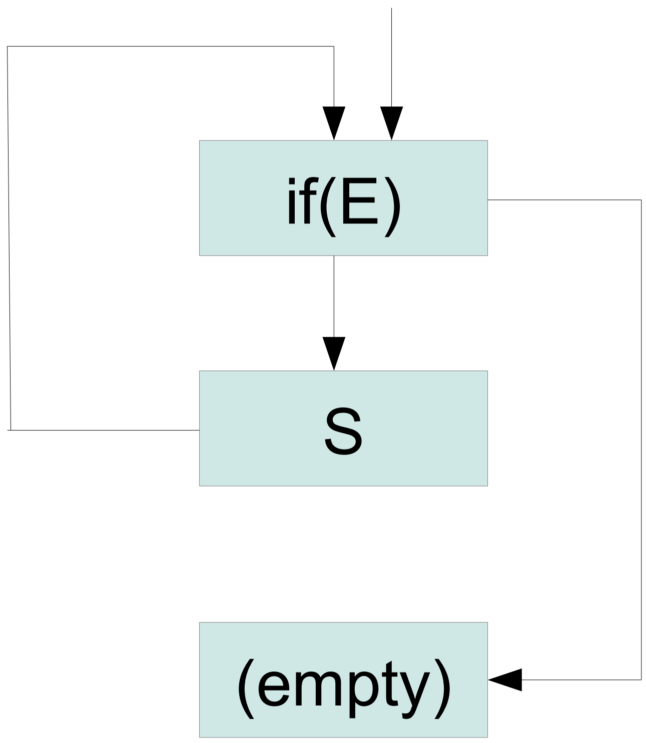
Recursive CFG construction: if

- CFG (if (E) S) =



Recursive CFG construction: while

- CFG (while (E) S) =

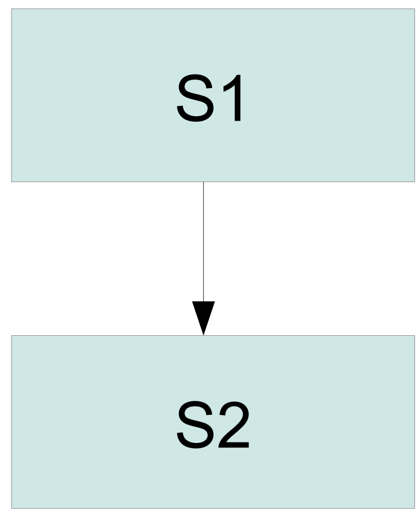


Recursive application

- As long as every statement is treated recursively, the whole becomes the sum of its parts:

```
while ( c ) {
  x = y + 1
  y = 2 * z
  if ( d ) x = y+z
  z = 1
}
z = x
```

(S1;S2)

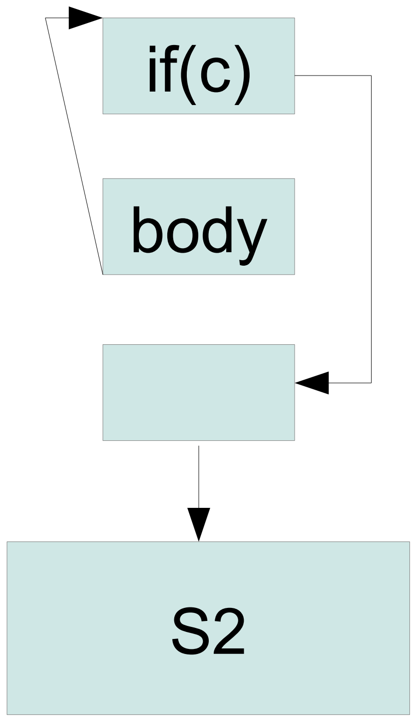


Recursive application

- As long as every statement is treated recursively, the whole becomes the sum of its parts:

```
while ( c ) {
  x = y + 1
  y = 2 * z
  if ( d ) x = y+z
  z = 1
}
z = x
```

(while)



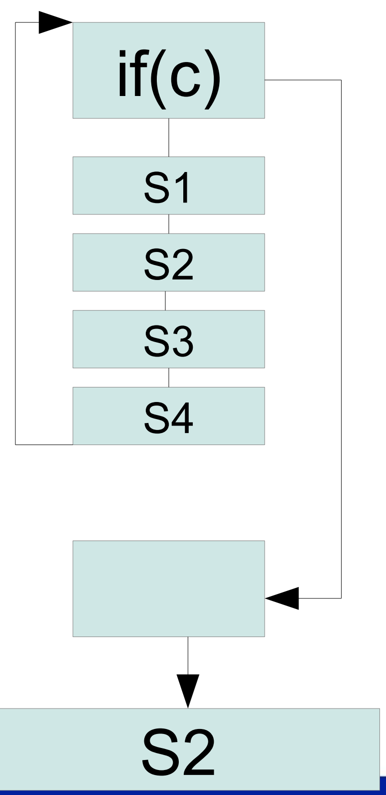
NTNU – Trondheim
Norwegian University of Science and Technology

Recursive application

- As long as every statement is treated recursively, the whole becomes the sum of its parts:

```
while ( c ) {
  x = y + 1
  y = 2 * z
  if ( d ) x = y+z
  z = 1
}
z = x
```

(S1;S2;S3;S4)

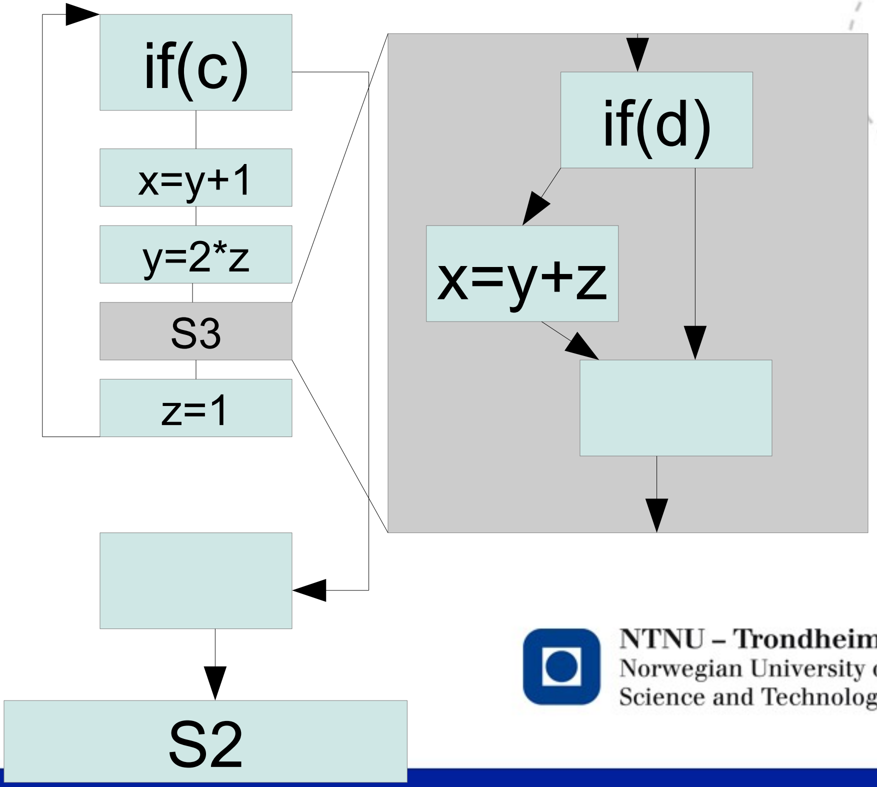


Recursive application

- As long as every statement is treated recursively, the whole becomes the sum of its parts:

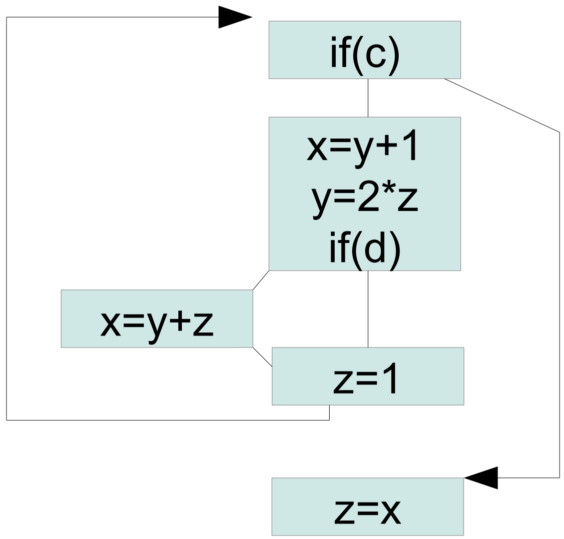
```
while ( c ) {
  x = y + 1
  y = 2 * z
  if ( d ) x = y+z
  z = 1
}
z = x
```

(S1;S2;S3;S4)



Efficiency

- Empty blocks and sequences can be pruned after or during construction



Efficiency

- These graphs grow large
  - It’s good to have as few basic blocks as possible
  - They should be as large as possible
- Merge linear subgraphs - if
  - B2 is a successor of B1
  - B1 has one outgoing edge
  - B2 has one incoming edge

    B1→ B2 should be a block
- Remove empty blocks

At low-level IR

- Split the operation sequence at labels and jumps
  - Labels can have incoming control flow
  - Jumps have outgoing control flow

```
L1:
ifFalse(c) jump L2
x=y+1
y=2*z
ifFalse(d) jump L3
x=y+z
L3:
z=1
jump L1
L2:
z=x
```

```
L1:
ifFalse(c) jump L2
```

```
x=y+1
y=2*z
IfFalse d jump L3
```

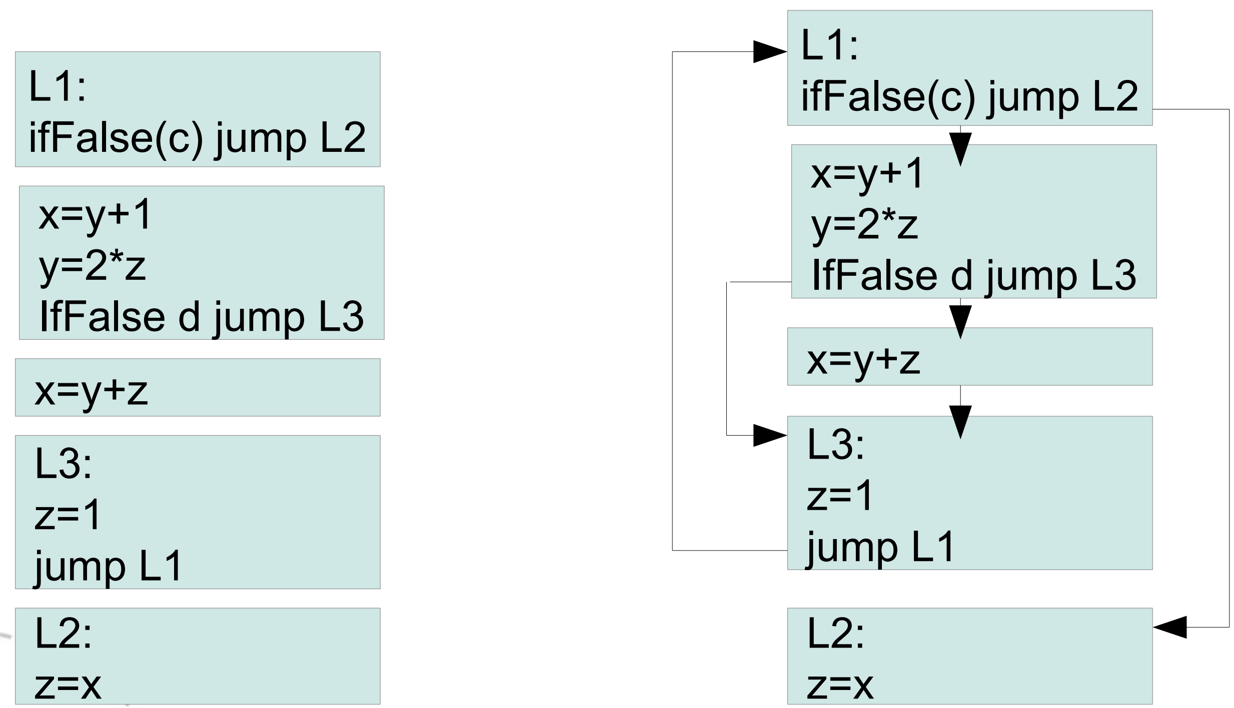
```
x=y+z
```

```
L3:
z=1
jump L1
```

```
L2:
z=x
```

At low-level IR

- Conditional jump = 2 successors
- Unconditional jump = 1 successor



The outcome is the same

- Both procedures give us equivalent program logic: