Check-In

Review: Machine-Code Optimization

Give an example of a snippet of x64 code that benefits from two peephole optimizations

Drew Davidson | University of Kansas



Flowgraphs

Previously…

Machine Code Optimization

Machine code optimization overview

Improving data allocation

- Register allocation

Improving Final Code

- Peephole optimization
- Instruction Pipelines

You should know

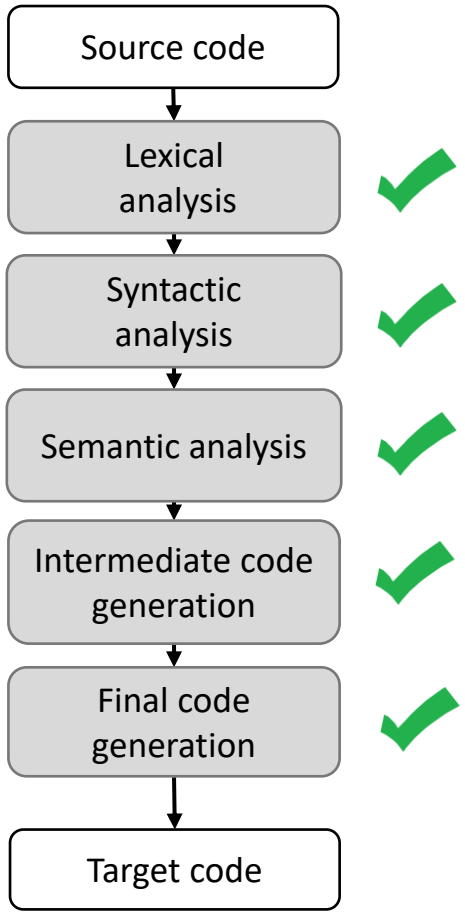
- Interference graphs
- Sharing AR slots / registers for allocation
- How/where to apply peephole optimizations
- How/where instruction reordering might aid a simple instruction pipeline



Optimization

Compiler Construction

Progress Pics



Basic source to target workflow:

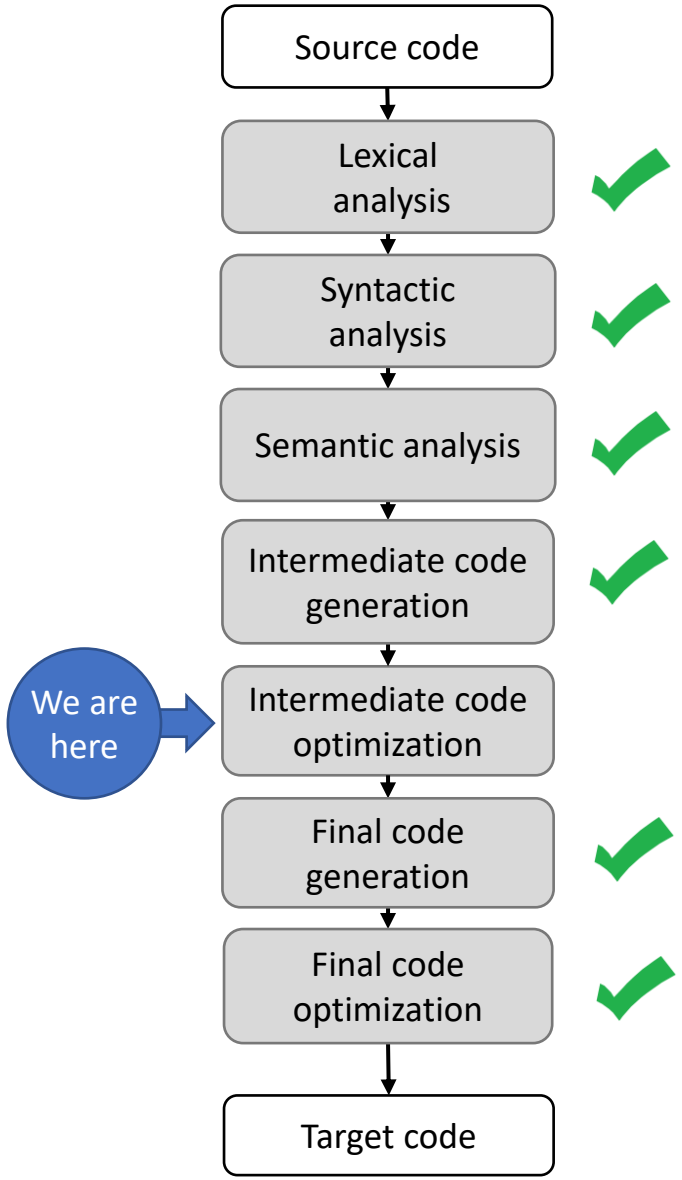
- Complete
- Outputs naïve code

Advanced workflow:

- “Postprocess” the output of a naïve phase

Compiler Construction

Progress Pics



Basic source to target workflow:

- Complete
- Outputs naïve code

Advanced workflow:

- "Postprocess" the output of a naïve phase
  - Discussed: final code "cleanup"
  - Next up: intermediate code

Lecture Outline

Flowgraphs

Program analysis:

- Goals
- Control flow graphs

Local Optimizations

- Dead code elimination
- Common subexpression elimination
- Constant/copy propagation



Optimization

Making faster IR programs

Flowgraphs: Program analysis

General constraints:

- We can't violate program semantics
- Minimal architecture details

Constraint-friendly goals:

- Don't do *useless* computation
- Don't do *redundant* computation



Simple Example: Constant Folding

Statically compute known expressions

- Replace the runtime expression with its value

Before

[z] := 1 + 2

Analysis

- Identify constant expressions
- Compute known value

Rewrite

- Replace expression with value

After

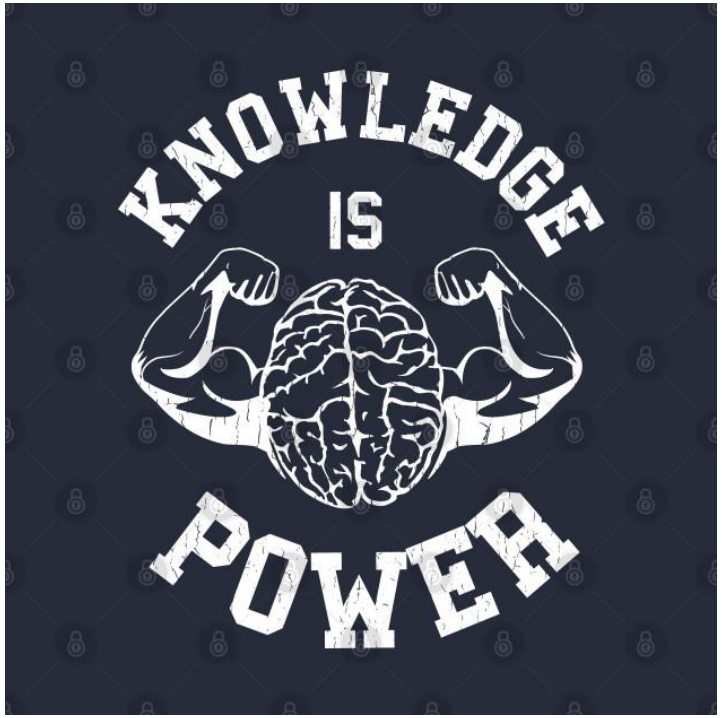
[z] := 3

Program Analysis

Flowgraphs: Program analysis

The more we know about the program the more we can improve it

- What might we be interested in knowing…?



"Structural" Properties of a Program

Flowgraphs: Program analysis

E.g. for a given program point:

- What paths lead there?
- Is it in a deeply nested loop?
- Is it reachable at all?

Knowing the above info supports other analyses

- Might a variable be uninitialized?



"Structural" Properties of a Program

Flowgraphs: Program analysis

E.g. for a given program point:

- What paths lead there?
- Is it in a deeply nested loop?
- Is it reachable at all?

We need a program abstraction to capture these details

Knowing the above info supports other analyses

- Might a variable be uninitialized?

Intuition: Flow charts

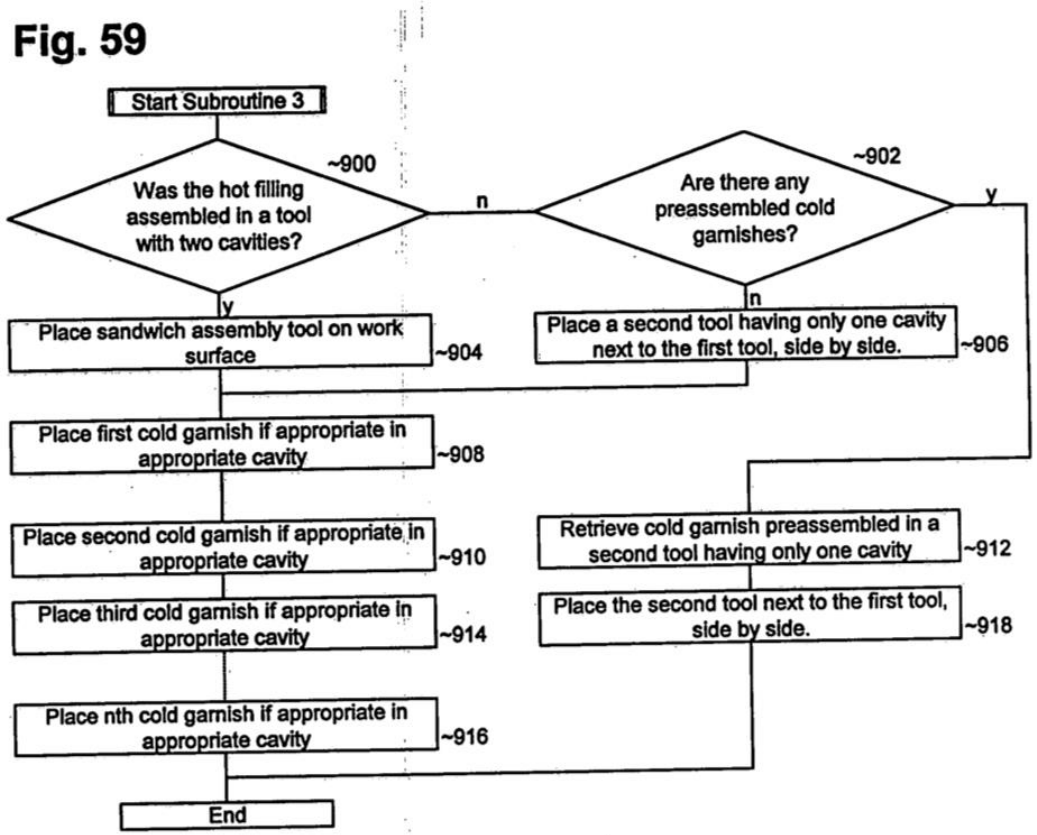
Flowgraphs: Program analysis

Notation

- Nodes are instructions
- Edges go to successor nodes

Operation

- Execute current instruction
- Proceed to the right successor



Flow chart for building a sandwich, appearing in a McDonald's patent

Intuition: Flow charts

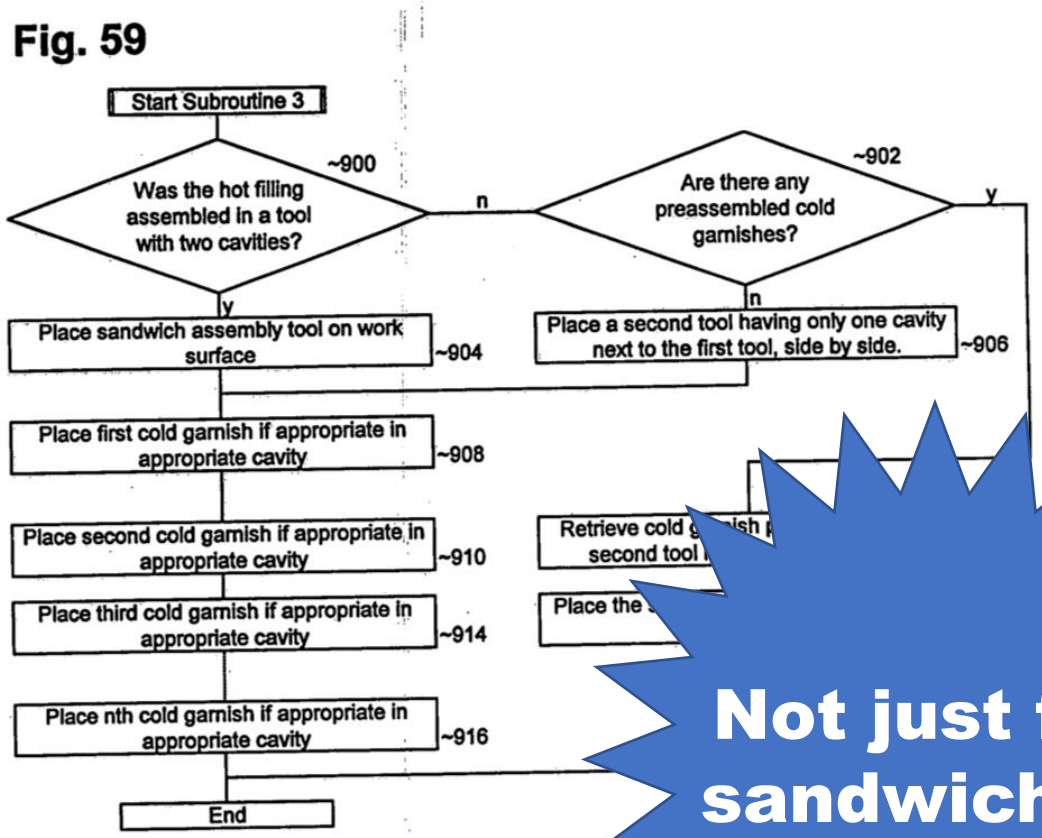
Flowgraphs: Program analysis

Notation

- Nodes are instructions
- Edges go to successor nodes

Operation

- Execute current instruction
- Proceed to the right successor



Flow chart for building a sa… appearing in a McDonald's po…

Not just for sandwiches!

Intuition: Flow Charts … for Code?!

Flowgraphs: Program analysis

Notation

- Nodes are instructions
- Edges go to successor nodes

Operation

- Execute current instruction
- Proceed to the right successor

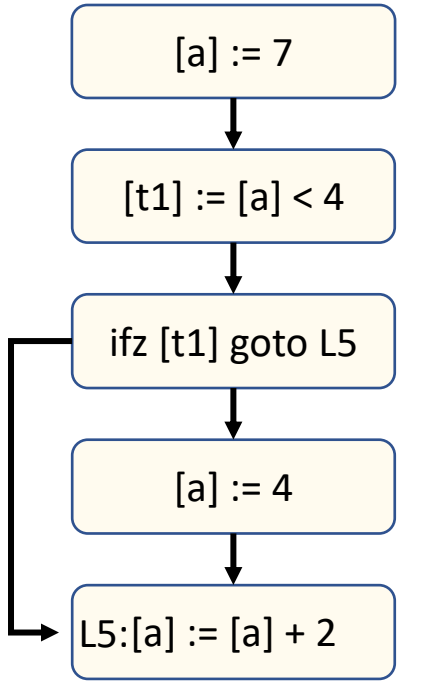
src code

```
a = 7;
if (a < 4){
  a = 4;
}
a += 2;
```

3AC code

```
    1. [a] := 7
    2. [t1] := [a] < 4
    3. ifz [t1] goto L5
    4. [a] := 4
L5: 5. [a] := [a] + 2
```

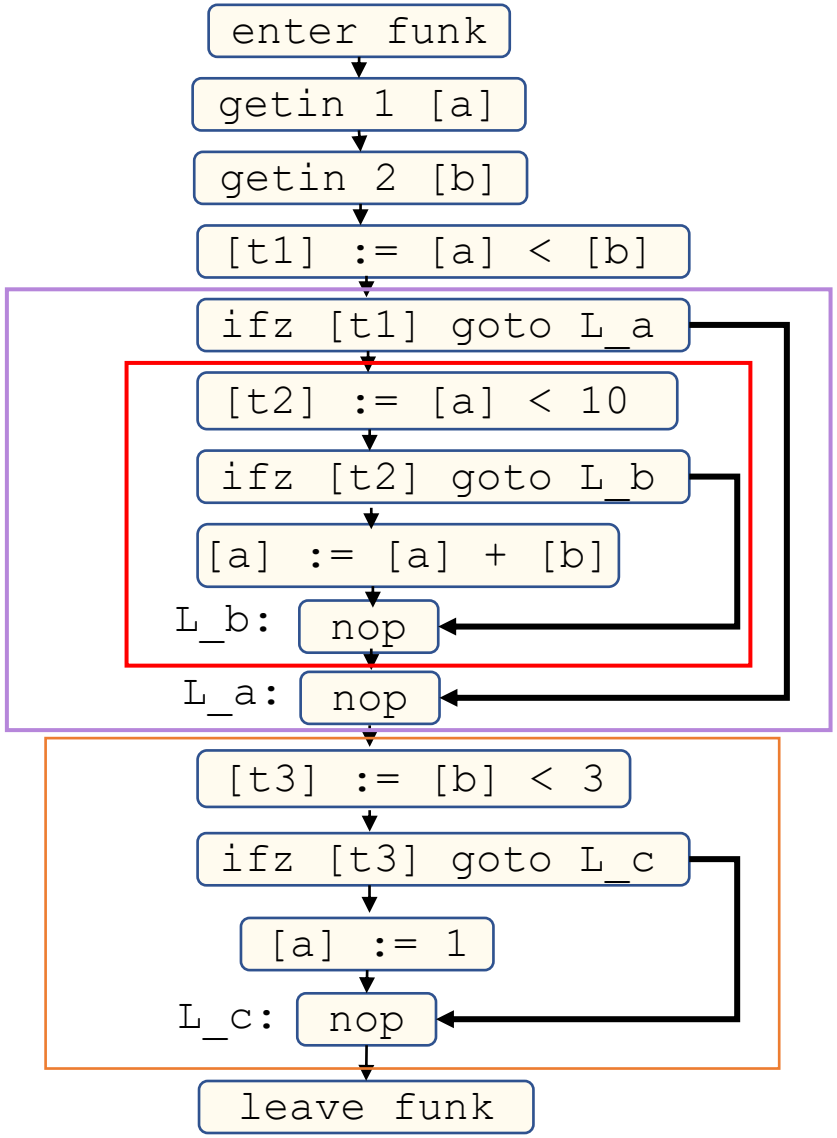
Instruction Flowgraph



Intuition: Flow Charts … FROM Code?!

Flowgraphs: Program analysis

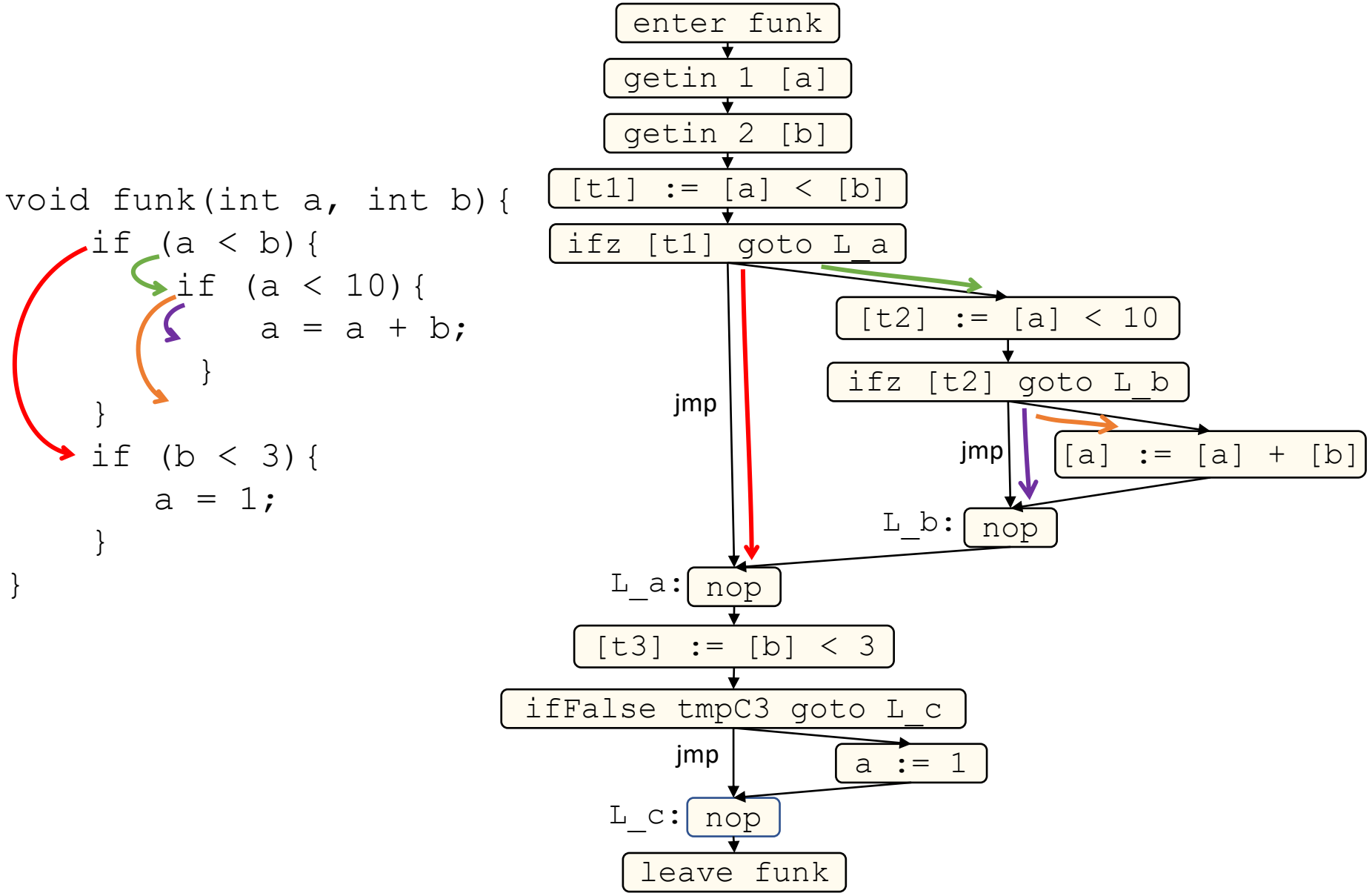
```
void funk(int a, int b){
    if (a < b){
        if (a < 10){
            a = a + b;
        }
    }
    if (b < 3){
        a = 1;
    }
}
```



Intuition: Flow Charts … FROM Code?!

Flowgraphs: Program analysis

```
void funk(int a, int b){
    if (a < b){
        if (a < 10){
            a = a + b;
        }
    }
    if (b < 3){
        a = 1;
    }
}
```



Code Flowcharts: Seem Familiar?

Flowgraphs: Program analysis

Maybe this is how you learned to think about code!

- It's a nice way to visualize the *control flow* of the program
- We can extend this intuition for program analysis



Lecture Outline

Flowgraphs

Program analysis:

- Goals
- Control flow graphs

Local Optimizations

- Dead code elimination
- Common subexpression elimination
- Constant/copy propagation

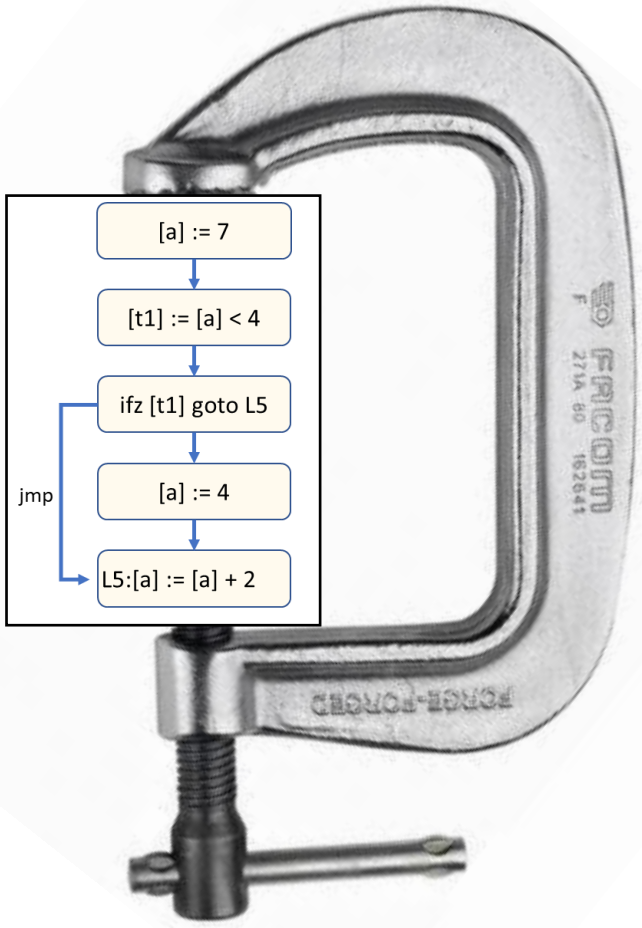


Optimization

Intuition

Flowgraphs: Control flow graphs

- A more compact version of the instruction flow chart
- But still preserves the way in which control passes through the program

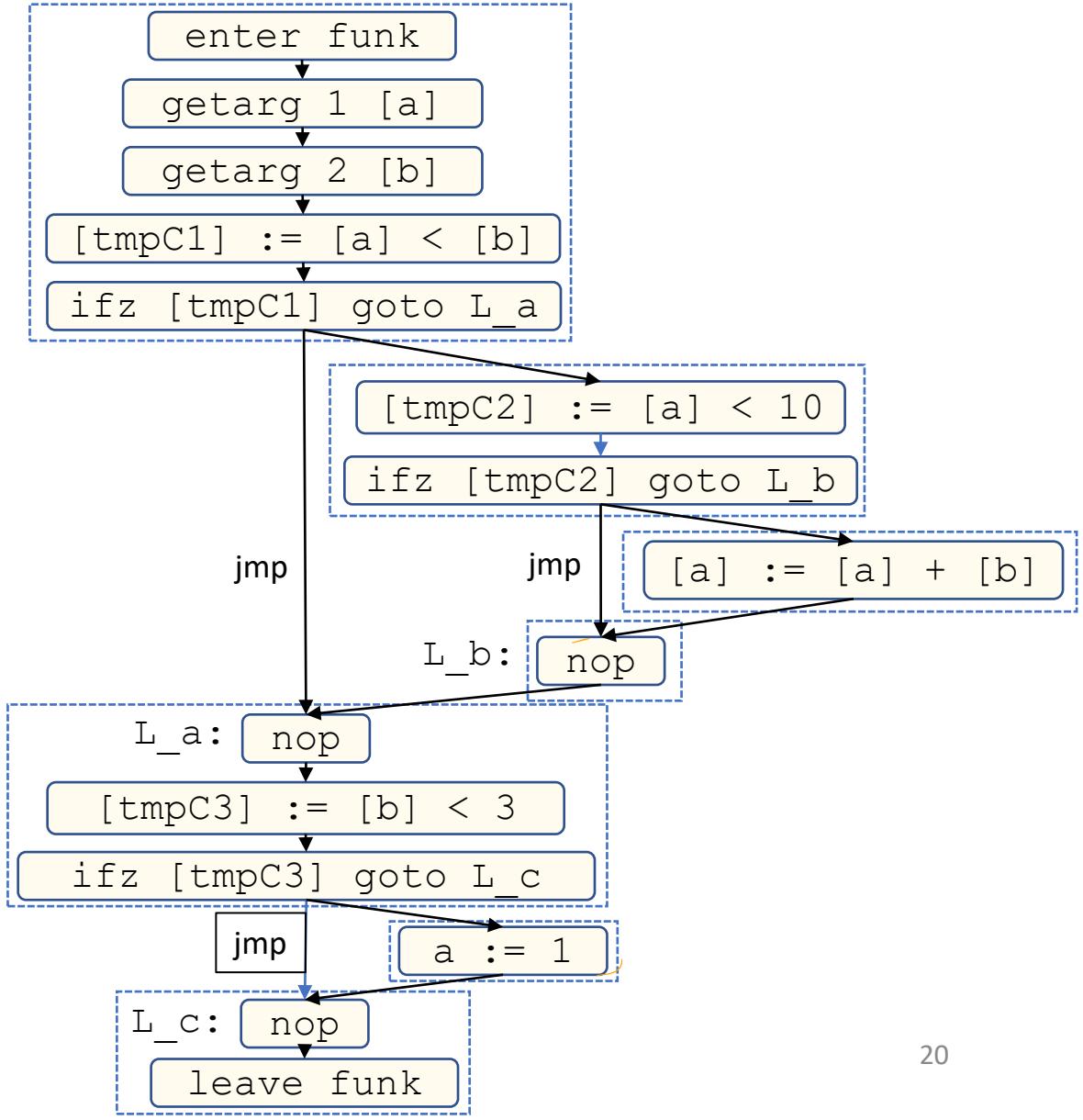


Compacting the Flowchart Concept

Flowgraphs: Control flow graphs

The flowchart is needlessly verbose

- We could put multiple instructions in a node
- Group the instructions that <u>always execute together</u>



Basic Blocks

Flowgraphs: Control flow graphs

- Definition: Sequence of instructions guaranteed to execute without interruption

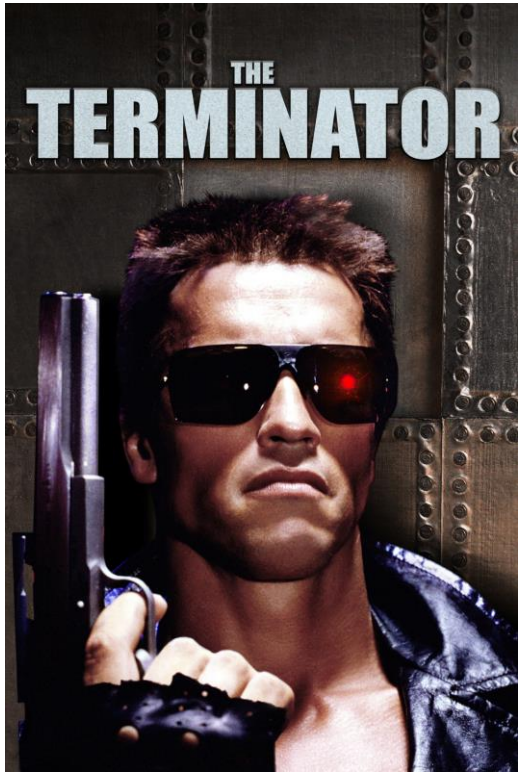




Basic Blocks Boundaries

Flowgraphs: Control flow graphs

- “Terminator” – An instruction that ends a basic block
- “Leader” – An instruction that begins a block



Basic Blocks

Flowgraphs: Control flow graphs

- Sequence of instructions guaranteed to execute without interruption
- Terminology:
  - "Leader" – An instruction that begins a block
    - *The first instruction in the procedure*
    - *The target of a jump*
    - *The instruction after an terminator*
  - "Terminator" – An instruction that ends a basic block
    - *The last instruction of the procedure*
    - *A jump (ifz, goto)*
    - *A call (We'll use a special LINK edge)*

Basic Blocks

Flowgraphs: Control flow graphs

Leaders

The first instruction in the procedure

The target of a jump

The instruction after a terminator

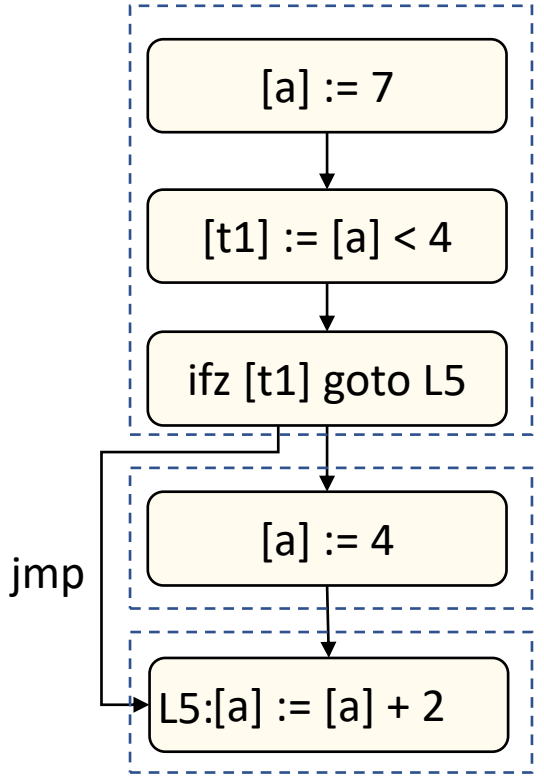
Terminators

The last instruction in the procedure

A jump (ifz, goto)

A call (We'll use a special LINK edge to successor)

Next instruction is a leader



Construction algorithm

```
foreach instr i in procedure:
   if i is a leader, begin a new BBL
   if i is a terminator, end current BBL
```

Building Basic Blocks

Flowgraphs: Control flow graphs

Leaders

The first instruction in the procedure

The target of a jump

The instruction after a terminator

Terminators

The last instruction in the procedure

A jump (ifz, goto)

A call (We'll use a special LINK edge to successor)

Next instruction is a leader



This algorithm isn't optimal, but we'll go with it

Construction algorithm

```
foreach instr i in procedure:
   if i is a leader, begin a new BBL
   if i is a terminator, end current BBL
```

example

```
      jmp L1
L1: nop
```

The Control Flow Graph: Summary

Flowgraphs: Control flow graphs

A graph of basic blocks

- One graph per procedure
  - Exactly one entry block
  - Exactly one exit block
- Distinguished edge types:
  - Back edges – an edge to a previously-encountered node
  - Call edge – Connects a call site to the called function
  - Link edge – Connects a function call to it's return point



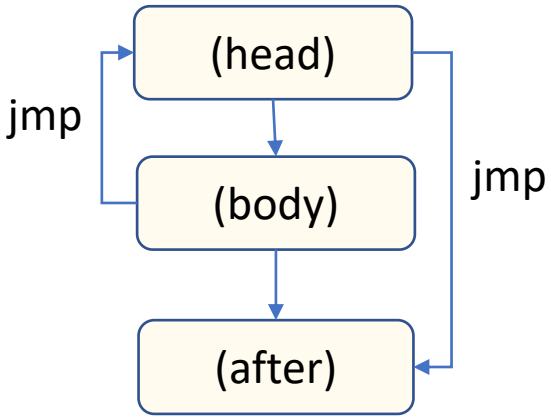
Benefits of Basic Blocks

Flowgraphs: Control flow graphs

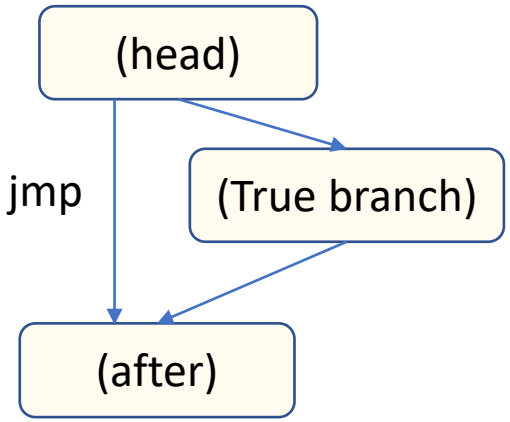
Makes CFGs a more manageable data structure

- Zoom out and observe procedure structure

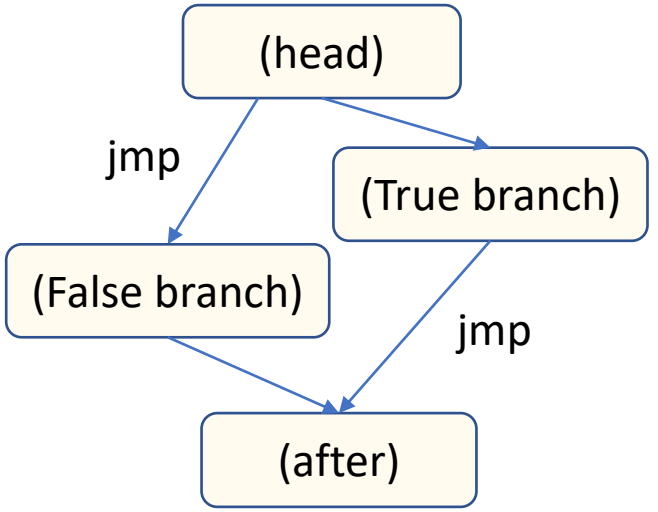
Loops



If-stmt



If-else



Benefits of Basic Blocks

Flowgraphs: Control flow graphs

Makes CFGs a more manageable data structure

- Zoom out and observe procedure structure
- Zoom in to a BBL's "uninterrupted sequences"

Simplifies analysis:

- Many properties we want to know are trivial to compute within a BBL

Types of CFG Analysis

Control Flow Graphs: Representation

Modularizes analysis:

- Analysis within a single basic block

  Traditionally called “Local” analysis

- Analysis between multiple basic blocks in a function

  Traditionally called “Global” analysis

- What about analysis between multiple functions?

  We’ll come back to this one

Lecture Outline

Flowgraphs

Program analysis:

- Goals
- Control flow graphs

Local Optimizations

- Dead code elimination
- Common subexpression elimination
- Constant/copy propagation



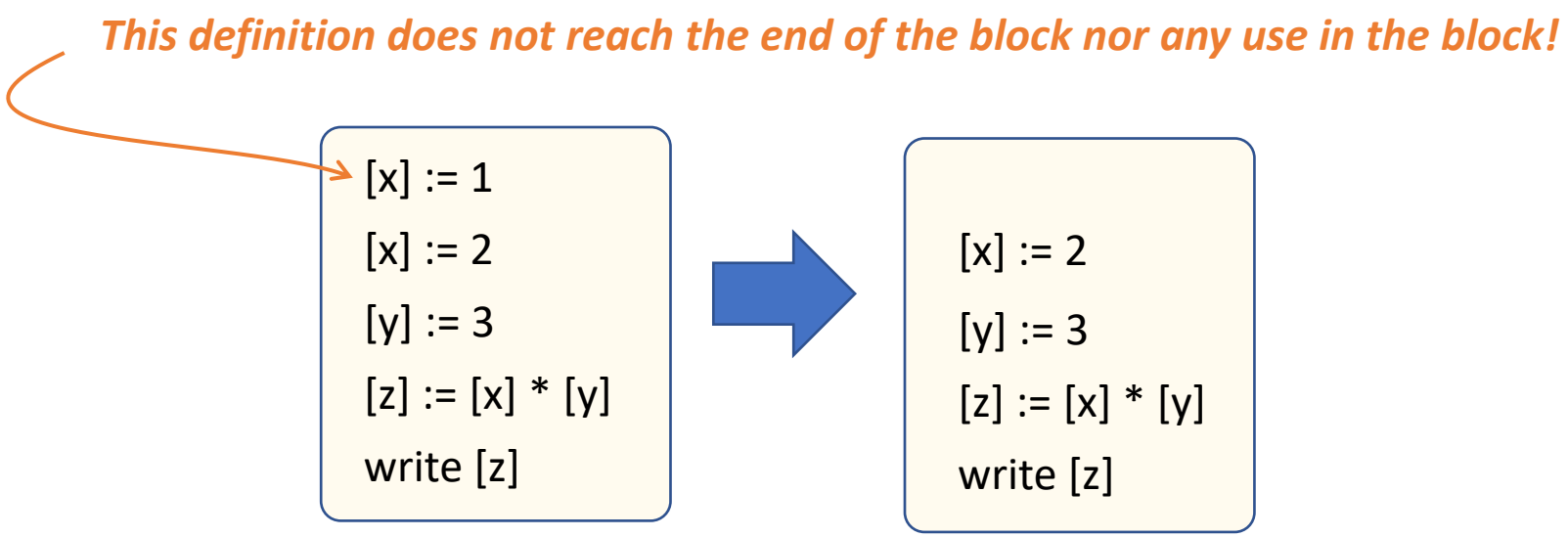
Optimization

Dead Code Elimination

Flowgraphs: Local Optimizations

Remove "useless" instructions (those with no effect)

- Analysis: live variable analysis

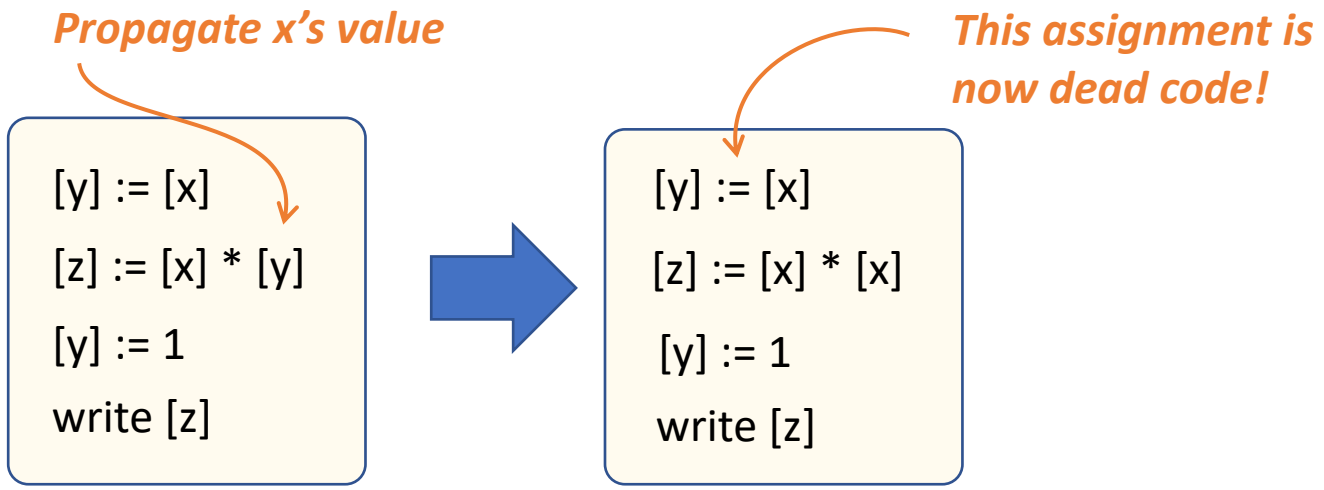


Constant/Copy Propagation

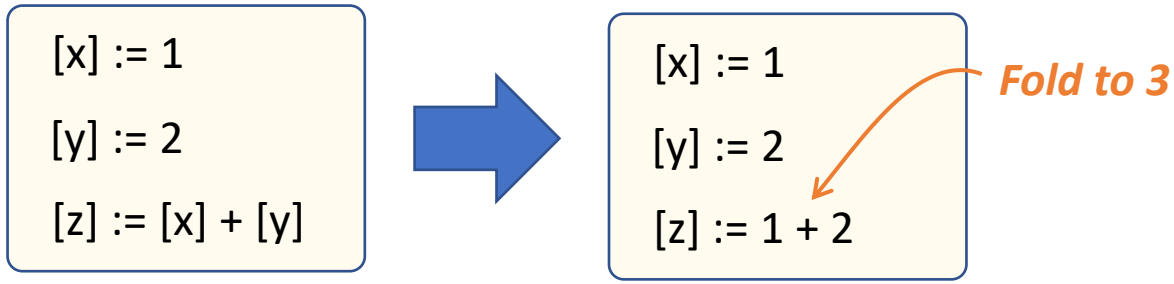
Flowgraphs: Local Optimizations

Replace a variable use with its definition

- Analysis: “copy identification” (doesn’t really have a name)



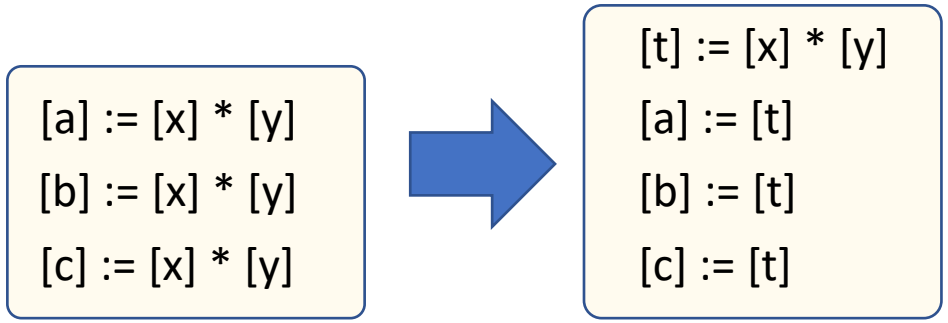
When propagating constant values, can aid in constant folding



Common Subexpression Elimination

Reuse already-computed expressions

- Analysis: available expression analysis



Lecture End

Flowgraphs

Summary

- Control Flow Graphs serve as an abstraction of the routes through the program
- Basic blocks summarize guaranteed sequences and enable local optimizations (DCE, CP, CSE)

Next Time

- Global optimizations – extending optimization across multiple basic blocks