Optimizations
• Code transformations to improve program
  – Mainly: improve execution time
  – Also: reduce program size
• Can be done at high level or low level
  – E.g. constant folding
• Optimizations must be safe
  – Execution of transformed code must yield same results as
    the original code for all possible executions

Optimization Safety
• Safety of code transformations usually requires certain
  information which may not explicit in the code
• Example: dead code elimination
  1. \( x = y + 1; \)
  2. \( y = 2 \cdot z; \)
  3. \( x = y + z; \)
  4. \( z = 1; \)
  5. \( z = x; \)
• What statements are dead and can be removed?

Dead Code Example
• Add control flow:
  \[
  x = y + 1; \\
  y = 2 \cdot z; \\
  \text{if (d) } x = y + z; \\
  z = 1; \\
  z = x;
  \]
• Is 'x = y+1' dead code? Is 'z = 1' dead code?
Dead Code Example

• More complex control flow:
  
  ```
  while (c) {
      x = y + 1;
      y = 2 * z;
      if (d) x = y + z;
      z = 1;
  }
  z = x;
  ```
  Is 'x = y+1' dead code? Is 'z = 1' dead code?

Dead Code Example

• Add a while loop:
  
  ```
  while (c) {
      x = y + 1;
      y = 2 * z;
      if (d) x = y + z;
      z = 1;
  }
  z = x;
  ```
  Statement 'x = y+1' not dead (as before)
  Statement 'z = 1' not dead either!
  On some executions, value from 'z+1' is used later

Three-Address Code

• Much harder to understand code in three-address form:

  ```
  label L1
  fjump c L2
  x = y + 1;
  y = 2 * z;
  fjump d L3
  z = y + z;
  label L3
  z = 1;
  jump L1
  label L2
  z = x;
  ```

Are these statements dead?

Three-Address Code

• Much harder to understand code in three-address form:

  ```
  label L1
  fjump c L2
  x = y + 1;
  y = 2 * z;
  fjump d L3
  z = y + z;
  label L3
  z = 1;
  jump L1
  label L2
  z = x;
  ```

It is harder to analyze flow of control in low level code

Optimizations and Control Flow

• Applying optimizations requires information
  – Dead code elimination: need to know if variables are dead
    when assigned values

• Required information:
  – Not explicit in the program
  – Must compute it statically (at compile-time)
  – Must characterize all dynamic (run-time) executions

• Control flow makes it hard to extract information
  – Branches and loops in the program
  – Different executions = different branches taken, different
    number of loop iterations executed

Control Flow Graphs

• Control Flow Graph (CFG) = graph representation of
  computation and control flow in the program
  – framework to statically analyze program control-flow

• Nodes are single instructions, edges describe flow of
  control.
  – Common optimization: use basic blocks as CFG nodes
  – basic blocks = sequences of consecutive non-branching
    statements
CFG Example

Program

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

Control Flow Graph

![Diagram showing control flow graph with nodes labeled B1 to B3 and T/F edges]

Basic Blocks

- Basic block = sequence of consecutive statements such that:
  - Control enters only at beginning of sequence
  - Control leaves only at end of sequence
- No branching in or out in the middle of basic blocks

Computation and Control Flow

- Basic Blocks = Nodes in the graph = computation in the program
- Edges in the graph = control flow in the program

Control Flow Graph

![Diagram showing control flow graph with nodes labeled B1 to B3 and T/F edges]

Multiple Program Executions

- CFG models all program executions
- Possible execution = path in the graph
- Multiple paths = multiple possible program executions

Control Flow Graph

![Diagram showing control flow graph with nodes labeled B1 to B3 and T/F edges]

Execution 1

- CFG models all program executions
- Possible execution = path in the graph
- Execution 1:
  - C is true
  - Program executes basic blocks B1, B2, B4

Control Flow Graph

![Diagram showing control flow graph with nodes labeled B1 to B3 and T/F edges]

Execution 2

- CFG models all program executions
- Possible execution = path in the graph
- Execution 2:
  - C is false
  - Program executes basic blocks B1, B2, B4

Control Flow Graph

![Diagram showing control flow graph with nodes labeled B1 to B3 and T/F edges]
Edges Going Out

- Multiple outgoing edges
- Basic block executed next may be one of the successor basic blocks
- Each outgoing edge = outgoing flow of control in some execution of the program

Building the CFG

- Build CFG from AST / High IR
  - Construct CFG for each High IR node
- Build CFG for three-address IR code
  - Analyze jump and label statements

From AST to CFG

- CFG(S) = flow graph of AST statement S
- CFG(S) is single-entry, single-exit graph:
  - one entry node (basic block)
  - one exit node (basic block)
- Recursively define CFG(S)

CFG for Block Statement

- CFG( S1; S2; ...; SN ) =

CFG for If-then-else Statement

- CFG( if (E) S1 else S2 )
CFG for If-then Statement

- CFG( if (E) S )

```
if (E)
  T
  CFG(S)
  F
```

CFG for While Statement

- CFG for: while (c) S

```
if (e)
  T
  CFG(S)
  F
```

Recursive CFG Construction

- Nested statements: recursively construct CFG while traversing IR nodes
- Example:

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

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

z = x;

From Three-Address Code to CFG

• Identify control in three-address code:
  – Identify label and jump instructions

label L1
fjump c L2
x = y + 1;
y = 2 * z;
fjump d L3
x = y * z;
label L3
z = 1;
jump L1
label L2
z = x;

• Group together to form CFG nodes:
  – Labels and successor instructions
  – Unconditional jumps and predecessor instructions
  – Otherwise, one instruction per node

From Three-Address Code to CFG

• Group together to form CFG nodes:
  – Labels and successor instructions
  – Unconditional jumps and predecessor instructions
  – Otherwise, one instruction per node

From Three-Address Code to CFG

• Group together to form CFG nodes:
  – Labels and successor instructions
  – Unconditional jumps and predecessor instructions
  – Otherwise, one instruction per node