Compilers

Optimization

Yannis Smaragdakis, U. Athens
(original slides by Sam Guyer@Tufts)

What we already saw

- Lowering

  From language-level constructs to machine-level constructs

- At this point we could generate machine code
  - Output of lowering is a correct translation
  - What's left to do?
    - Map from lower-level IR to actual ISA
    - Maybe some register management **(*could be required*)**
    - Pass off to assembler

- Why have a separate assembler?
  - Handles "packing the bits"

Assembly	**addi *<target>*, *<source>*, *<value>***
Machine	**0010 00*ss ssst tttt iiii iiii iiii iiii***

But first…

- The compiler "understands" the program
  - IR captures program semantics
  - Lowering: semantics-preserving transformation
  - Why not do others?

- Compiler optimizations
  - Oh great, now my program will be optimal!
  - Sorry, it's a misnomer
  - What is an "optimization"?

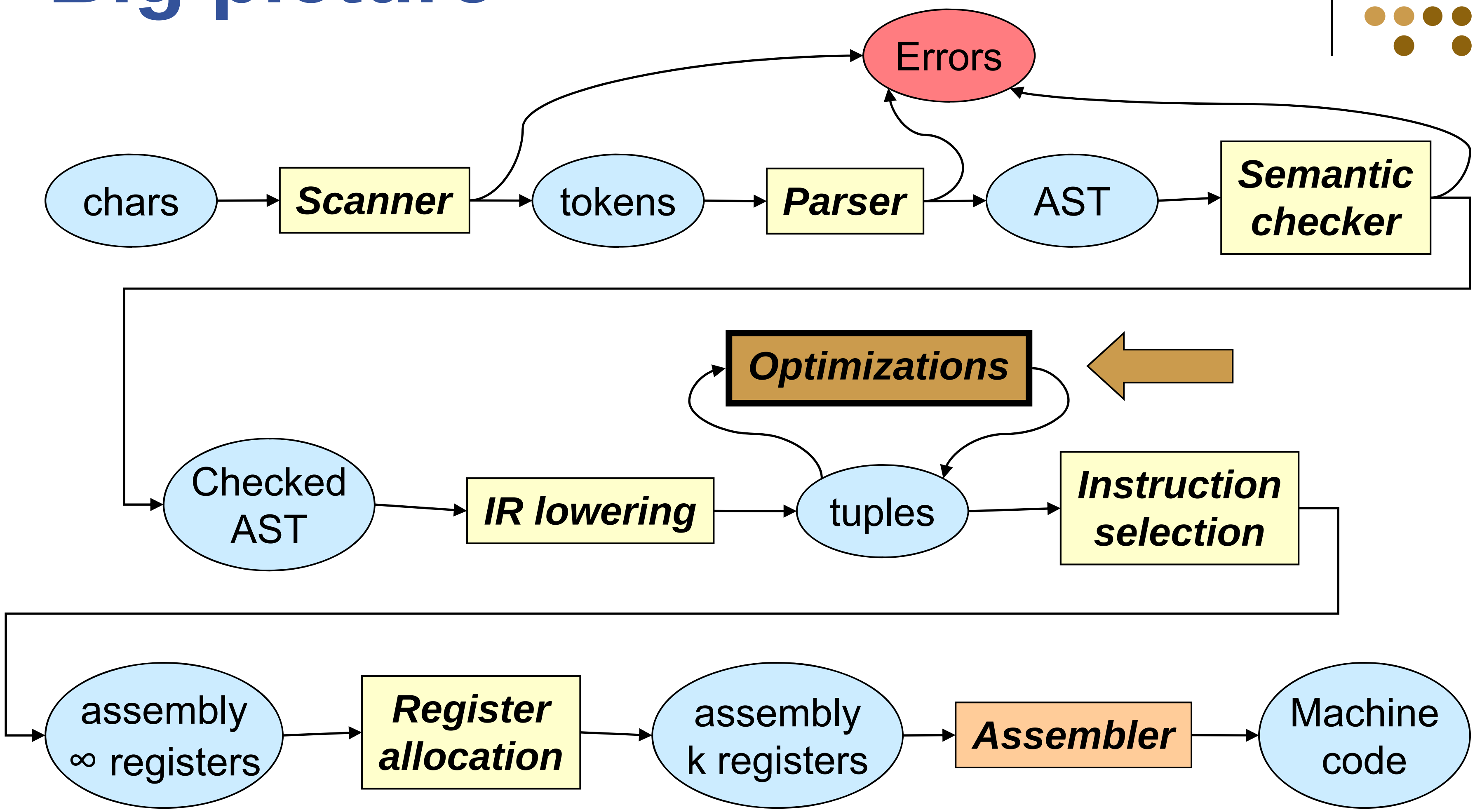
Optimizations

- What are they?
  - Code transformations
  - Improve some metric

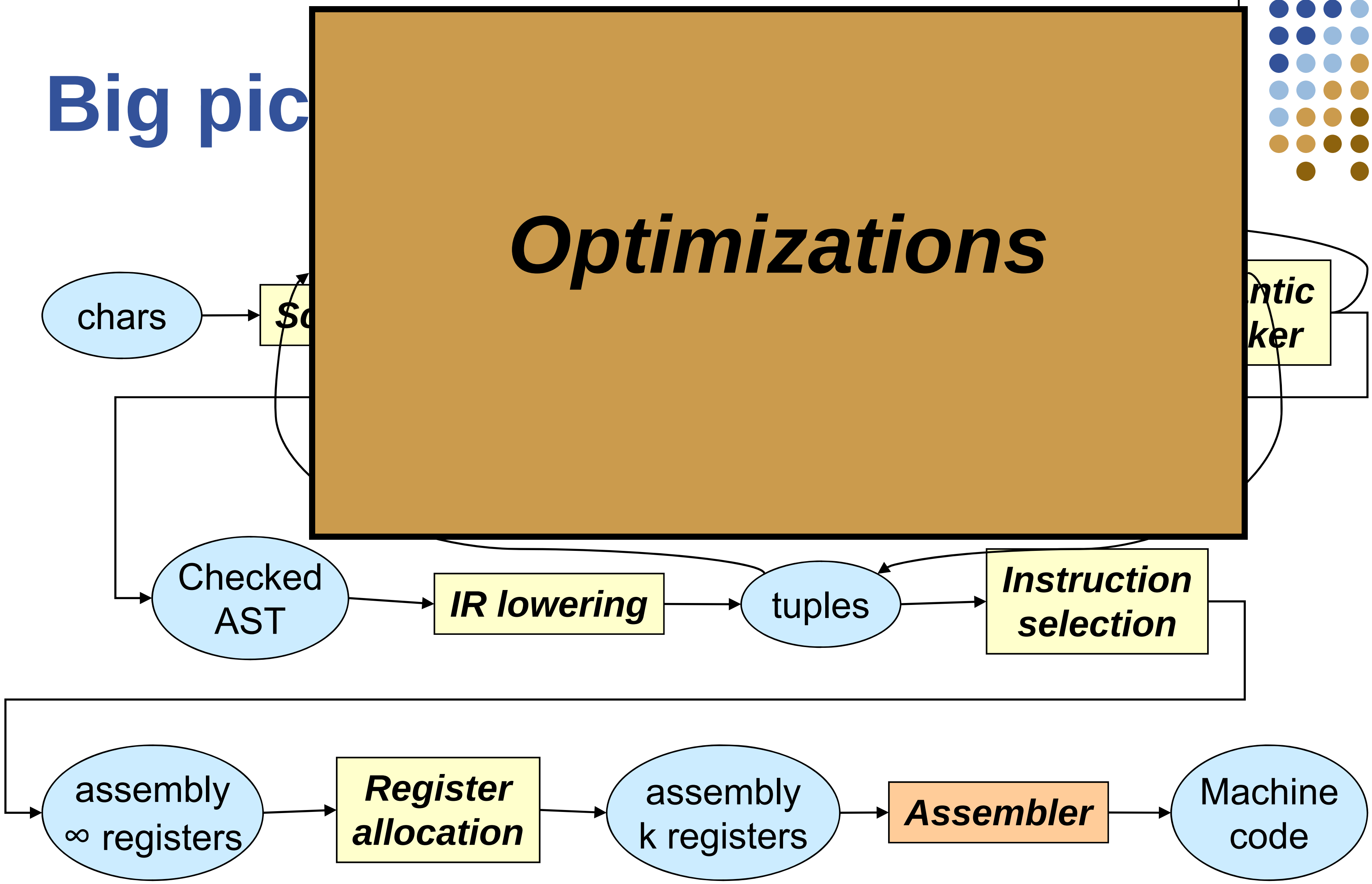
- Metrics
  - Performance: time, instructions, cycles

    Are these metrics equivalent?
  - Memory
    - Memory hierarchy (reduce cache misses)
    - Reduce memory usage
  - Code Size
  - Energy

Big picture


Errors
chars
Scanner
tokens
Parser
AST
Semantic checker
Optimizations
Checked AST
IR lowering
tuples
Instruction selection
assembly
∞ registers
Register allocation
assembly
k registers
Assembler
Machine code

Big pic



Optimizations
chars
Sc
ntic
ker
Checked AST
IR lowering
tuples
Instruction selection
assembly ∞ registers
Register allocation
assembly k registers
Assembler
Machine code

Why optimize?

- High-level constructs may make some optimizations difficult or impossible:

```
A[i][j] = A[i][j-1] + 1
```

```
t = A + i*row + j
s = A + i*row + j – 1
(*t) = (*s) + 1
```

- High-level code may be more desirable
  - Program at high level
  - Focus on design; clean, modular implementation
  - Let compiler worry about gory details
- Premature optimization is the root of all evil!

Limitations

- What are optimizers good at?
  - Consistent and thorough
  - Find all opportunities for an optimization
  - Uniformly apply the transformation
- What are they *not* good at?
  - Asymptotic complexity
  - Compilers can't fix bad algorithms
  - Compilers can't fix bad data structures
- There's no magic

Requirements

- Safety
  - Preserve the semantics of the program
  - What does that mean?
- Profitability
  - Will it help our metric?
  - Do we need a guarantee of improvement?
- Risk
  - How will interact with other optimizations?
  - How will it affect other stages of compilation?

Example: loop unrolling

```
for (i=0; i<100; i++)
  *t++ = *s++;
```

```
for (i=0; i<25; i++) {
  *t++ = *s++;
  *t++ = *s++;
  *t++ = *s++;
  *t++ = *s++;  }
```

- Safety:
  - Always safe; getting loop conditions right can be tricky.
- Profitability
  - Depends on hardware – usually a win – why?
- Risk?
  - Increases size of code in loop
  - May not fit in the instruction cache

Optimizations

- Many, many optimizations invented
  - *Constant folding, constant propagation, tail-call elimination, redundancy elimination, dead code elimination, loop-invariant code motion, loop splitting, loop fusion, strength reduction, array scalarization, inlining, cloning, data prefetching, parallelization. . .etc . .*
- How do they interact?
  - Optimist: we get the sum of all improvements!
  - Realist: many are in direct opposition

Rough categories

- Traditional optimizations
  - Transform the program to reduce work
  - Don’t change the level of abstraction
- Resource allocation
  - Map program to specific hardware properties
  - Register allocation
  - Instruction scheduling, parallelism
  - Data streaming, prefetching
- Enabling transformations
  - Don’t necessarily improve code on their own
  - Inlining, loop unrolling

Constant propagation

- **Idea**
  - If the value of a variable is known to be a constant at compile-time, replace the use of variable with constant

```
n = 10;
c = 2;
for (i=0;i<n;i++)
  s = s + i*c;
```

→

```
n = 10;
c = 2;
for (i=0;i<10;i++)
  s = s + i*2;
```

- Safety
  - Prove the value is constant
- **Notice**:
  - May interact favorably with other optimizations, like loop unrolling – now we know the *trip count*

Constant folding

- **Idea**
  - If operands are known at compile-time, evaluate expression at compile-time

```
r = 3.141 * 10;
```

→

```
r = 31.41;
```

  - What do we need to be careful of?
    - Is the result the same as if executed at runtime?
    - Overflow/underflow, rounding and numeric stability
- Often repeated throughout compiler

```
x = A[2];
```

→

```
t1 = 2*4;
t2 = A + t1;
x = *t2;
```

Partial evaluation

- Constant propagation and folding together
- **Idea**:
  - Evaluate as much of the program at compile-time as possible
  - More sophisticated schemes:
    - Simulate data structures, arrays
    - Symbolic execution of the code
- Caveat: floating point
  - Preserving the error characteristics of floating point values

Algebraic simplification

- **Idea**:
  - Apply the usual algebraic rules to simplify expressions

```
a * 1
a/1
a * 0
a + 0
b || false
```

⇒

```
a
a
0
a
b
```

- Repeatedly apply to complex expressions
- Many, many possible rules
  - Associativity and commutativity come into play

Common sub-expression elimination

- **Idea**:
  - If program computes the same expression multiple times, reuse the value.

```
a = b + c;
c = b + c;
d = b + c;
```

⇒

```
t = b + c
a = t;
c = t;
d = b + c;
```

- Safety:
  - Subexpression can only be reused until operands are redefined
- Often occurs in address computations
  - Array indexing and struct/field accesses

Dead code elimination

- **Idea**:
  - If the result of a computation is never used, then we can remove the computation

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

⇨

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

- Safety
  - Variable is dead if it is never used after defined
  - Remove code that assigns to dead variables
- This may, in turn, create more dead code
  - Dead-code elimination usually works transitively

Copy propagation

- **Idea**:
  - After an assignment x = y, replace any uses of x with y

```
x = y;
if (x>1)
  s = x+f(x);
```

⇒

```
x = y;
if (y>1)
  s = y+f(y);
```

- Safety:
  - Only apply up to another assignment to x, **or**
  - …another assignment to y!
- What if there was an assignment y = z earlier?
  - Apply transitively to all assignments

Unreachable code elimination

- **Idea**:
  - Eliminate code that can never be executed

```
#define DEBUG 0
. . .
if (DEBUG)
  print("Current value = ", v);
```

- Different implementations
  - High-level: look for if (false) or while (false)
  - Low-level: more difficult
    - Code is just labels and gotos
    - Traverse the graph, marking reachable blocks

How do these things happen?

- Who would write code with:
  - Dead code
  - Common subexpressions
  - Constant expressions
  - Copies of variables
- Two ways they occur
  - High-level constructs – we've already seen examples
  - Other optimizations
    - Copy propagation often leaves dead code
    - Enabling transformations: inlining, loop unrolling, etc.

Loop optimizations

- Program hot-spots are usually in loops
  - Most programs: 90% of execution time is in loops
  - What are possible exceptions?

    OS kernels, compilers and interpreters

- Loops are a good place to expend extra effort
  - Numerous loop optimizations
  - Often expensive – complex analysis
  - For languages like Fortran, very effective
  - What about C?

Loop-invariant code motion

- **Idea**:
  - If a computation won't change from one loop iteration to the next, move it outside the loop

```
for (i=0;i<N;i++)
  A[i] = A[i] + x*x;
```

⇒

```
t1 = x*x;
for (i=0;i<N;i++)
  A[i] = A[i] + t1;
```

- Safety:
  - Determine when expressions are invariant
  - Just check for variables with no assignments?
- Useful for array address computations
  - Not visible at source level

Strength reduction

- **Idea**:
  - Replace expensive operations (mutiplication, division) with cheaper ones (addition, subtraction, bit shift)
- Traditionally applied to ***induction variables***
  - Variables whose value depends linearly on loop count
  - Special analysis to find such variables

```
for (i=0;i<N;i++)
  v = 4*i;
  A[v] = . . .
```

⇒

```
v = 0;
for (i=0;i<N;i++)
  A[v] = . . .
  v = v + 4;
```

Strength reduction

- Can also be applied to simple arithmetic operations:

```
x * 2
x * 2^c
x/2^c
```

```
x + x
x<<c
x>>c
```

- Typical example of premature optimization
  - Programmers use bit-shift instead of multiplication
  - “x<<2” is harder to understand
  - Most compilers will get it right automatically

Inlining

- **Idea**:
  - Replace a function call with the body of the callee
- Safety
  - What about recursion?
- Risk
  - Code size
  - Most compilers use heuristics to decide when
  - Has been cast as a *knapsack problem*

Inlining

- What are the benefits of inlining?
  - Eliminate call/return overhead
  - Customize callee code in the context of the caller
    - Use actual arguments
    - Push into copy of callee code using constant prop
    - Apply other optimizations to reduce code
  - Hardware
    - Eliminate the two jumps
    - Keep the pipeline filled
- Critical for OO languages
  - Methods are often small
  - Encapsulation, modularity force code apart

Inlining

- In C:
  - At a call-site, decide whether or not to inline
    - (Often a heuristic about callee/caller size)
  - Look up the callee
  - Replace call with body of callee
- What about Java?
  - What complicates this?
  - Virtual methods
  - Even worse?
  - Dynamic class loading

```
class A { void M() {…} }
class B extends A
        { void M() {…} }

void foo(A x)
{
  x.M(); // which M?
}
```

Inlining in Java

- With guards:

```
void foo(A x)
{
  if (x is type A)
    x.M(); // inline A's M
  if (x is type B)
    x.M(); // inline B's M
}
```

- Specialization
  - At a given call, we may be able to determine the type
  - Requires fancy analysis

```
y = new A();
foo(y);
z = new B();
foo(z);
```

Big picture

- When do we apply these optimizations?
  - High-level:
    - Inlining, cloning
    - Some algebraic simplifications
  - Low-level
    - Everything else
- It's a black art
  - Ordering is often arbitrary
  - Many compilers just repeat the optimization passes over and over

Writing fast programs

In practice:

- Pick the right algorithms and data structures
  - Asymptotic complexity and constants
  - Memory usage, memory layout, data representation
- Turn on optimization and profile
  - Run-time
  - Program counters (e.g., cache misses)
- Evaluate problems
- Tweak source code
  - Help the optimizer do “the right thing”

Anatomy of an optimization

Two big parts:

- Program analysis

  Pass over code to find:
  - Opportunities
  - Check safety constraints

- Program transformation
  - Change the code to exploit opportunity

- Often: rinse and repeat

Dead code elimination

- **Idea**:
  - Remove a computation if result is never used

```
y = w - 7;
x = y + 1;
y = 1;
x = 2 * z;
```

→

```
y = w - 7;

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

→

```


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

- Safety
  - Variable is dead if it is never used after defined
  - Remove code that assigns to dead variables
- This may, in turn, create more dead code
  - Dead-code elimination usually works transitively

Dead code

- Another example:

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

Which statements can be safely removed?

- Conditions:
  - Computations whose value is never used
  - Obvious for straight-line code
  - What about control flow?

Dead code

- With if-then-else:

Which statements are can be removed?

```
x = y + 1;
y = 2 * z;
if (c) x = y + z;
z = 1;
z = x;
```

- Which statements are dead code?
  - What if “c” is false?
  - Dead only on some paths through the code

Dead code

- And a loop:

Which statements are can be removed?

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

- Now which statements are dead code?

Dead code

- And a loop:

Which statements are can be removed?

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

- Statement “x = y+1” not dead
- What about “z = 1”?

Low-level IR

- Most optimizations performed in low-level IR
  - Labels and jumps
  - No explicit loops
- Even harder to see possible paths

```
label1:
jumpifnot p label2
x = y + 1
y = 2 * z
jumpifnot c label3
x = y + z
label3:
z = 1
jump label1
label2:
z = x
```

Optimizations and control flow

- Dead code is ***flow sensitive***
  - Not obvious from program

    Dead code example: are there any possible paths that make use of the value?
  - Must characterize all possible dynamic behavior
  - Must verify conditions at compile-time
- Control flow makes it hard to extract information
  - High-level: different kinds of control structures
  - Low-level: control-flow hard to infer
- Need a unifying data structure

Control flow graph

- ***Control flow graph*** (CFG):

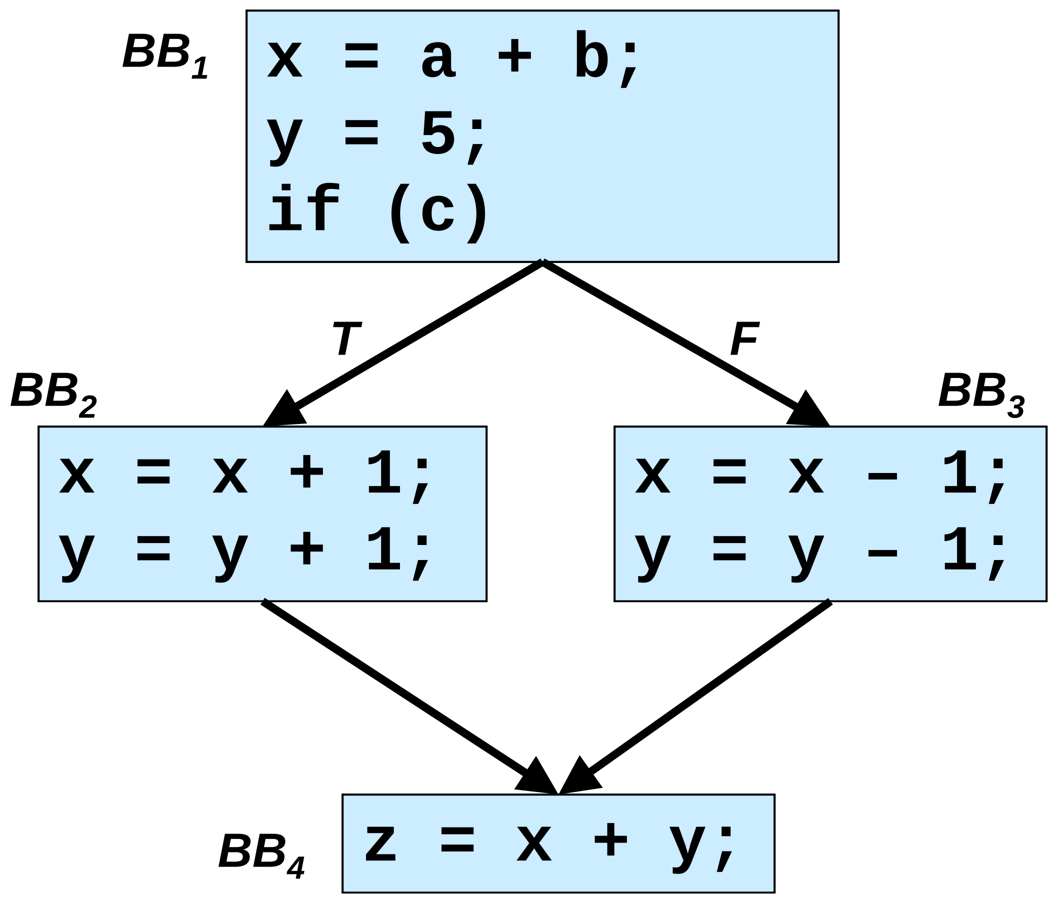
  a graph representation of the program
  - Includes both computation and control flow
  - Easy to check control flow properties
  - Provides a framework for global optimizations and other compiler passes
- Nodes are ***basic blocks***
  - Consecutive sequences of non-branching statements
- Edges represent control flow
  - From jump to a label
  - Each block may have multiple incoming/outgoing edges

CFG Example

Program

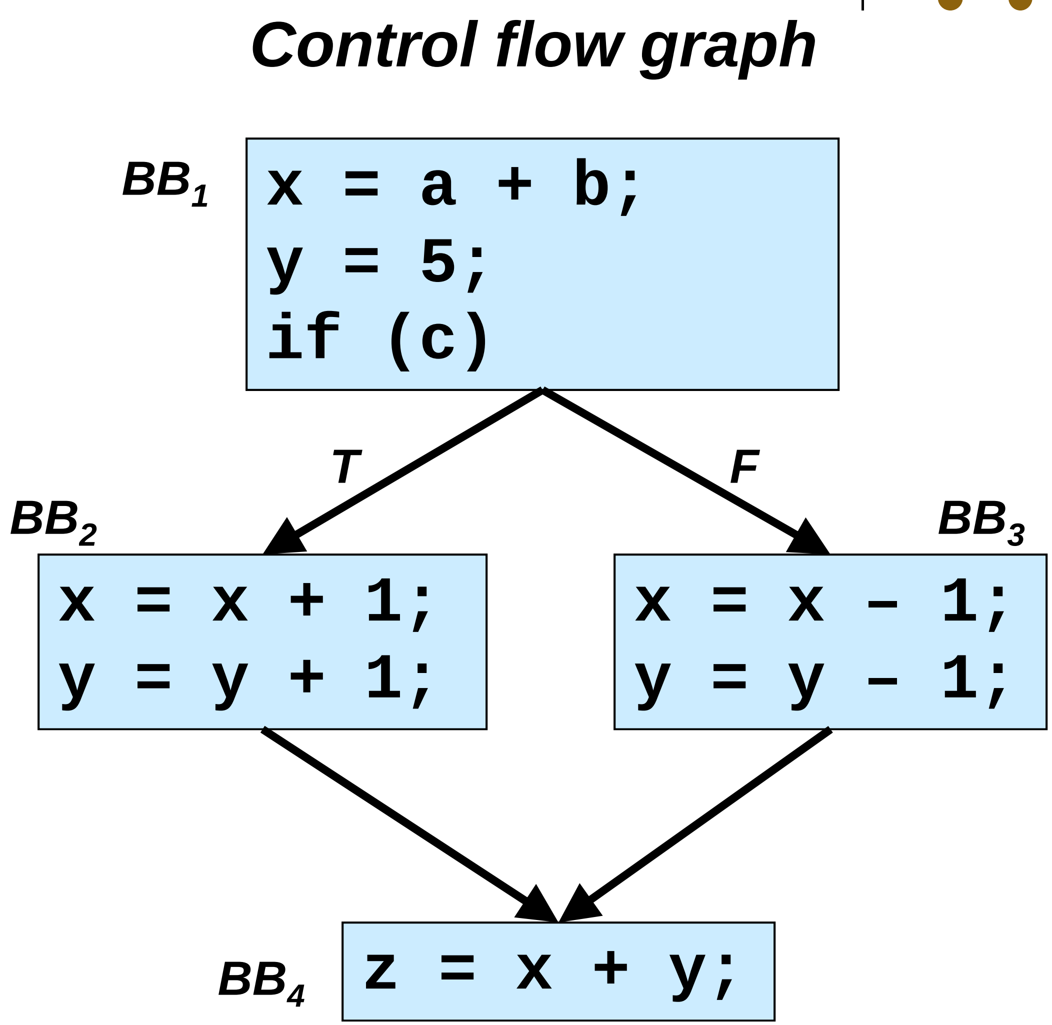
```
x = a + b;
y = 5;
if (c) {
  x = x + 1;
  y = y + 1;
} else {
  x = x – 1;
  y = y – 1;
}
z = x + y;
```

Control flow graph



Multiple program executions

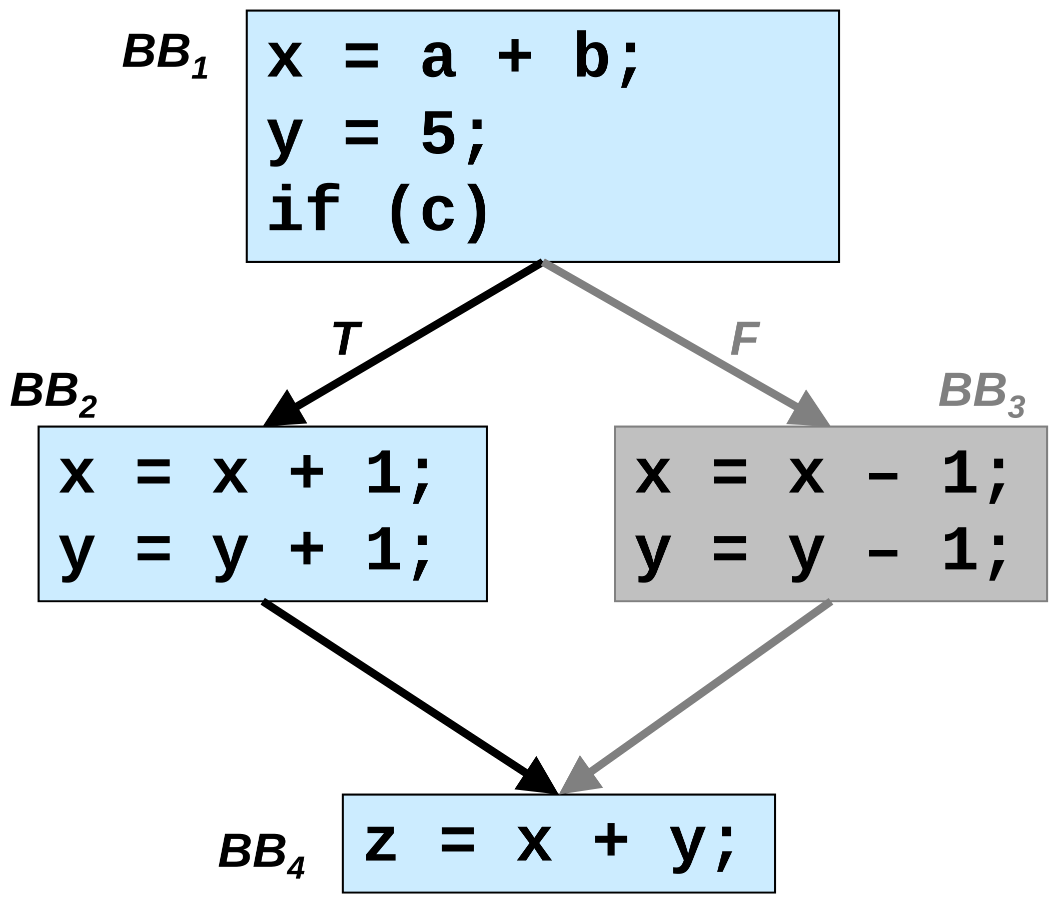
- CFG models all program executions
- An actual execution is a path through the graph
- Multiple paths: multiple possible executions
  - How many?



Execution 1

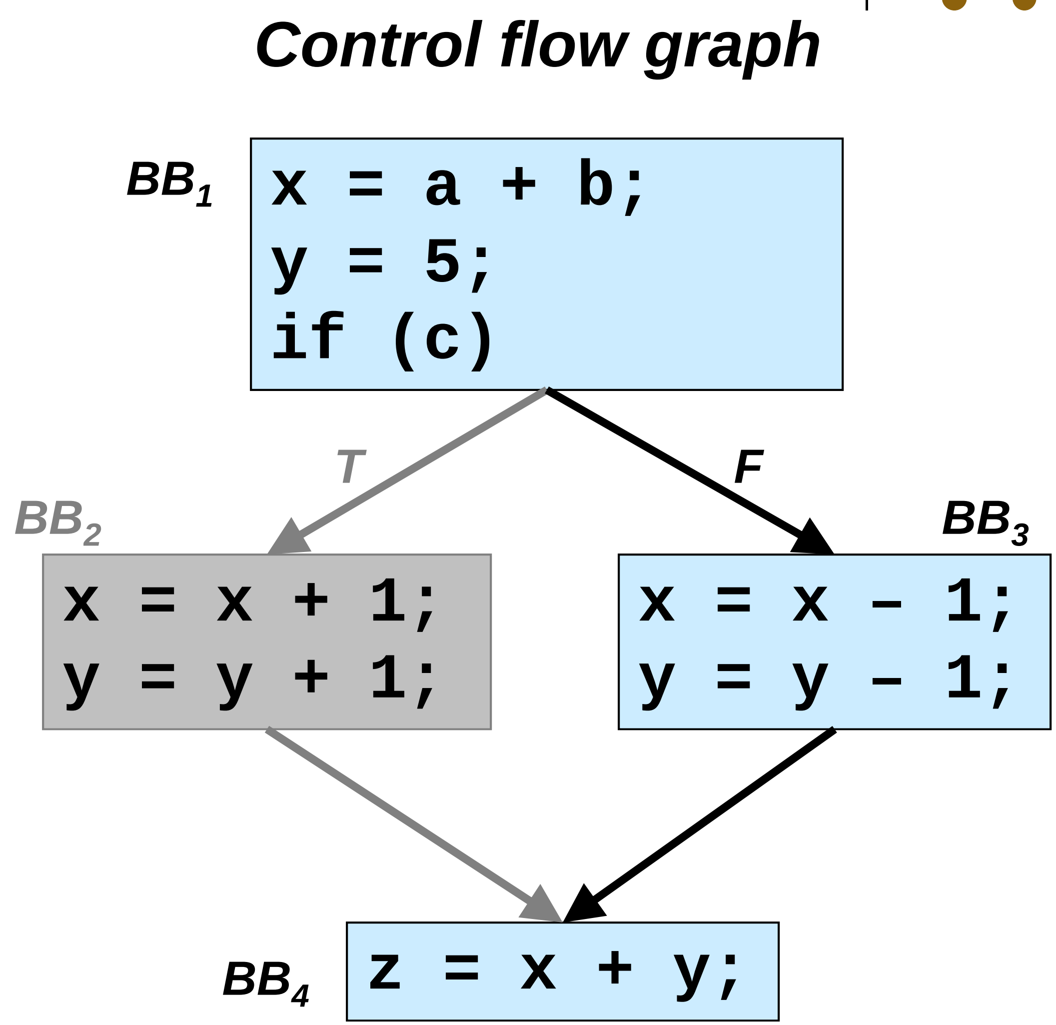
- CFG models all program executions
- Execution 1:
  - c is true
  - Program executes BB_1, BB_2, and BB_4

Control flow graph



Execution 2

- CFG models all program executions
- Execution 2:
  - c is false
  - Program executes BB_1, BB_3, and BB_4



Basic blocks

- **Idea**:
  - Once execution enters the sequence, all statements (or instructions) are executed
  - Single-entry, single-exit region
- Details
  - Starts with a label
  - Ends with one or more branches
  - Edges may be labeled with predicates

    May include special categories of edges
    - Exception jumps
    - Fall-through edges
    - Computed jumps (jump tables)

Building the CFG

- Two passes
  - First, group instructions into basic blocks
  - Second, analyze jumps and labels

- How to identify basic blocks?
  - Non-branching instructions

    Control cannot flow out of a basic block without a jump
  - Non-label instruction

    Control cannot enter the middle of a block without a label

Basic blocks

- Basic block starts:
  - At a label
  - After a jump
- Basic block ends:
  - At a jump
  - Before a label

```
label1:
jumpifnot p label2
x = y + 1
y = 2 * z
jumpifnot c label3
x = y + z
label3:
z = 1
jump label1
label2:
z = x
```

Basic blocks

- Basic block starts:
  - At a label
  - After a jump
- Basic block ends:
  - At a jump
  - Before a label
- **Note**: order still matters

```
label1:
jumpifnot p label2
```

```
x = y + 1
y = 2 * z
jumpifnot c label3
```

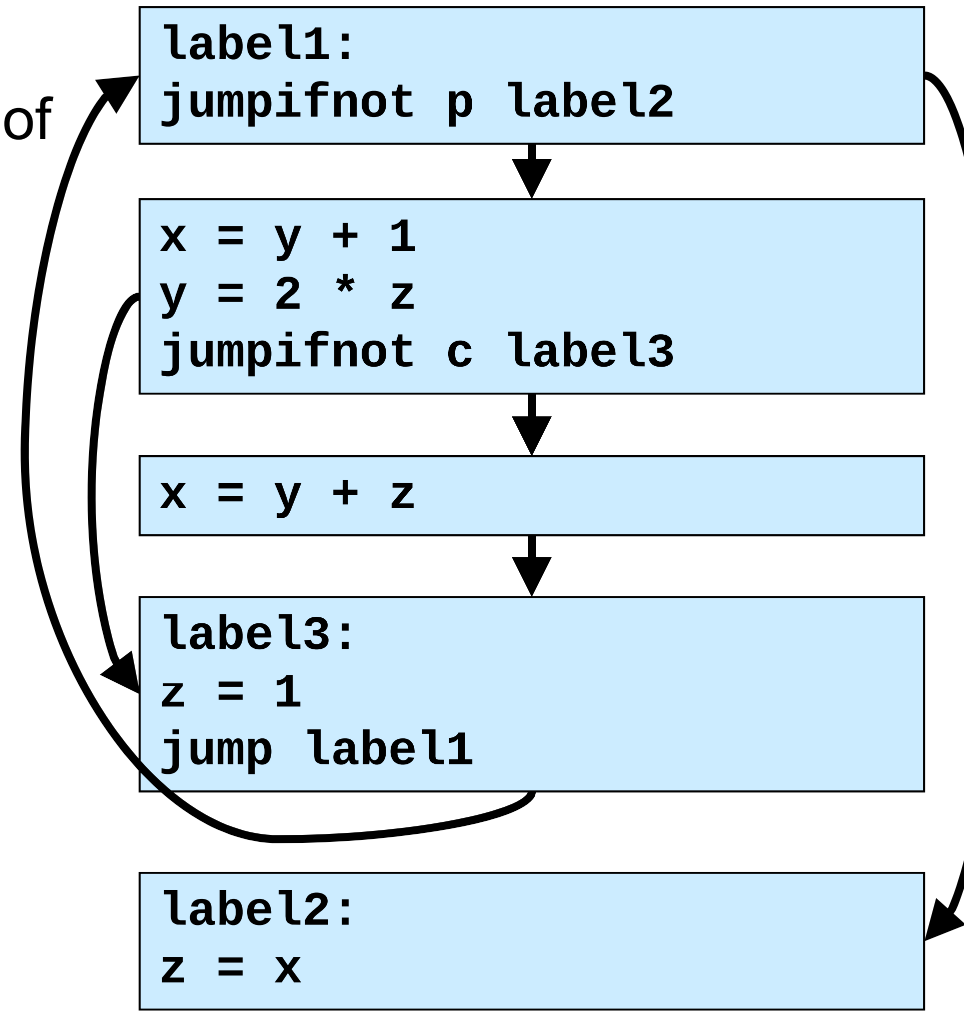
```
x = y + z
```

```
label3:
z = 1
jump label1
```

```
label2:
z = x
```

Add edges

- Unconditional jump
  - Add edge from source of jump to the block containing the label
- Conditional jump
  - 2 successors
  - One may be the fall-through block
- Fall-through



Two CFGs

- From the high-level
  - Break down the complex constructs
  - Stop at sequences of non-control-flow statements
  - Requires special handling of break, continue, goto
- From the low-level
  - Start with lowered IR – tuples, or 3-address ops
  - Build up the graph
  - More general algorithm
  - Most compilers use this approach
- Should lead to roughly the same graph

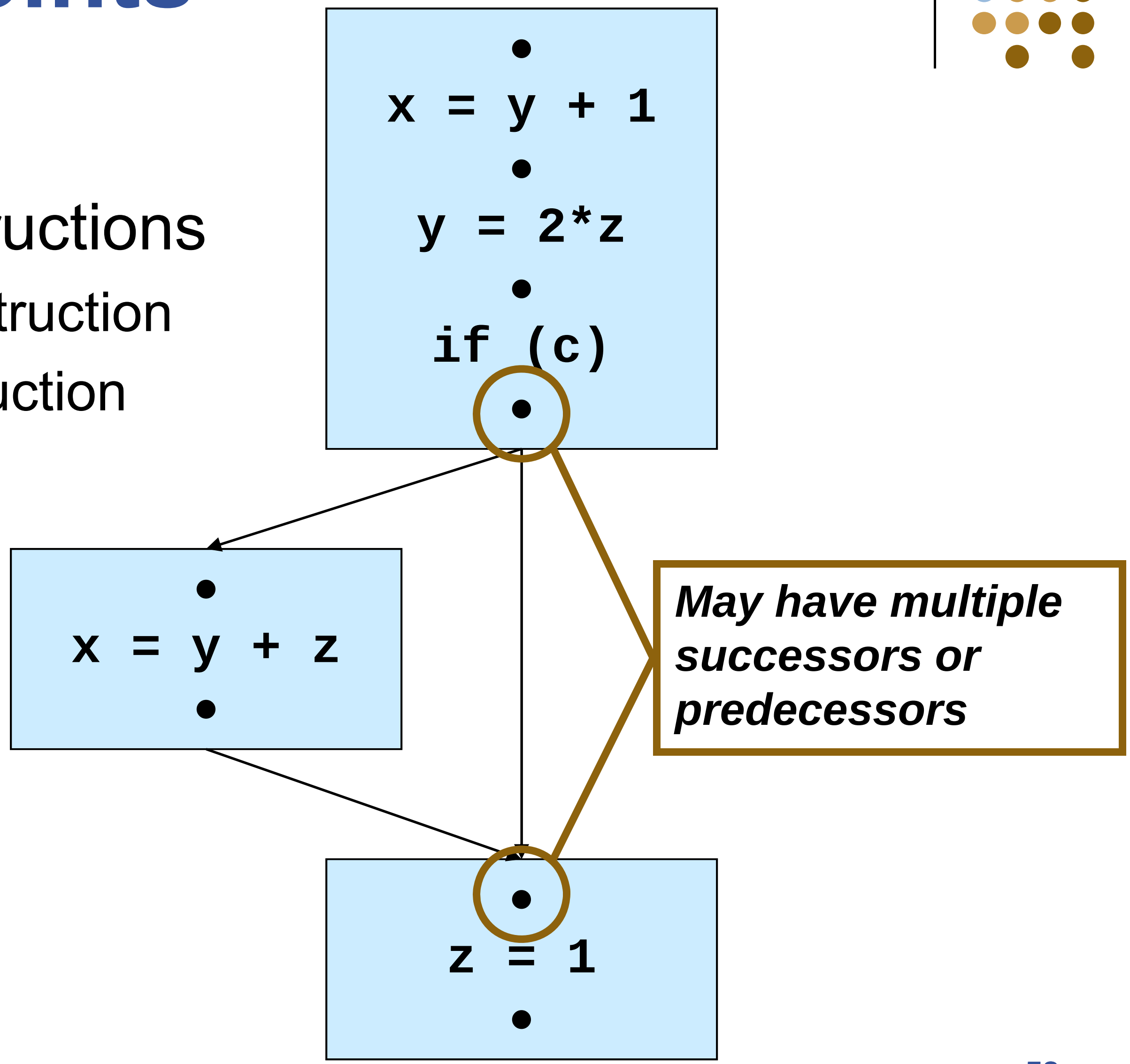
Using the CFG

- Uniform representation for program behavior
  - Shows all possible program behavior
  - Each execution represented as a path
  - Can reason about potential behavior
    Which paths can happen, which can't
  - Possible paths imply possible values of variables

- Example: ***liveness*** information
- **Idea**:
  - Define program points in CFG
  - Describe how information flows between points

Program points

- In between instructions
  - Before each instruction
  - After each instruction



Live variables analysis

- **Idea**
  - Determine ***live range*** of a variable

    Region of the code between when the variable is assigned and when its value is used
  - Specifically:

    Def: A variable v is live at point p if
    - There is a path through the CFG from p to a use of v
    - There are no assignments to v along the path

  ⇨ Compute a set of live variables at each point p

- Uses of live variables:
  - Dead-code elimination – find unused computations
  - Also: register allocation, garbage collection

Computing live variables

- How do we compute live variables?

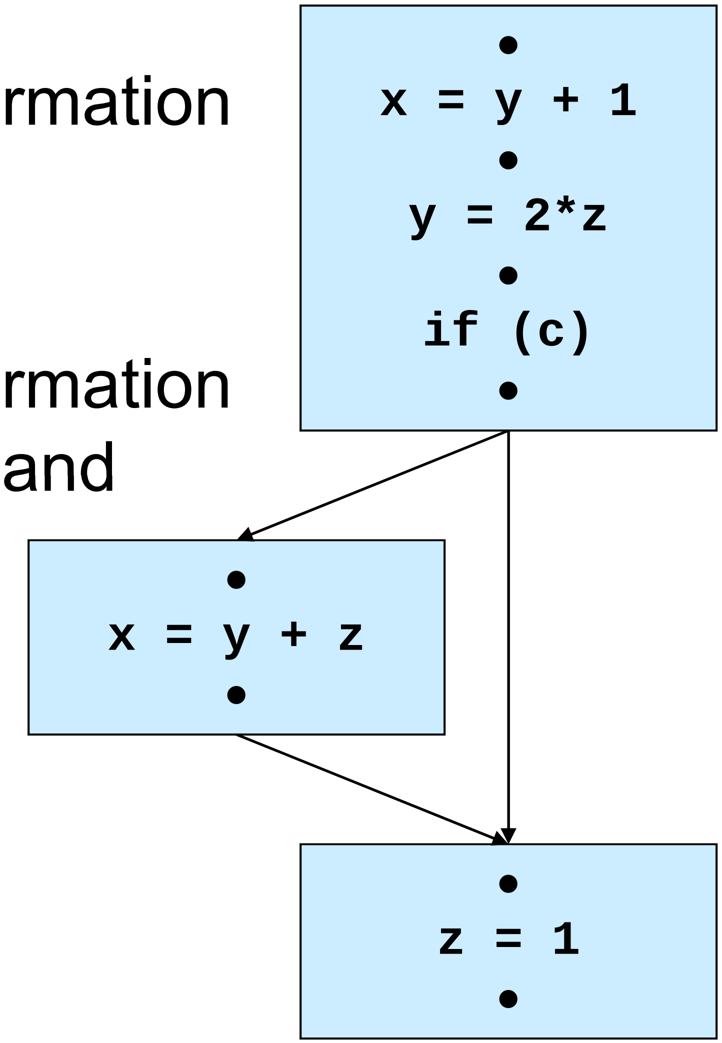
  (Specifically, a set of live variables at each program point)

- What is a straight-forward algorithm?
  - Start at uses of v, search backward through the CFG
  - Add v to live variable set for each point visited
  - Stop when we hit assignment to v

- Can we do better?
  - Can we compute liveness for all variables at the same time?
  - **Idea**:
    - Maintain a set of live variables
    - Push set through the CFG, updating it at each instruction

Flow of information

- ***Question 1***: how does information flow across instructions?
- ***Question 2***: how does information flow between predecessor and successor blocks?



Live variables analysis

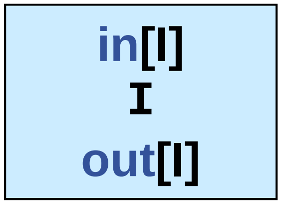
- At each program point:

  Which variables contain values computed earlier and needed later

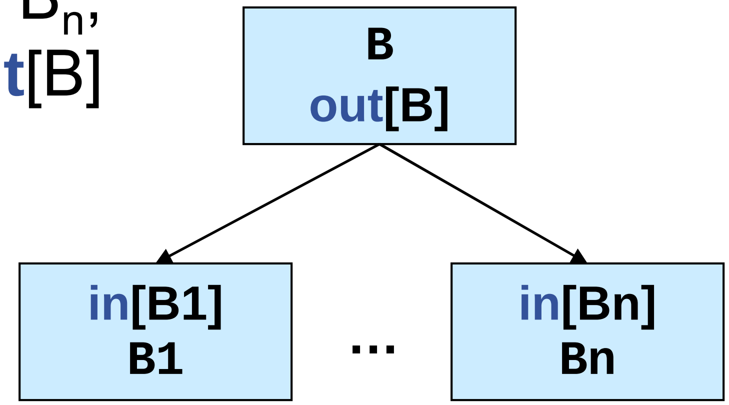
- For instruction I:
  - **in**[I] : live variables at program point before I
  - **out**[I] : live variables at program point after I
- For a basic block B:
  - **in**[B] : live variables at beginning of B
  - **out**[B] : live variables at end of B
- **Note**: **in**[I] = **in**[B] for first instruction of B
  out[I] = **out**[B] for last instruction of B

Computing liveness

- ***Answer question 1***: for each instruction I, what is relation between **in**[I] and **out**[I]?



- ***Answer question 2***: for each basic block B, with successors B_1, …, B_n, what is relationship between **out**[B] and **in**[B_1] … **in**[B_n]



Part 1: Analyze instructions

- Live variables across instructions
- Examples:

in[I] = {y,z}	in[I] = {y,z,t}	in[I] = {x,t}
`x = y + z`	`x = y + z`	`x = x + 1`
out[I] = {x}	out[I] = {x,t,y}	out[I] = {x,t}

- Is there a general rule?

Liveness across instructions

- How is liveness determined?
  - All variables that I uses are live before I

    *Called the **uses** of I*
  - All variables live after I are also live before I, unless I writes to them

    *Called the **defs** of I*
- Mathematically:

in[I] = {b}

a = b + 2

in[I] = {y,z}

x = 5

out[I] = {x,y,z}

$$in[I] = (out[I] - def[I]) \cup use[I]$$

Example

- Single basic block
(obviously: **out**[I] = **in**[succ(I)])
  - Live1 = in[B] = in[I1]
  - Live2 = out[I1] = in[I2]
  - Live3 = out[I2] = in[I3]
  - Live4 = out[I3] = out[B]

- Relation between live sets
  - Live1 = (Live2 – {x}) ∪ {y}
  - Live2 = (Live3 – {y}) ∪ {z}
  - Live3 = (Live4 – {}) ∪ {d}

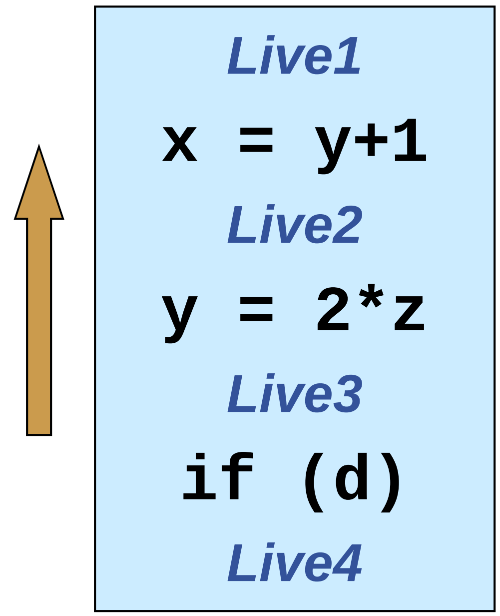
Live1
```
x = y+1
```
Live2
```
y = 2*z
```
Live3
```
if (d)
```
Live4

Flow of information

- Equation:

$$\textbf{in}[I] = (\ \textbf{out}[I] - \textbf{def}[I]\ ) \cup \textbf{use}[I]$$

- Notice: information flows ***backwards***
  - Need out[] sets to compute in[] sets
  - Propagate information up

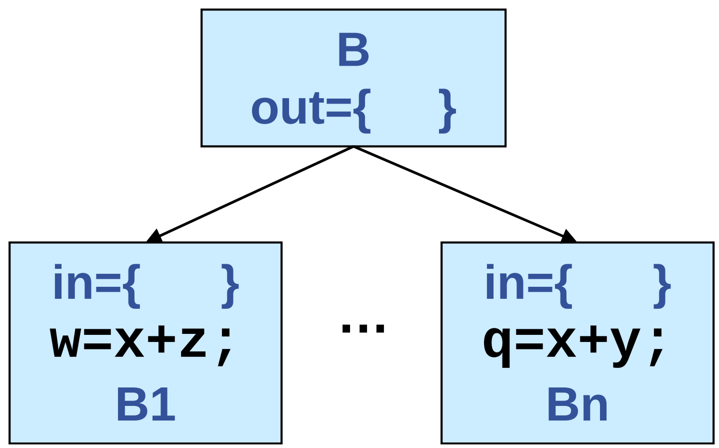


- Many problems are ***forward***

  Common sub-expressions, constant propagation, others

Part 2: Analyze control flow

- ***Question 2***: for each basic block B, with successors B_1, …, B_n, what is relationship between **out**[B] and **in**[B_1] … **in**[B_n]
- Example:



- What's the general rule?

Control flow

- Rule: A variable is live at end of block B if it is live at the beginning of ***any*** of the successors
  - Characterizes all possible executions
  - ***Conservative***: some paths may not actually happen

- Mathematically:

$$\textbf{out}[B] = \bigcup_{B' \in \text{succ}(B)} \textbf{in}[B']$$

- Again: information flows backwards

System of equations

- Put parts together:

$$\mathbf{in}[I] = (\ \mathbf{out}[I] - \mathbf{def}[I]\ ) \cup \mathbf{use}[I]$$

$$\mathbf{out}[I] = \mathbf{in}[succ(I)]$$

$$\mathbf{out}[B] = \bigcup_{B' \in succ(B)} \mathbf{in}[B']$$

Often called a system of ***Dataflow Equations***

- Defines a system of equations (or constraints)
  - Consider equation instances for each instruction and each basic block
  - What happens with loops?
    - Circular dependences in the constraints
    - Is that a problem?

Solving the problem

- Iterative solution:
  - Start with empty sets of live variables
  - Iteratively apply constraints
  - Stop when we reach a ***fixpoint***

For all instructions in[I] = out[I] = ∅

Repeat

For each instruction I

in[I] = (**out**[I] – **def**[I]) ∪ **use**[I]

out[I] = **in**[succ(I)]

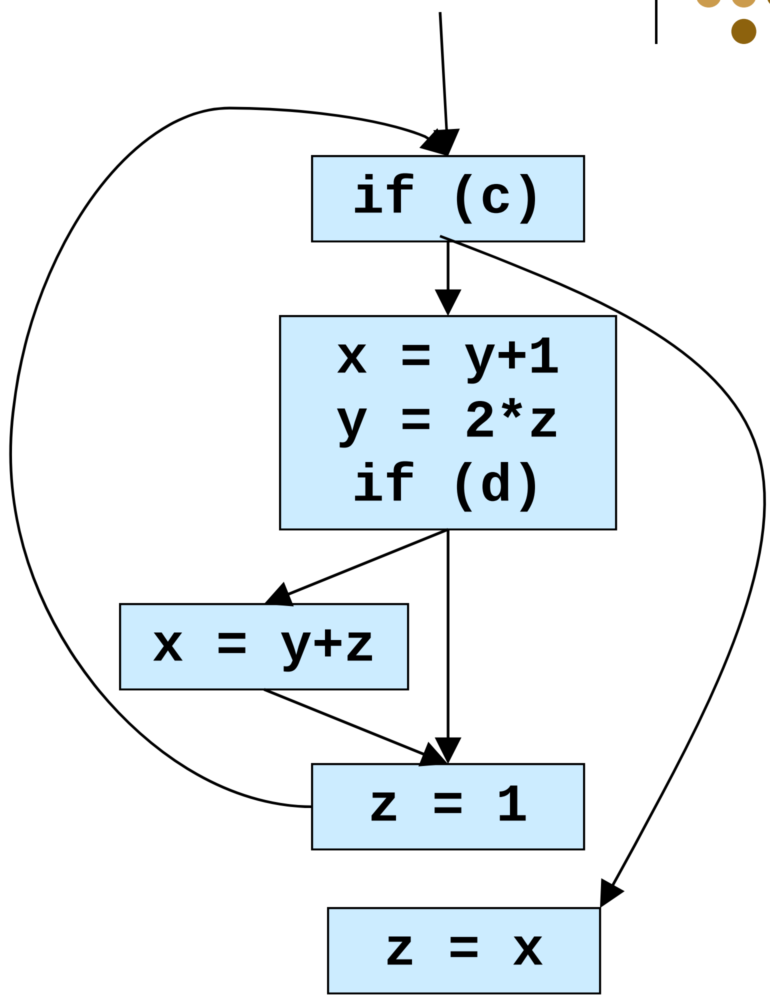
For each basic block B

$$out[B] = \bigcup_{B' \in succ(B)} in[B']$$

Until no new changes in sets

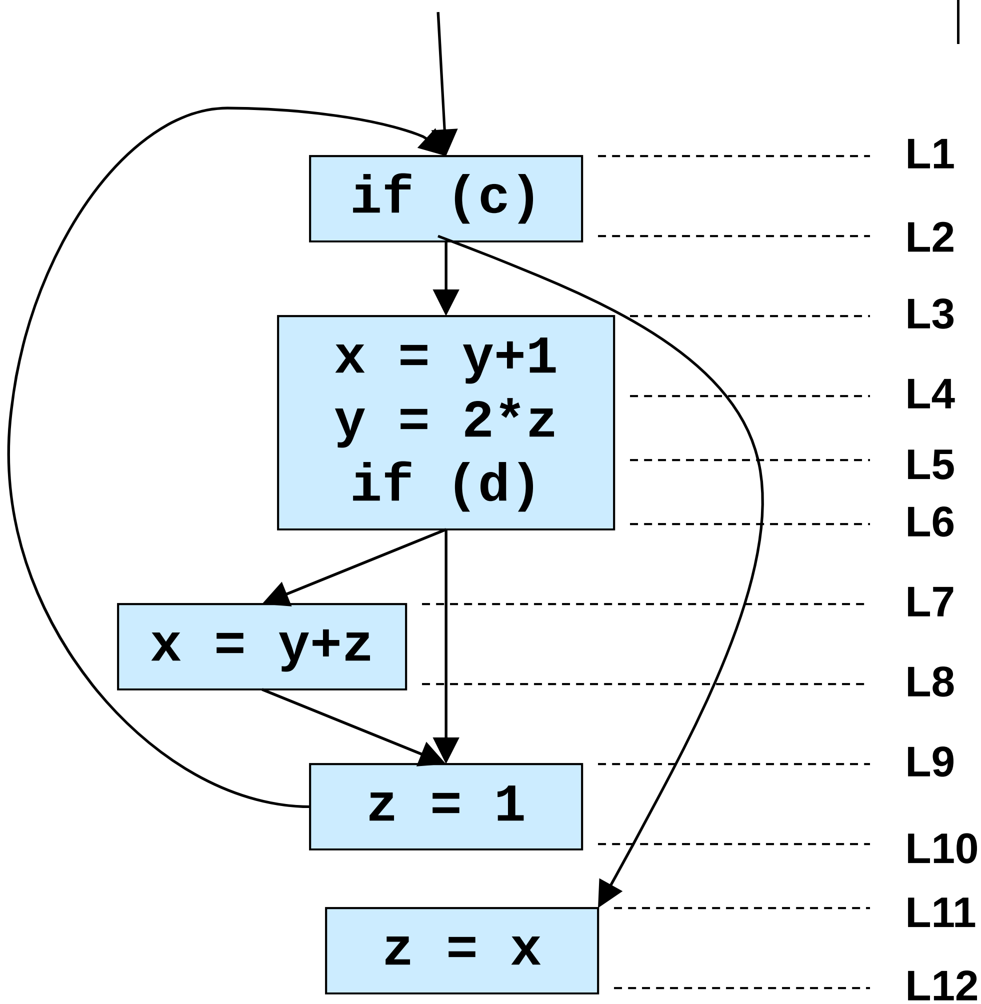
Example

- Steps:
  - Set up live sets for each program point
  - Instantiate equations
  - Solve equations



Example

- Program points



Example

$L1 = L2 \cup \{c\}$

$L2 = L3 \cup L11$

$L3 = (L4 - \{x\}) \cup \{y\}$

$L4 = (L5 - \{y\}) \cup \{z\}$

$L5 = L6 \cup \{d\}$

$L6 = L7 \cup L9$

$L7 = (L8 - \{x\}) \cup \{y,z\}$

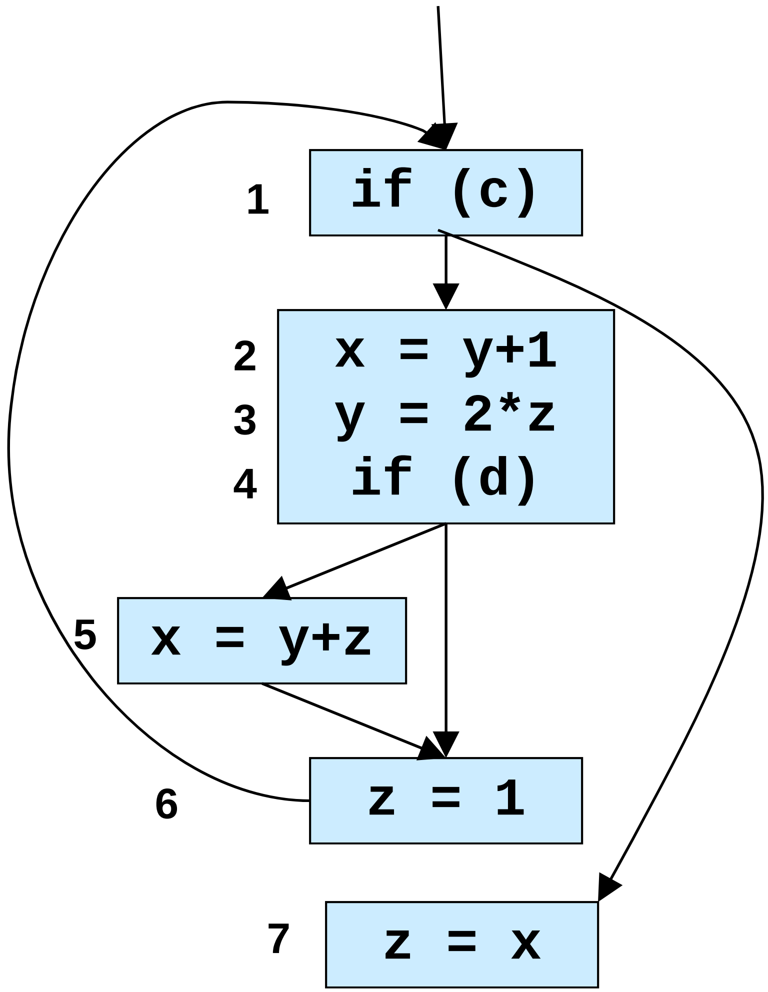
$L8 = L9$

$L9 = L10 - \{z\}$

$L10 = L1$

$L11 = (L12 - \{z\}) \cup \{x\}$

$L12 = \{\}$



$L1 = \{ x, y, z, c, d \}$

$L2 = \{ x, y, z, c, d \}$

$L3 = \{ y, z, c, d \}$

$L4 = \{ x, z, c, d \}$

$L5 = \{ x, y, z, c, d \}$

$L6 = \{ x, y, z, c, d \}$

$L7 = \{ y, z, c, d \}$

$L8 = \{ x, y, c, d \}$

$L9 = \{ x, y, c, d \}$

$L10 = \{ x, y, z, c, d \}$

$L11 = \{ x \}$

$L12 = \{ \}$

Questions

- Does this terminate?
- Does this compute the right answer?
- How could generalize this scheme for other kinds of analysis?

Generalization

- Dataflow analysis
  - A common framework for such analysis
  - Computes information at each program point
  - Conservative: characterizes all possible program behaviors

- Methodology
  - Describe the information (e.g., live variable sets) using a structure called a ***lattice***
  - Build a system of equations based on:
    - How each statement affects information
    - How information flows between basic blocks
  - Solve the system of constraints

Parts of live variables analysis

- Live variable sets
  - Called ***flow values***
  - Associated with program points
  - Start "empty", eventually contain solution
- Effects of instructions
  - Called ***transfer functions***
  - Take a flow value, compute a new flow value that captures the effects
  - One for each instruction – often a schema
- Handling control flow
  - Called ***confluence operator***
  - Combines flow values from different paths

Mathematical model

- Flow values
  - Elements of a lattice $L = (P, \subseteq)$
  - Flow value $v \in P$
- Transfer functions
  - Set of functions (one for each instruction)
  - $F_i : P \rightarrow P$
- Confluence operator
  - Merges lattice values
  - $C : P \times P \rightarrow P$
- How does this help us?

Lattices

- Lattice L = (P, $\subseteq$)
- A partial order relation $\subseteq$

  Reflexive, anti-symmetric, transitive
- Upper and lower bounds

  Consider a subset S of P
  - Upper bound of S: $u \in S : \forall x \in S \;\; x \subseteq u$
  - Lower bound of S: $l \in S : \forall x \in S \;\; l \subseteq x$
- Lattices are complete

  Unique greatest and least elements
  - "Top" $T \in P : \forall x \in P \;\; x \subseteq T$
  - "Bottom" $\bot \in P : \forall x \in P \; \bot \subseteq x$

Confluence operator

- Combine flow values
  - "Merge" values on different control-flow paths
  - Result should be a safe over-approximation
  - We use the lattice $\subseteq$ to denote "more safe"

- Example: live variables
  - v1 = {x, y, z} and v2 = {y, w}
  - How do we combine these values?
  - v = v1 $\cup$ v2 = {w, x, y, z}
  - What is the "$\subseteq$" operator?
  - Superset

Meet and join

- **Goal**:
  Combine two values to produce the "best" approximation
  - Intuition:
    - Given v1 = {x, y, z} and v2 = {y, w}
    - A safe over-approximation is "all variables live"
    - We want the smallest set
- Greatest lower bound
  - Given $x,y \in P$
  - GLB(x,y) = z such that
    - $z \subseteq x$ and $z \subseteq y$ and
    - $\forall w\ w \subseteq x$ and $w \subseteq y \Rightarrow w \subseteq z$
  - ***Meet*** operator: $x \wedge y = \text{GLB}(x, y)$
- Natural "opposite": Least upper bound, ***join*** operator

Termination

- Monotonicity

  Transfer functions F are ***monotonic*** if
  - Given x,y $\in$P
  - If x $\subseteq$ y then F(x) $\subseteq$ F(y)
  - Alternatively: F(x) $\subseteq$ x

- Key idea:

  Iterative dataflow analysis terminates if
  - Transfer functions are monotonic
  - Lattice has finite height
  - *Intuition*: values only go down, can only go to bottom

Example

- Prove monotonicity of live variables analysis
  - Equation: **in**[i] = (**out**[i] – **def**[i]) $\cup$ **use**[i]

    (For each instruction i)

  - As a function: $F(x) = (x - def[i]) \cup use[i]$
  - Obligation: If $x \subseteq y$ then $F(x) \subseteq F(y)$
  - Prove:

    $x \subseteq y \quad => \quad (x - def[i]) \cup use[i] \subseteq (y - def[i]) \cup use[i]$

    - Somewhat trivially:
    - $x \subseteq y \Rightarrow x - s \subseteq y - s$
    - $x \subseteq y \Rightarrow x \cup s \subseteq y \cup s$

Dataflow solution

- Question:
  - What is the solution we compute?
  - Start at lattice top, move down
  - Called greatest ***fixpoint***
  - Where does approximation come from?
  - Confluence of control-flow paths
- Knaster Tarski theorem
  - Every monotonic function F over a complete lattice L has a unique least (and greatest) fixpoint
  - (Actually, the theorem is more general)

Summary

- Dataflow analysis
  - Lattice of flow values
  - Transfer functions (encode program behavior)
  - Iterative fixpoint computation

- **Key insight**:

  If our dataflow equations have these properties:
  - Transfer functions are monotonic
  - Lattice has finite height
  - Transfer functions distribute over meet operator

  Then:
  - Our fixpoint computation will terminate
  - Will compute meet-over-all-paths solution