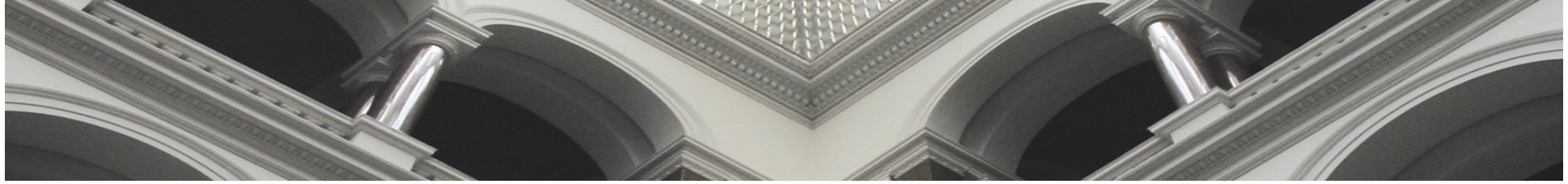
Technische Universität Berlin

Compiler Design

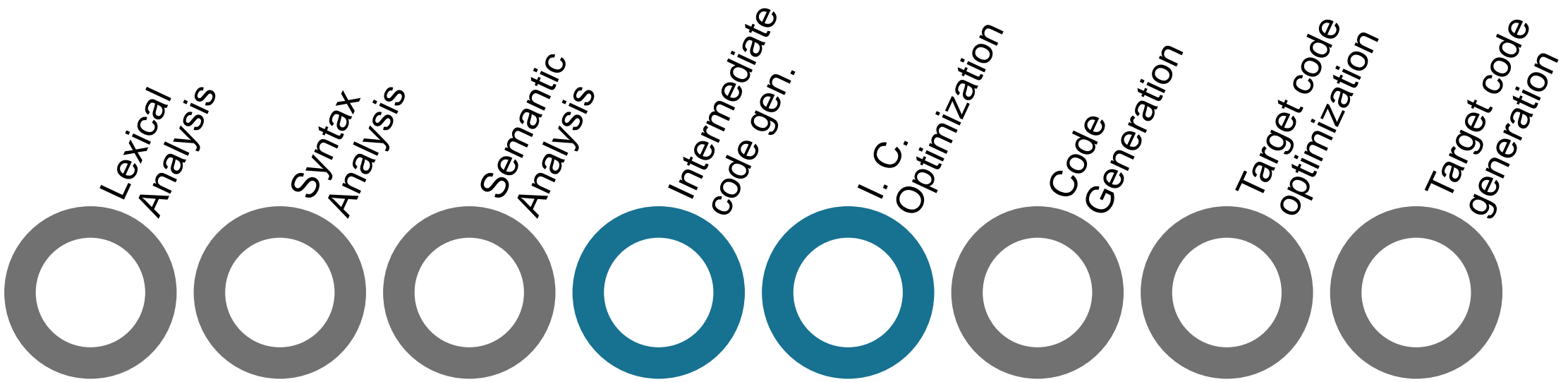
Data-Flow Analysis

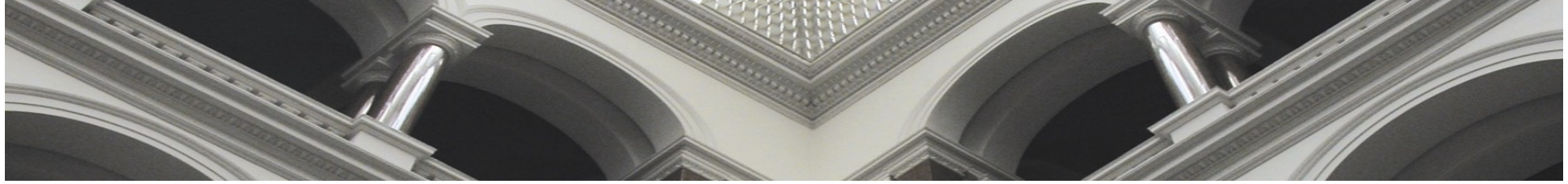
Dr. Nicolai Stawinoga, Dr. Biagio Cosenza | TU Berlin | Wintersemester 2022-23



Technische Universität Berlin

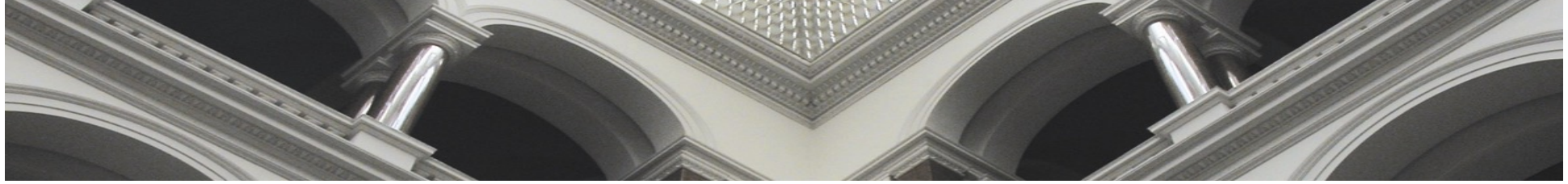
We are here





Data-Flow Analysis

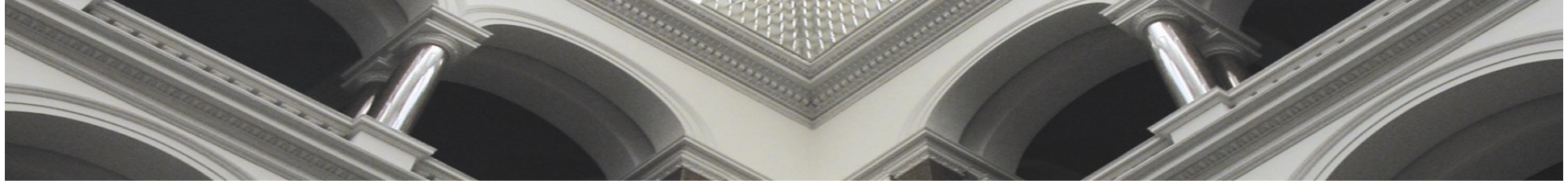
- Source code parsed to produce AST
- AST transformed to Control Flow Graph (CFG)
- Data-flow analysis operates on CFG (and other intermediate representations)



Technische Universität Berlin

AST

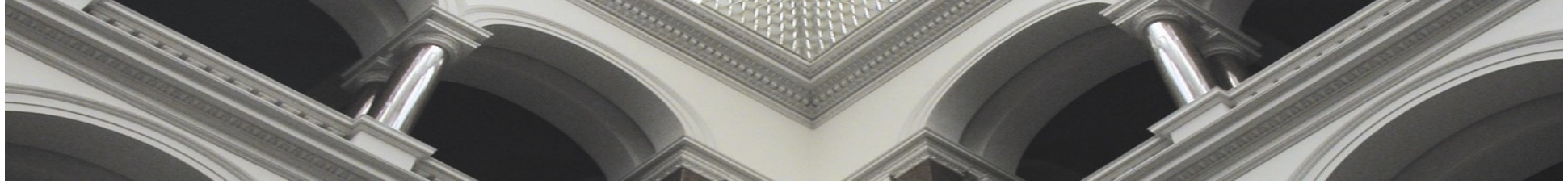
- AST is a high-level, close-to-source, abstract representation
  - ➢ do not contain all information in the program
    - E.g., spacing, comments, brackets, parentheses
  - ➢ any ambiguity has been resolved
    - E.g., `a+b+c` produces the same AST as `(a+b)+c`
- Disadvantages
  - ➢ AST has many similar forms
    - `for, while, repeat...until`
    - `if, ?:, switch`
  - ➢ expressions in AST may be complex, nested
    - `(42 * y) + (z > 5 ? 12 * z : z + 20)`
- Want simpler representation for analysis



Control-Flow Graph (CFG)

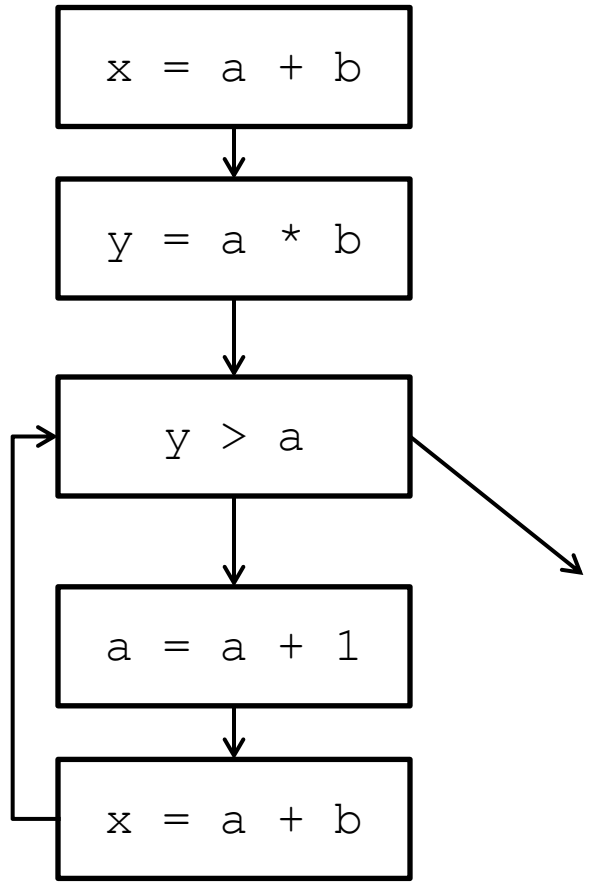
- A directed graph where
  - each node represents a statement
  - edges represent control flow
- Statements may be
  - assignments `x = y op z` or `x = op z`
  - copy statements `x = y`
  - branches `goto L` or `if x relop y goto L`
  - etc

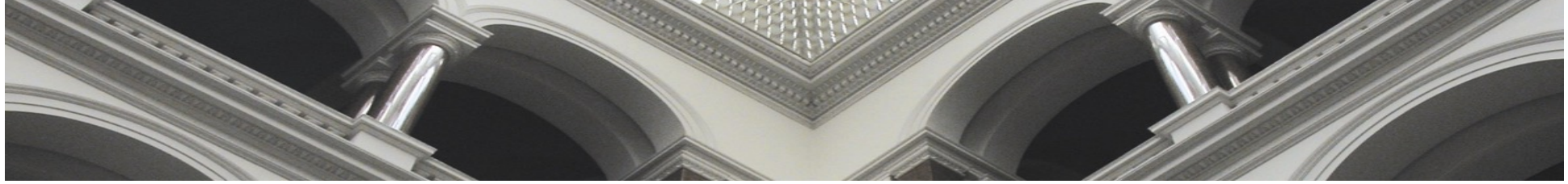
Relational operator such as <,≤,=,…



Control-Flow Graph Example (recap)

```
1. x = a + b;
2. y = a * b;
3. while (y > a) {
4.     a = a + 1;
5.     x = a + b;
6. }
```

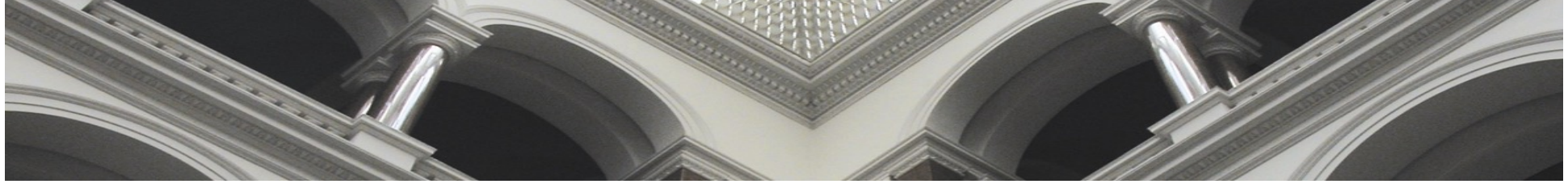




Technische Universität Berlin

CFG vs AST

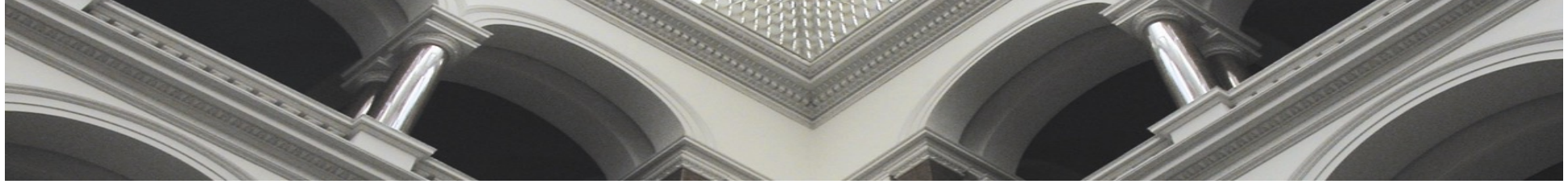
- ASTs are closer to the input language
  - CFGs introduce temporaries
  - lose block structure of program
  - easier to report error and other messages
  - easier to unparse and explain to programmer
- CFGs are much simpler than ASTs
  - fewer forms
  - less redundancy
  - only simple expressions



Technische Universität Berlin

Variations on CFGs

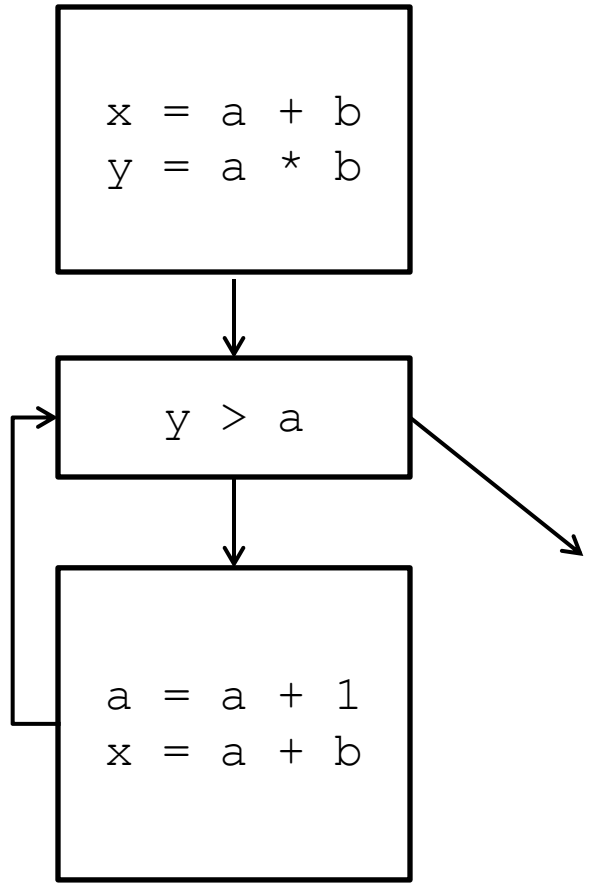
- CFGs usually don't include declarations (e.g., `int x`)
  - ➢ but implementations have *something* for memory allocation
  - ➢ E.g.: LLVM's `alloca` instruction
- May want a unique entry and exit node
  - ➢ E.g.: LLVM Regions: A single entry single exit Region
    - A (LLVM) region is a connected subgraph of a control flow graph that has exactly two connections to the remaining graph. It can be used to analyze or optimize parts of the control flow graph
- May group statements into basic blocks
  - ➢ a sequence of instructions with no branches into or out of the block

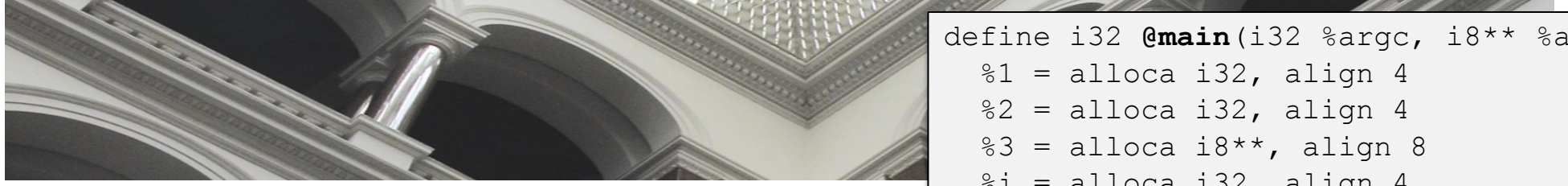


CFG with Basic Blocks (recap)

- Basic Block: Straight-line code sequence with no branches in except to the entry and no branches out except at the exit
  - ➢ more efficient representation

```
1. x = a + b;
2. y = a * b;
3. while (y > a) {
4.       a = a + 1;
5.       x = a + b;
6. }
```

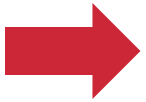




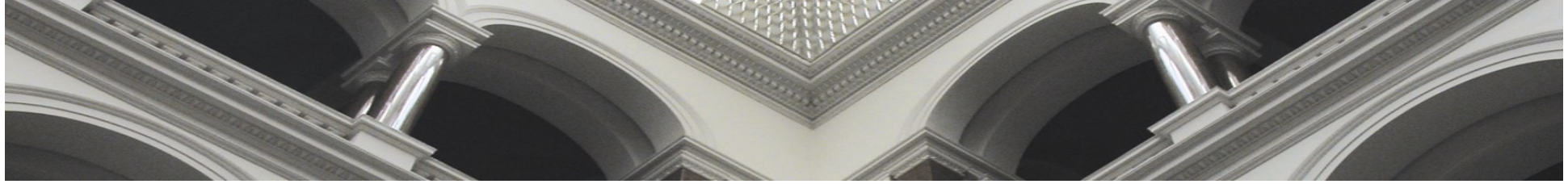
Example: BBs in the LLVM IR

- LLVM basic blocks on a simple input code
  - `clang ./loop2.c -o loop2.bc -c -emit-llvm`
  - `llvm-dis < loop2.bc`

```
int main(int argc, char **argv) {
 for (unsigned i = 0; i < 10; ++i){
   printf("%d\n",i);
 }
 return 0;
}
```



```
define i32 @main(i32 %argc, i8** %argv) #0 {
  %1 = alloca i32, align 4
  %2 = alloca i32, align 4
  %3 = alloca i8**, align 8
  %i = alloca i32, align 4
  store i32 0, i32* %1
  store i32 %argc, i32* %2, align 4
store i8** %argv, i8*** %3, align 8
  store i32 0, i32* %i, align 4
  br label %4

; <label>:4  ; preds = %10, %0
  %5 = load i32* %i, align 4
  %6 = icmp ult i32 %5, 10
  br i1 %6, label %7, label %13

; <label>:7  ; preds = %4
  %8 = load i32* %i, align 4
  %9 = call i32 (i8*, ...)* @printf(i8* getelementptr
inbounds ([4 x i8]* @.str, i32 0, i32 0), i32 %8)
  br label %10

; <label>:10  ; preds = %7
  %11 = load i32* %i, align 4
  %12 = add i32 %11, 1
  store i32 %12, i32* %i, align 4
  br label %4

; <label>:13  ; preds = %4
  ret i32 0
}
```



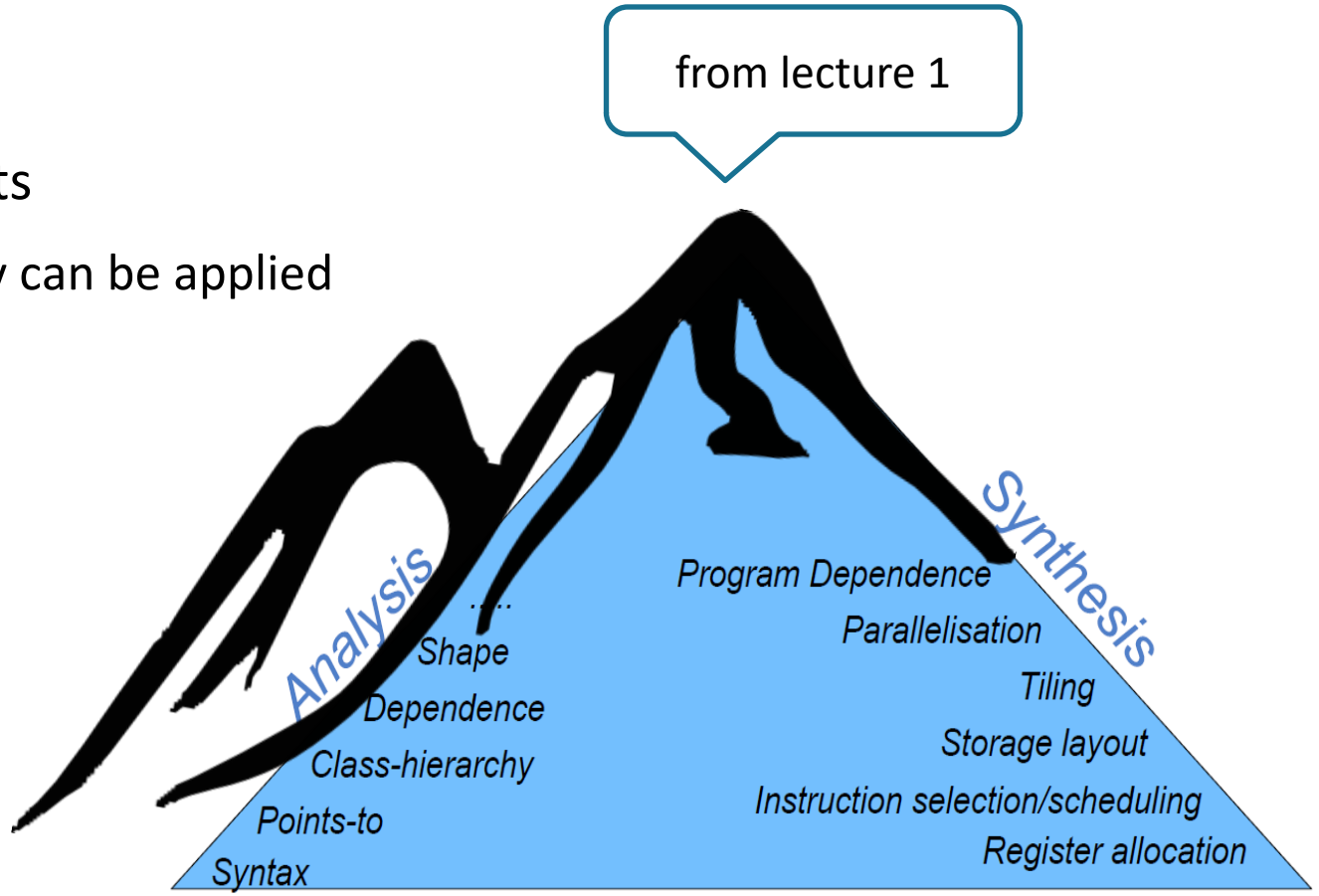
Exercise

- Given the following code, organize it on Basic Blocks
  - For printf/scanf just use a generic CallInst

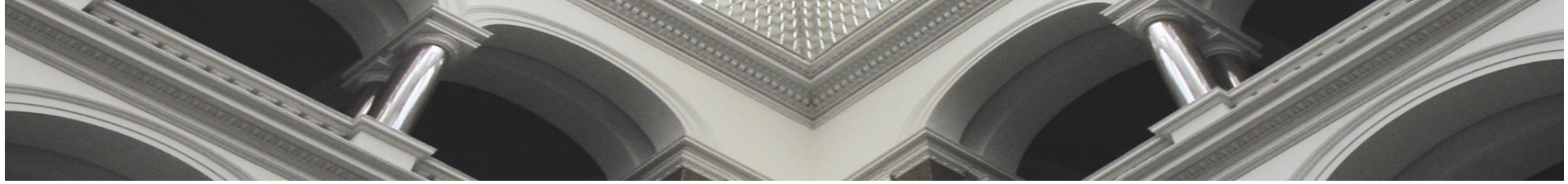
```
#include <stdio.h>
main()
{
   int n, c = 2;

   printf("Enter a number to check if it is prime\n");
   scanf("%d",&n);

   for ( c = 2 ; c <= n - 1 ; c++ )
   {
      if ( n%c == 0 )
      {
         printf("%d is not prime.\n", n);
         break;
      }
   }
   if ( c == n )
      printf("%d is prime.\n", n);
   return 0;
}
```

Analysis & Optimization

- Any optimization has two components
  1. identify conditions under which they can be applied
  2. code transformation itself
- (1) is (usually) the hardest part!
  - ➢analysis is required to solve it



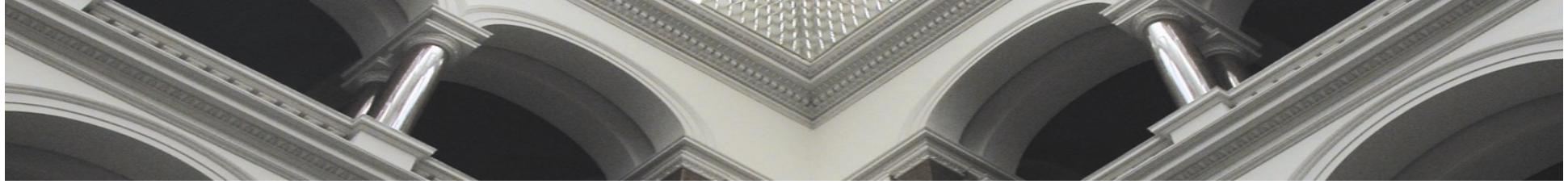
Courtesy of Paul Kelly, Imperial College London



Technische Universität Berlin

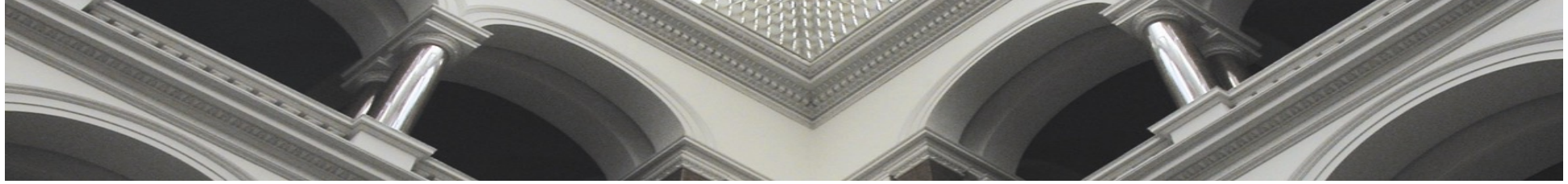
Data-Flow Analysis

- A framework for proving facts (properties) about programs (control flow graph)
- Reasons about lots of little facts (instruction/basic block property)
- Little or no interaction between facts
  - ➢ works best on properties about how program computes
  - ➢ fact are usually propagated forward or backward on the control flow graph
- Based on all paths through program
  - ➢ including infeasible paths



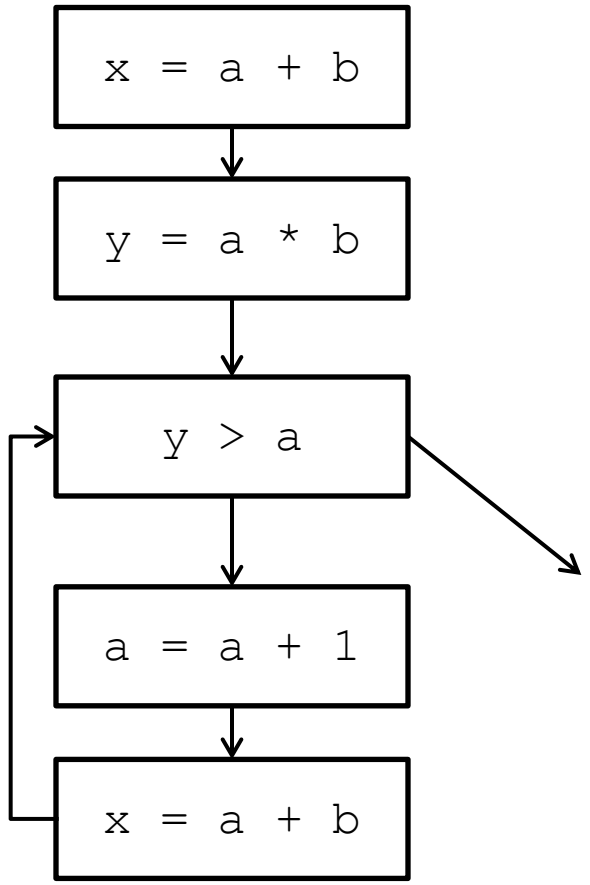
Available Expression

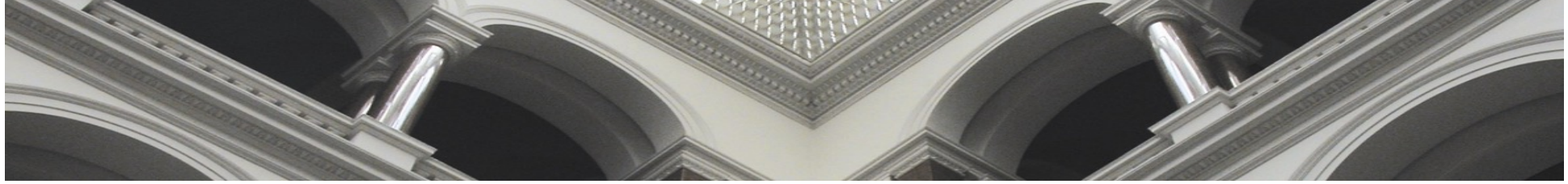
- An expression e is available at program point p if
  - e is computed on every path to p, and
  - the value of e has not changed since last time e is computed on p
- In other words
  - an expression `x+y` is available at point **p** if every path from the entry node to **p** evaluates `x+y`, and after the last such evaluation prior to reaching **p**, there are no subsequent assignment to **x** or **y**
    - expression like `x+y`, where `+` can be a generic operator
- Optimization
  - if an expression is available, no need to be recomputed
    - at least, if it is still in a register somewhere



Data-Flow Facts

- Is expression **e** available?
- Facts, for the three expressions
  - `a + b` is available
  - `a * b` is available
  - `a + 1` is available
- We start with `AVAIL=∅`
- Note
  - Expression has operand, e.g., `a + b`
  - `AVAIL` set before vs after the Basic Block

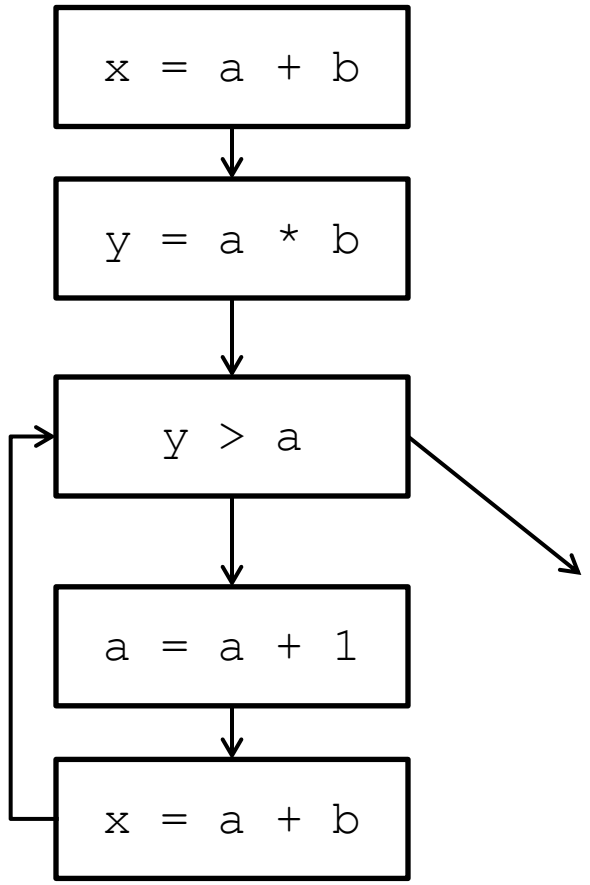


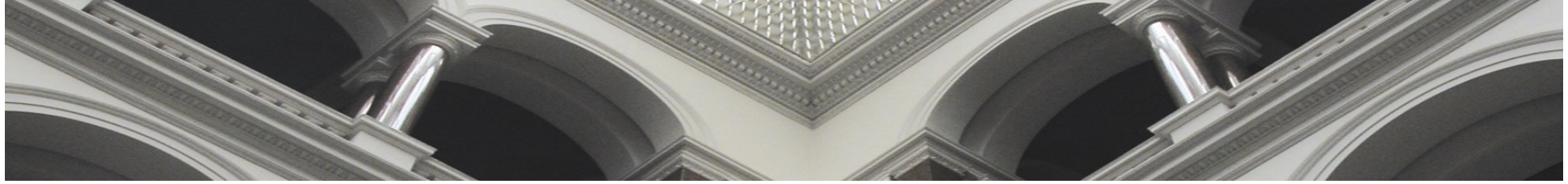


Gen and Kill

- What is the effect of each statement on the set of facts?

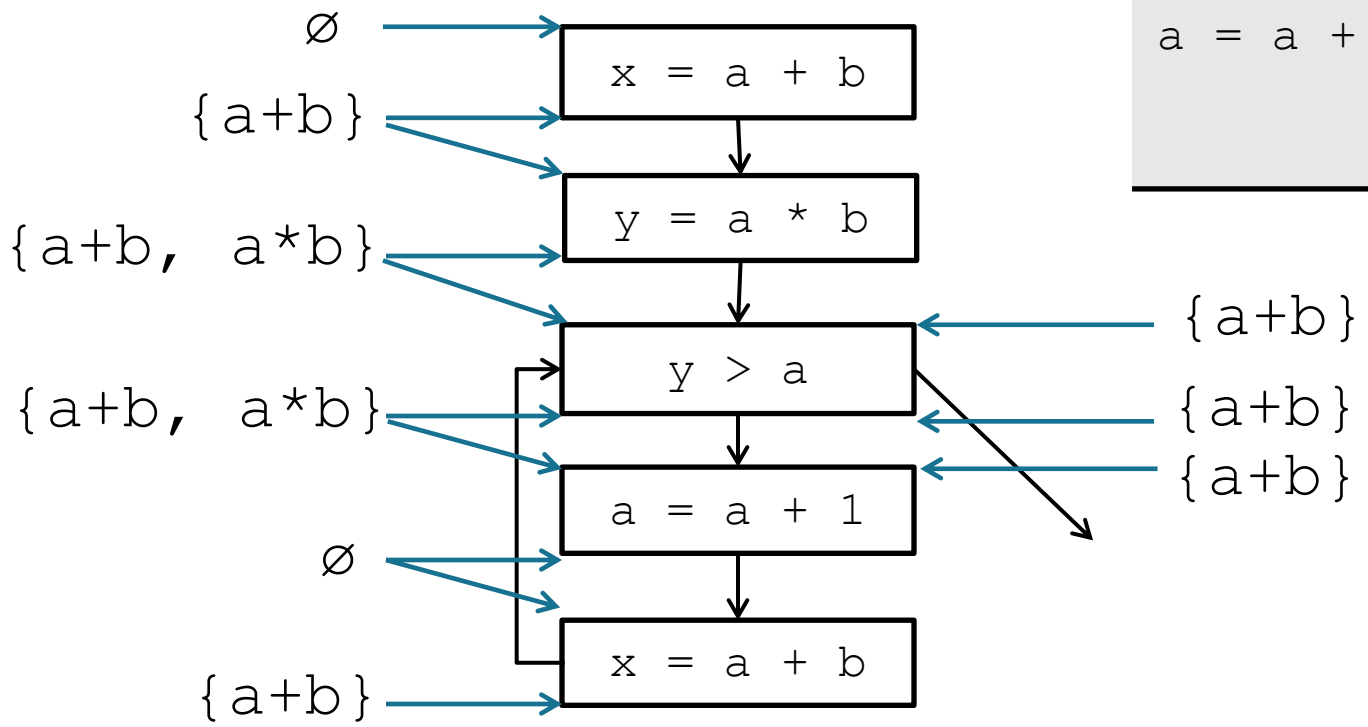
Statement	Gen	Kill
`x = a + b`	`a + b`	
`y = a * b`	`a * b`	
`a = a + 1`		`a + 1`, `a + b`, `a * b`

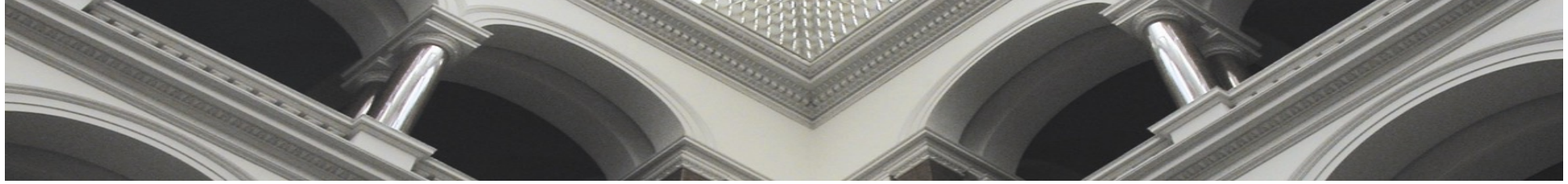




Computing Available Expressions

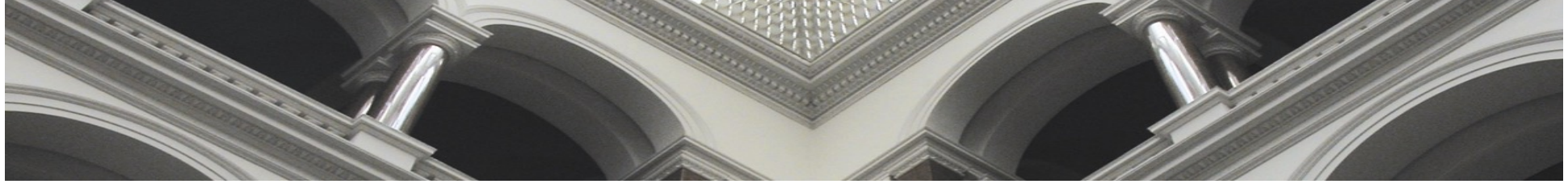
Statement	Gen	Kill
x = a + b	a + b	
y = a * b	a * b	
a = a + 1		a + 1, a + b, a * b





Terminology

- A join point is a program point where two branches meet
- Available expressions is a forward must problem
  - Forward = Data flow from in to out
  - Must = At join point, property must hold on all paths that are joined



Data-flow Equations for AVAIL

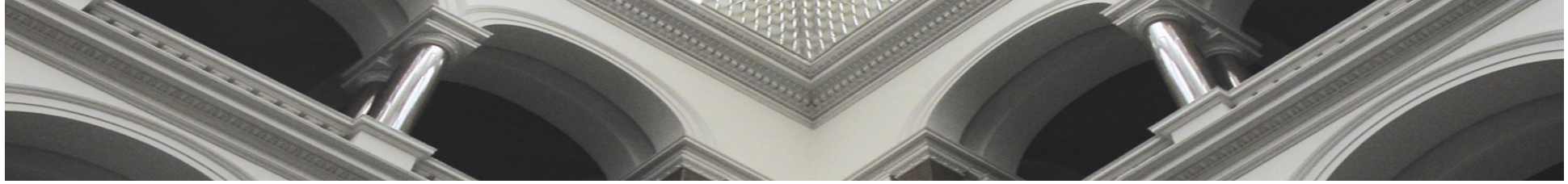
- Let **s** be a statement
  - succ(s)= { *immediate successor statements of* s }
  - pred(s)= { *immediate predecessor statements of* s }
  - In(s) = { expressions at program point just before executing s }
  - Out(s) = { expressions at program point just after executing s }

they also apply with basic block

- We define the following transfer functions

$$\mathbf{In(s) = \bigcap_{s' \in pred(s)} Out(s')}$$

$$\mathbf{Out(s) = Gen(s) \cup (In(s) - Kill(s))}$$

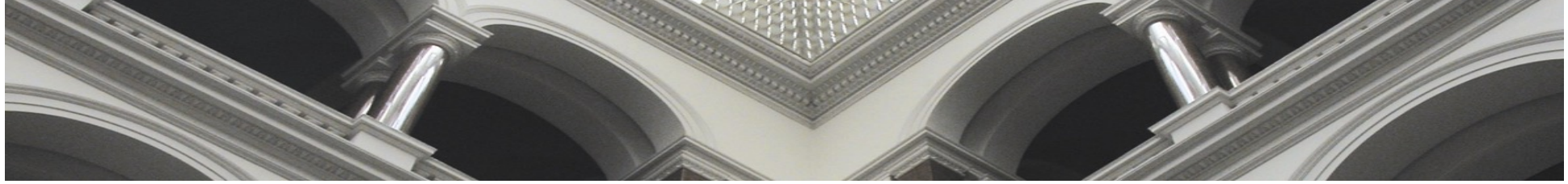


Available Expressions: Exercise 1

- Write a CFG for the following code

```
x = a + b;
y = a * b;
z = c + d;
while (y > a + b){
  a = a + 1;
  x = a + b;
}
z = c + d;
```

- Compute the available expressions set for each statement

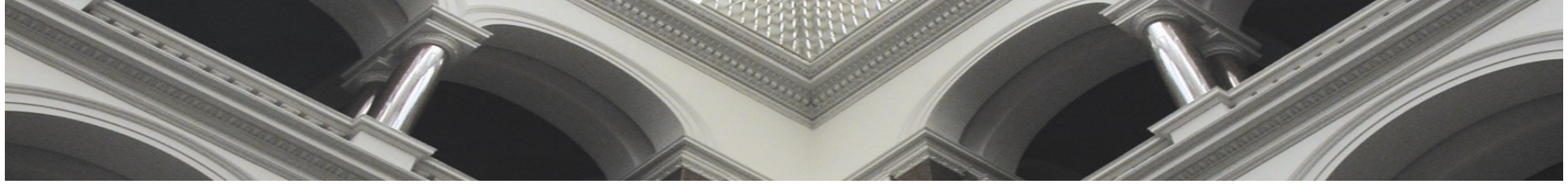


Available Expressions: Exercise 2

- Write a CFG for the following code

1. compute the available expressions set for each statement
2. do it again by using a CFG organized in Basic Blocks

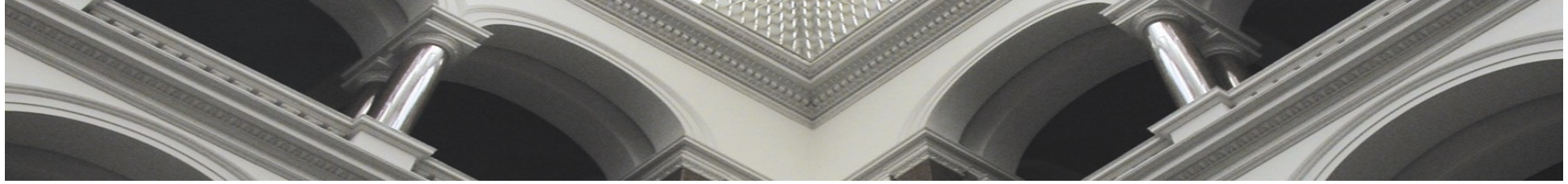
```
int binary_search(int n, int a[n], int who)
{
  int left = 0;
  int right = n-1;
  while (left <= right) {
    int mid = left + (right-left)/2;
    if (who < a[mid])
      right = mid - 1;
    else if (who > a[mid])
      left = mid + 1;
    else
      return mid;
  }
  return -1;
}
```



Technische Universität Berlin

General Terminology on Data-Flow

- Domain
  - ➢ The type of information handled by the analysis. Here: (set of available) expressions; later: variables, definitions
- Direction
  - ➢ Forward vs backward
- Transfer function
- Boundary (the starting point, here: ∅)
- Meet operator (∧)
  - ➢ Usually union or intersection
- Equations
- Initialization

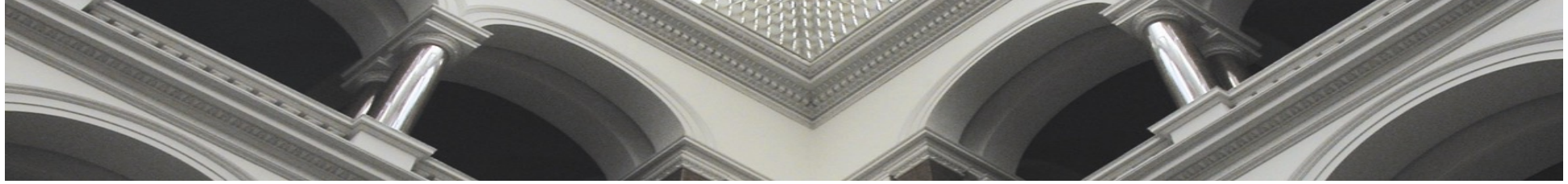


Liveness Analysis

- A variable **v** is live at program point **p** if
  - **v** will be used on some execution path originating from **p**...
  - before **v** is overwritten

Live = may be used in the future

- Simply: a variable is live if it holds a value that may be needed in the future
- Each instruction has an associated set of live variables
  - `int z = x * y` → `LIVE(n) = {s, t, x, y}`
- Optimization
  - if a variable is not live, no need to keep it in a register
  - if variable is dead at assignment, can eliminate assignment



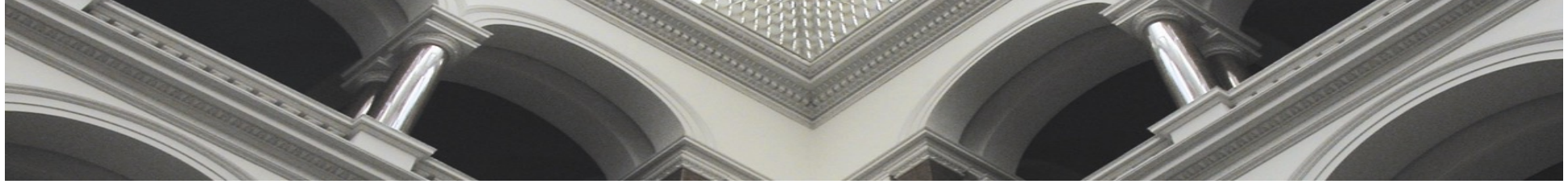
Data-Flow Equations for Liveness Analysis

- Available expressions is a forward must analysis
  - ➢ data flow propagate in same direction as CFG edges
- Liveness is a backward may problem
  - ➢ to know if variable live, need to look at future uses
  - ➢ variable is live if used on some path
- Transfer functions

$$\mathtt{Out(s)} = \bigcup_{s' \in succ(s)} \mathtt{In(s')}$$

$$\mathtt{In(s)} = \mathtt{Gen(s)} \cup (\mathtt{Out(s)} - \mathtt{Kill(s)})$$

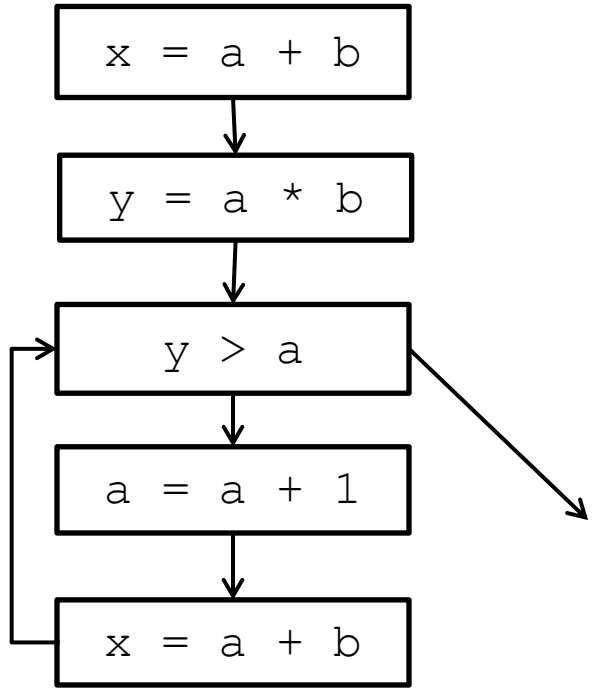
Note on terminology:
USE substitute for **GEN**
DEF substitute for **KILL**

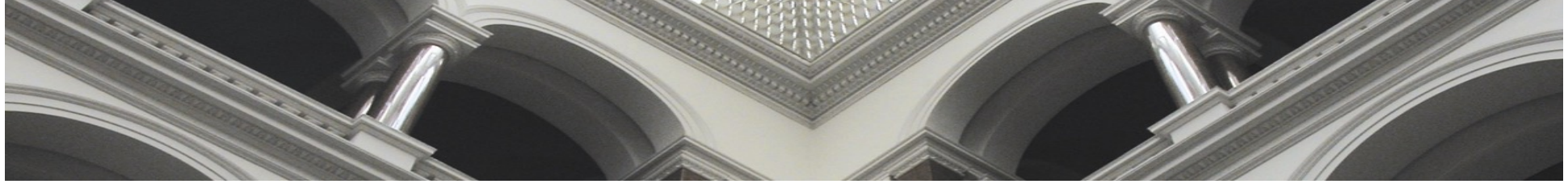


Gen and Kill (also: Use and Def)

- What is the effect of each statement on the set of facts?

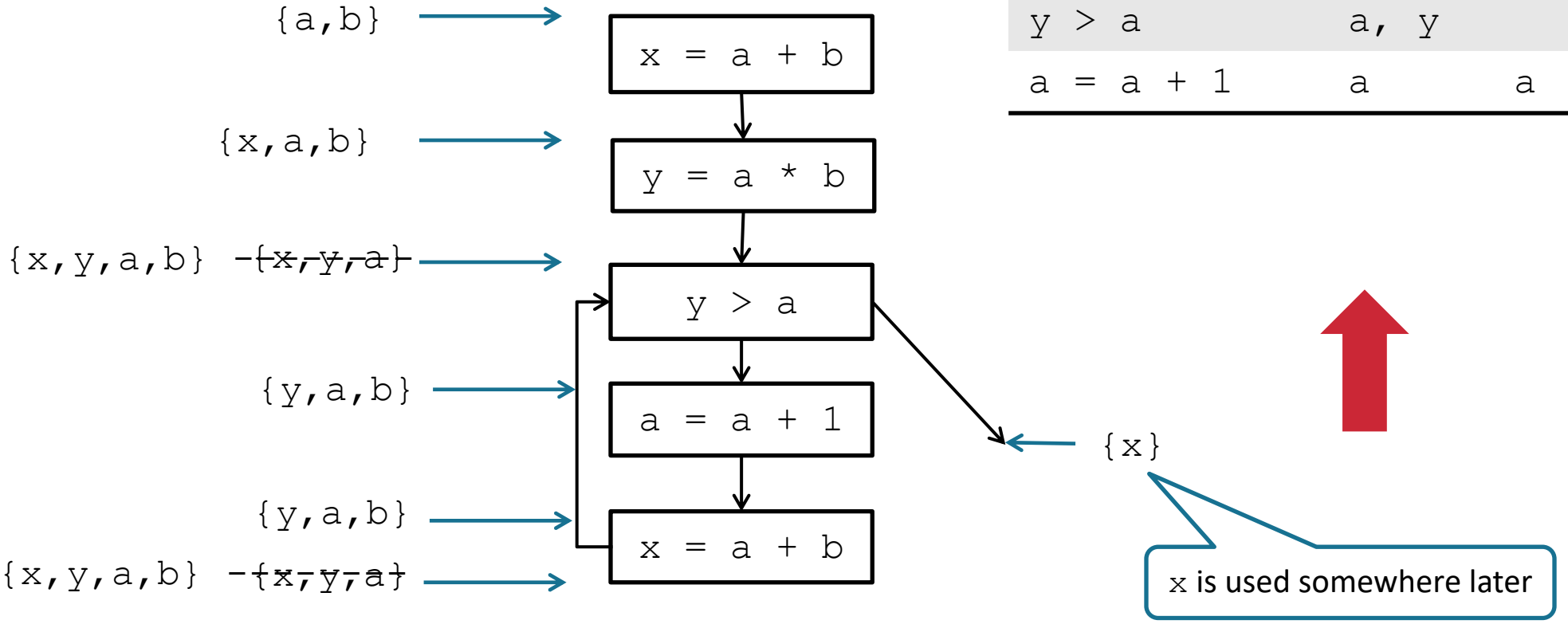
Statement	Gen	Kill
`x = a + b`	`a, b`	`x`
`y = a * b`	`a, b`	`y`
`y > a`	`a, y`	
`a = a + 1`	`a`	`a`

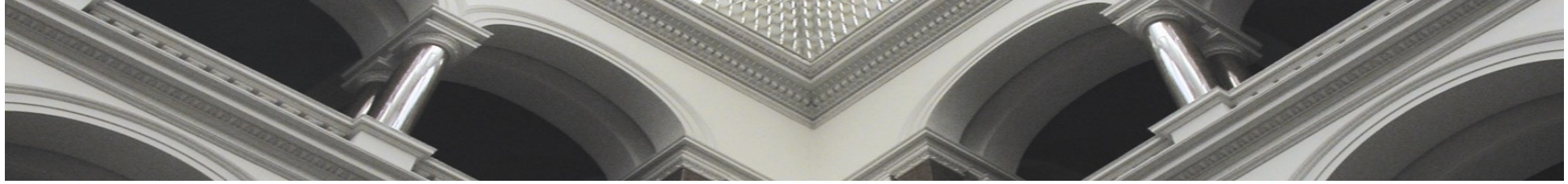




Computing Live Variables

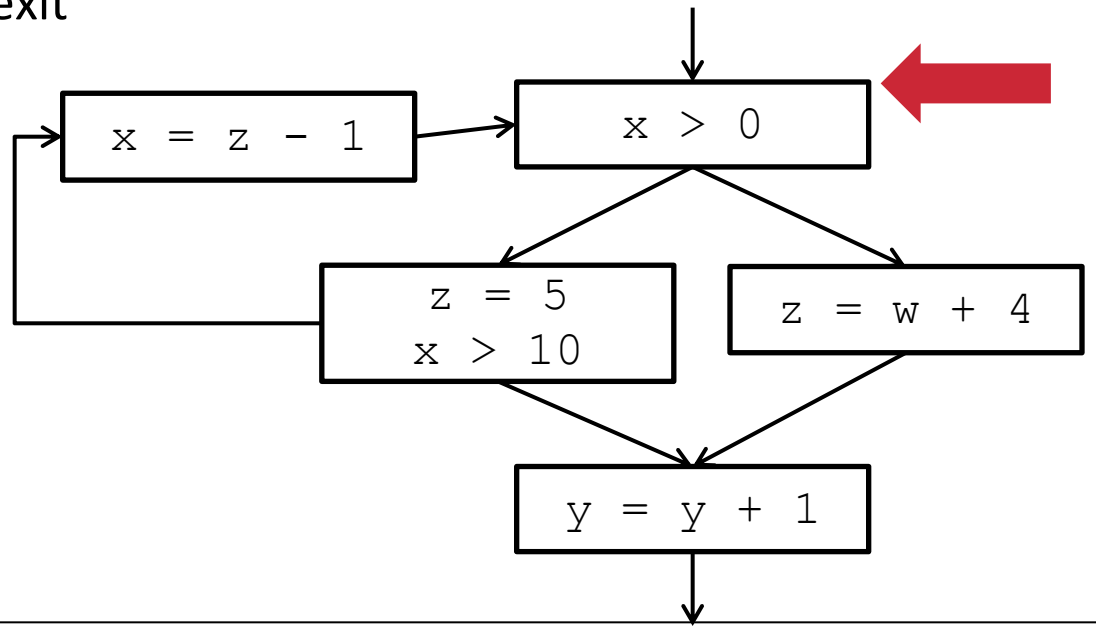
Statement	Gen	Kill
x = a + b	a, b	x
y = a * b	a, b	y
y > a	a, y	
a = a + 1	a	a



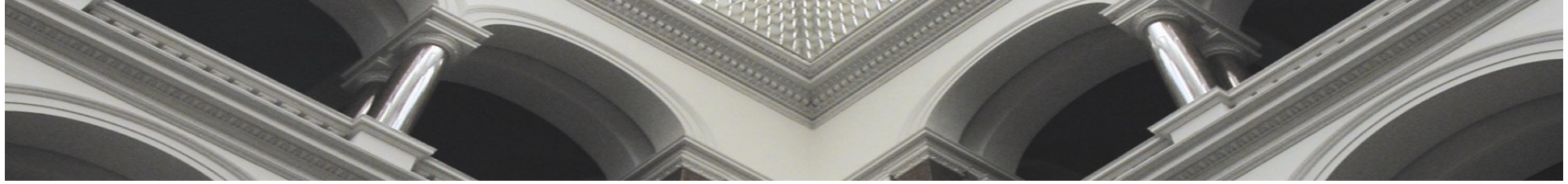


Exercise on Liveness Analysis

- After running the liveness analysis algorithm to completion, which of `w`, `x`, `y`, and `z` are live at the program point labeled at right?
  - Assume all variables are dead on exit
  - Reminder: backward analysis

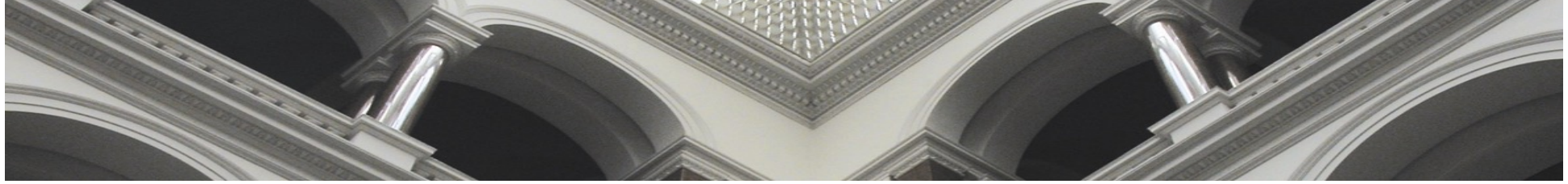


Answer: `w`, `x`, `y`



End of part one

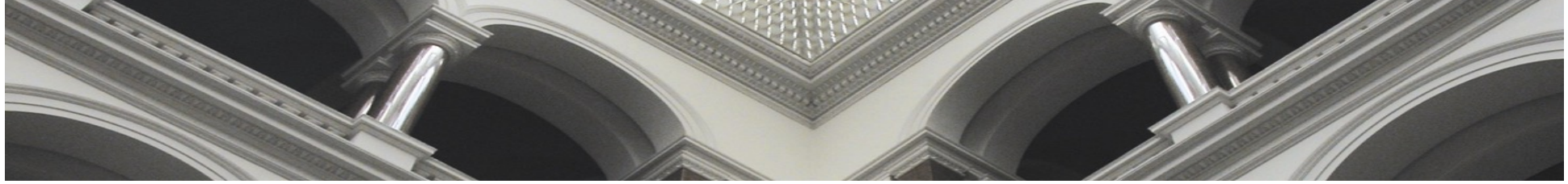
- Next
  - algorithms for data-flow analysis
  - reaching definitions
  - implementation issues
  - recap of data-flow algorithms



Technische Universität Berlin

Dataflow Analysis, Part 2

- We have seen two examples of dataflow analysis
  - available expression, liveness analysis
- Today's lecture
  - one more example: reaching definitions
  - algorithms for data-flow analysis: round-robin, worklist
  - implementation issues: bitset, traversal ordering
  - concepts of a general framework for dataflow analysis
  - recap of data-flow algorithms

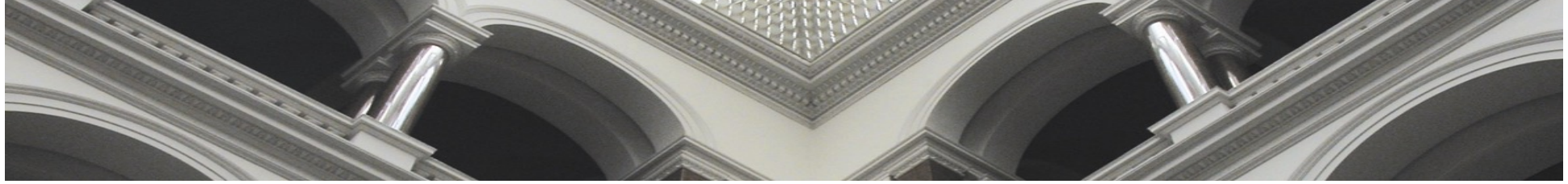


Reaching Definitions

- Determine a set of definitions reaching the entry and exit of each block of the control graph
- A definition **d** reaches a point **p** if there is a path from the point immediately following **d** to **p**, such that **d** is not killed along that path
  - ➢Kill a definition of a variable x: if there is any other definition of x anywhere along that path
- Optimizations

Domain: definitions

  - ➢constant propagation
    - we need to know that all the definitions that reach a variable assign it to the same constant
  - ➢copy propagation
    - we need to know whether a particular copy statement is the only definition that reaches a use.
  - ➢code motion
    - we need to know whether a computation is loop-invariant



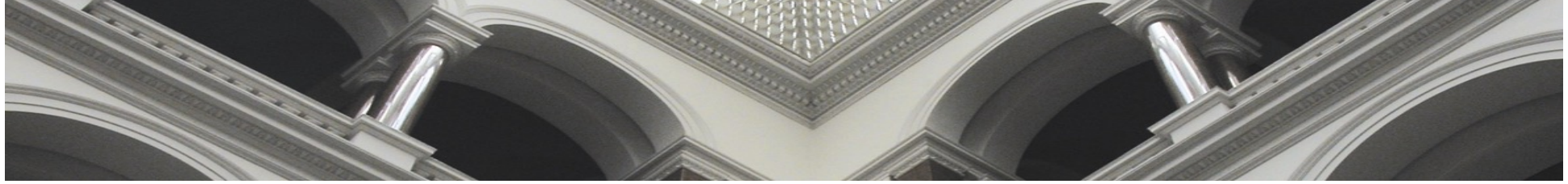
Reaching Definitions: Gen and Kill

- Consider the definition: **d:** `u = v + w`
  - `GEN(s)={d}`
  - Kills all other definitions in the program that define the variable **u**, while leaving the remaining incoming definitions unaffected
- Flow: forward
- Meet operator: union
- Transfer functions

domain is a set of definitions $d_1, d_2, \ldots, d_n$

$$In(s) = \bigcup_{p \in pred(s)} Out(p)$$

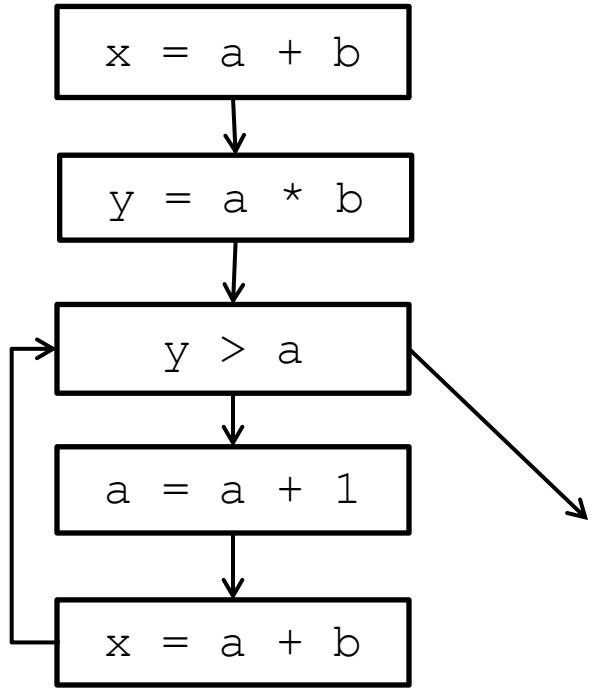
$$Out(s) = Gen(s) \cup (In(s) - Kill(s))$$

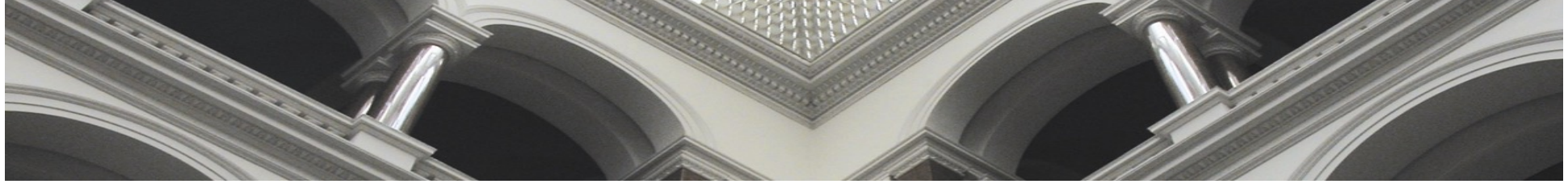


Reaching Definitions: Gen and Kill

- What is the effect of each statement on the set of facts?

Statement	Gen	Kill
d_1: x = a + b	d_1	d_4
d_2: y = a * b	d_2	
d_3: a = a + 1	d_3	d_3
d_4: x = a + b	d_4	d_1



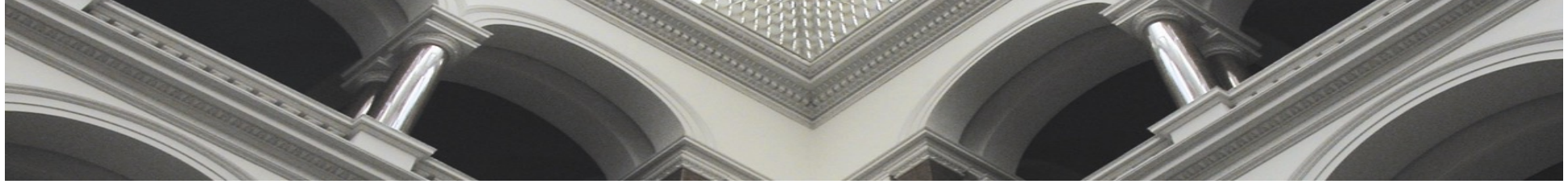


Reaching Definitions: Single Basic Block Example

- What is the `Gen` set for the following basic block `B`?

```
d1: a = 3
d2: a = 4
```

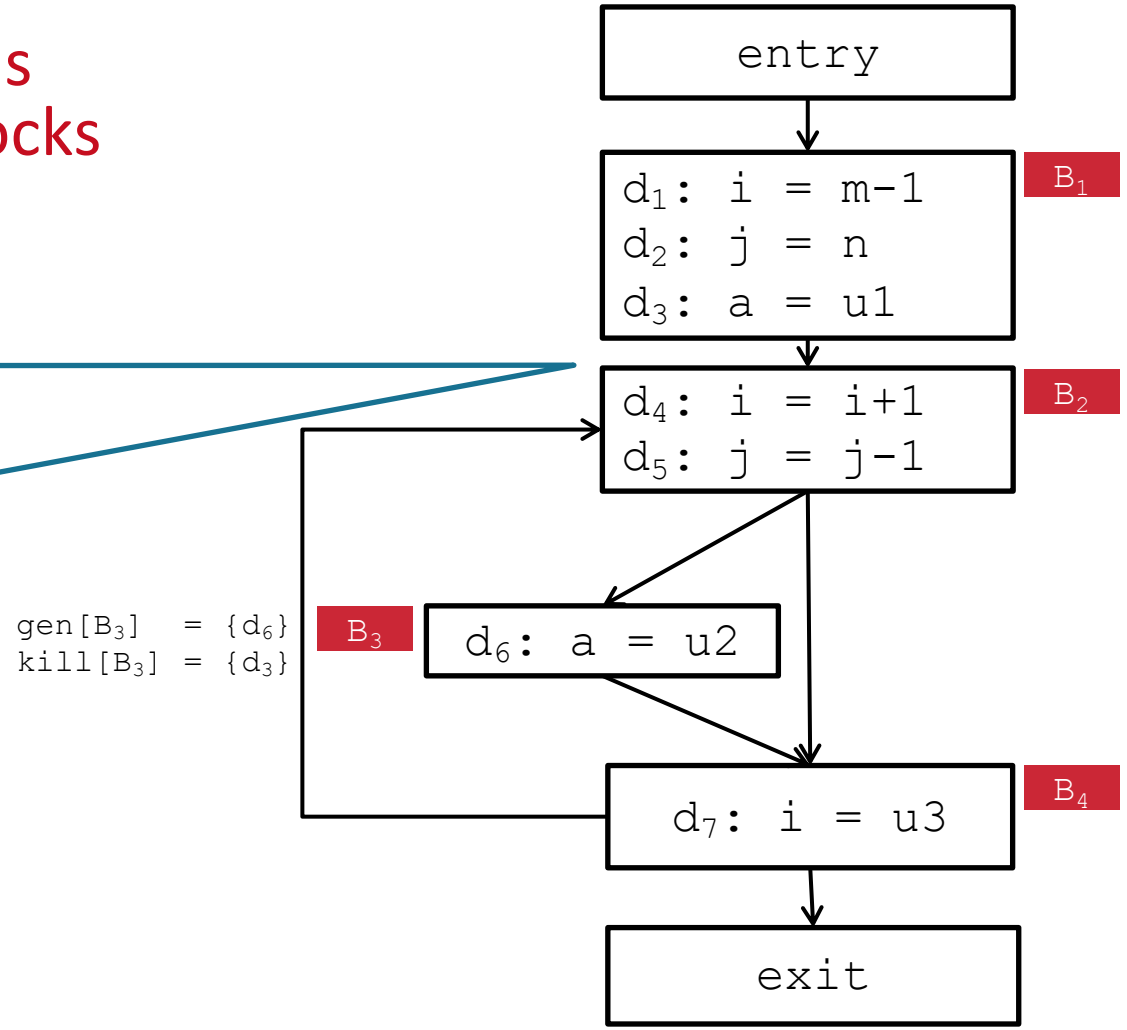
- $Gen[B] = \{d_2\}$
  - d_1 is not downward exposed
  - the kill set contains both d_1 and d_2, since d_1 kills d_2 and vice-versa. Nonetheless, since the subtraction of the kill set precedes the union operation with the gen set, the results of the transfer function for this block always includes definition d_2



Reaching Definitions with more Basic Blocks
[ALSU pp 602 and 604]

Which definitions reach the beginning of B_2?

- all def of B_1
- d_5 (no other def of j kill d_5 in the loop)
- d_5 kills d_2
- d_7 kills d_4

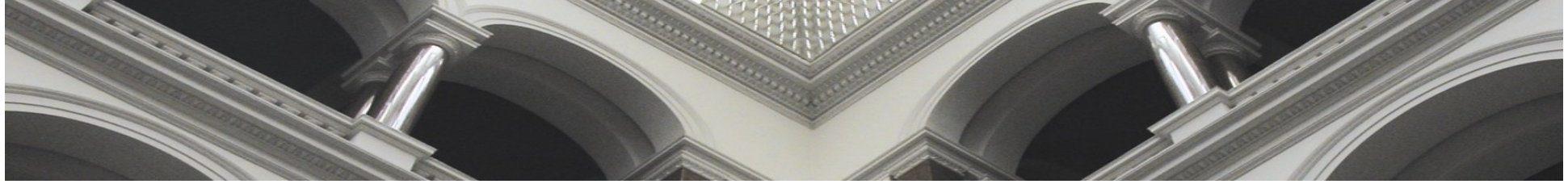


gen[B_3] = {d_6}
kill[B_3] = {d_3}

gen[B_1] = {d_1,d_2,d_3}
kill[B_1] = {d_4,d_5,d_6,d_7}

gen[B_2] = {d_4,d_5}
kill[B_2] = {d_1,d_2,d_7}

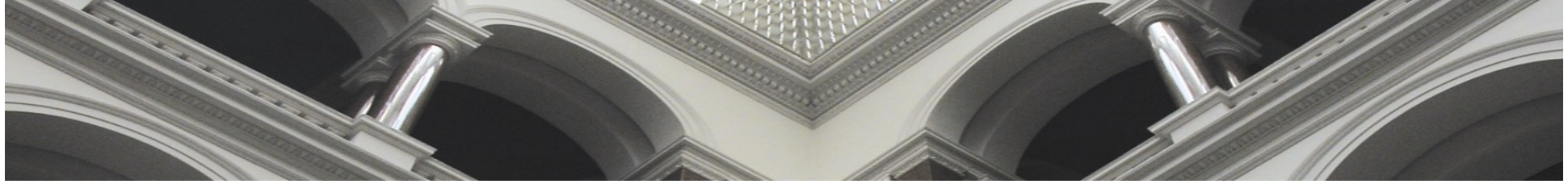
gen[B_4] = {d_7}
kill[B_4] = {d_1,d_4}



An Iterative Algorithm for Data-Flow Analysis

- The most common way of solving the data-flow equations is by using an iterative algorithm
  - starts with an approximation of the in-state of each block
  - out-states are then computed by applying the transfer functions on the in-states
  - from these, the in-states are updated by applying the join operations
  - the latter two steps are repeated until we reach the so-called fixpoint: the situation in which the in-states (and the out-states in consequence) do not change
- Round-robin iterative algorithm

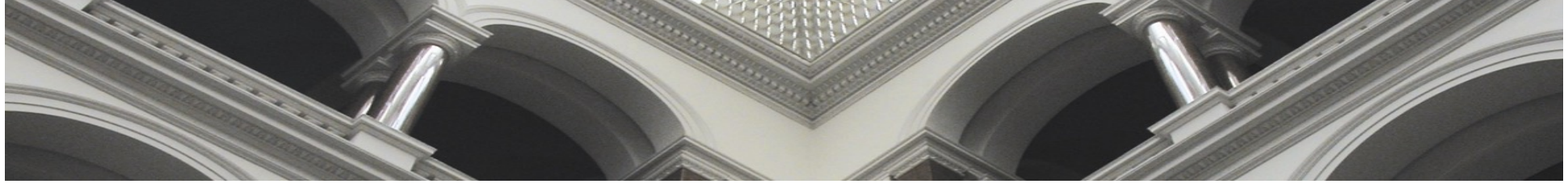
```
for (i = 1 to N)
  initialize node i
while (sets are still changing)
  for (i = 1 to N)
    recompute sets at node i
```



An Iterative Algorithm for Reaching Definitions

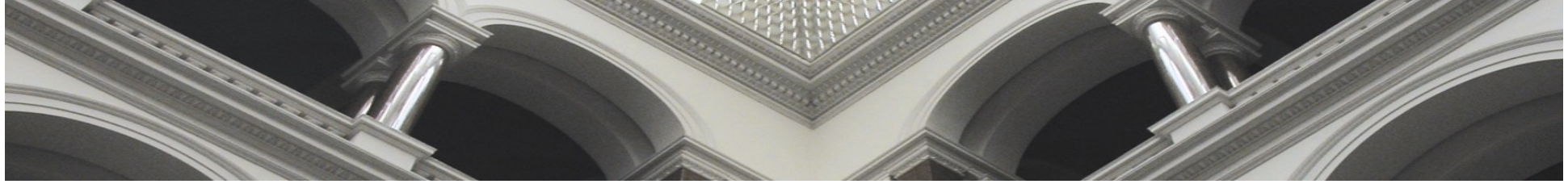
- Input: A control flow graph with `Kill[B]` and `Gen[B]` computed for each block B
- Output: `In[b]` and `Out[B]`, the set of definitions reaching the entry and exit of each block B of the flow graph

```
OUT[ENTRY] = ∅;
for each basic block B in CFG other than ENTRY
    OUT[B] = ∅;
while (changes to any OUT occur) // convergence test
    for each basic block B in CFG other than ENTRY
        // solving data flow equation
        IN[B] = U_{P predecessor of B} OUT[P]
        OUT[B] = gen[B] U (IN[B] – kill[B])
```



The Worklist Algorithm

- Observation: the in-state of a block will not change if the out-states of its predecessors don't change
- Work list: a list of blocks that still need to be processed
  - whenever the **out-state** of a block changes, we add its successors to the work list
  - in each iteration, a block is removed from the work list and its out-state is computed
  - **if** the **out-state changed**, the **block's successors** are **added** to the work list
  - starts by putting information generating blocks in the work list
  - **terminates** when the work list is empty



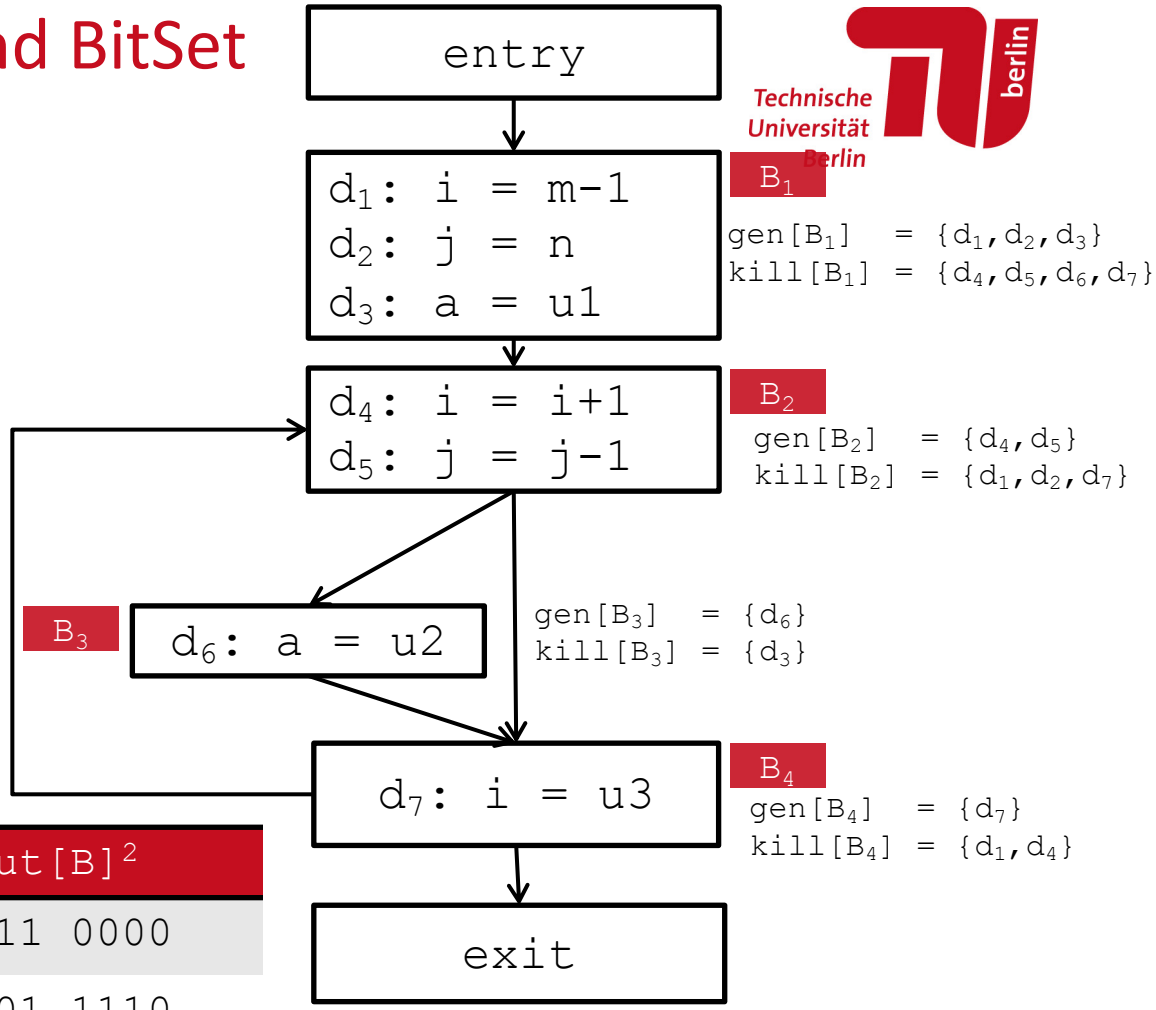
Technische Universität Berlin

Set Implementation

- How do we implement sets?
  - simple way: hashset, treemap, using reference/pointer to elements
- Efficient representation: vector of bit (aka bit set or bitvector)
  - we mark with 0 and 1 for each element in the set
  - e.g., the set $\{d_1, d_2, d_5\}$ becomes `110 010`
- Very efficient join
  - union: logical or
  - intersection: logical and

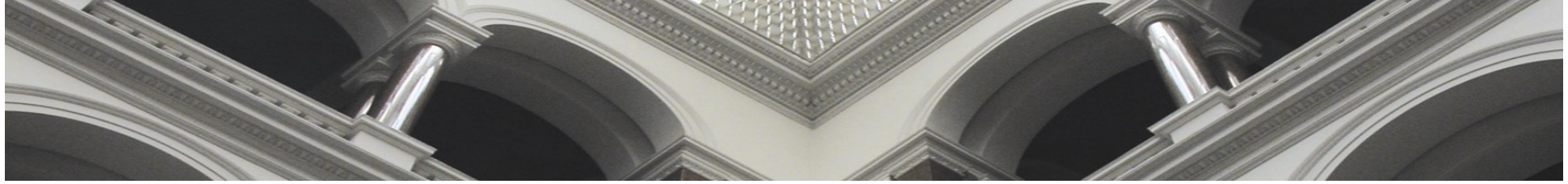
Reaching Definitions with Basic Blocks and BitSet

$$In(s) = \bigcup_{p \in pred(s)} Out(p)$$
$$Out(s) = Gen(s) \cup (In(s) - Kill(s))$$

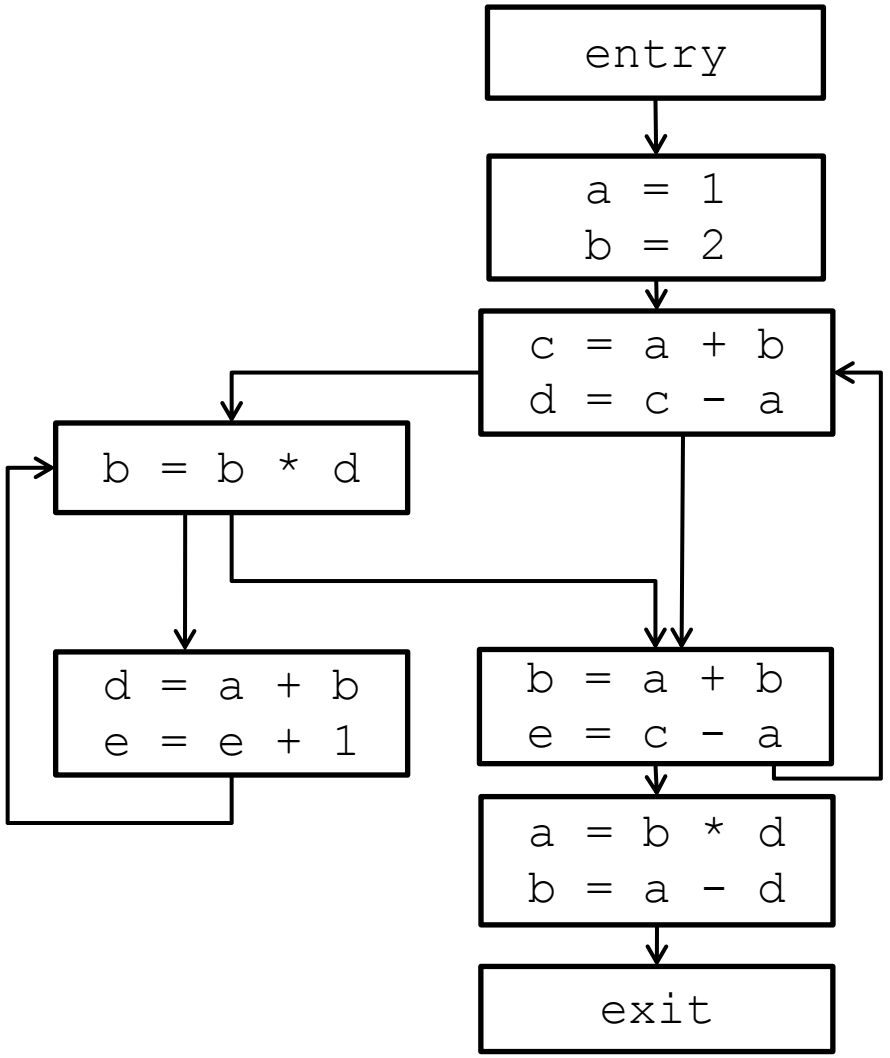


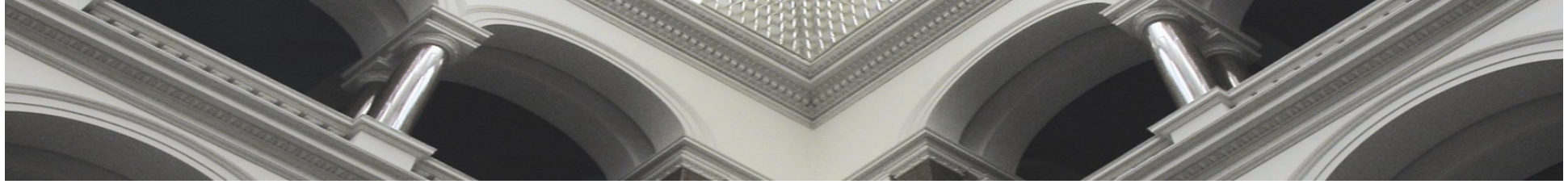
iterations

Block B	Out[B]^0	In[B]^1	Out[B]^1	In[B]^2	Out[B]^2
B_1	000 0000	000 0000	111 0000	000 0000	111 0000
B_2	000 0000	**111 0000**	001 1100	**111 0111**	001 1110
B_3	000 0000	**001 1100**	000 1110	**001 1110**	000 1110
B_4	000 0000	**001 1110**	001 0111	001 1110	001 0111
EXIT	000 0000	**001 0111**	001 0111	001 0111	001 0111



Exercise

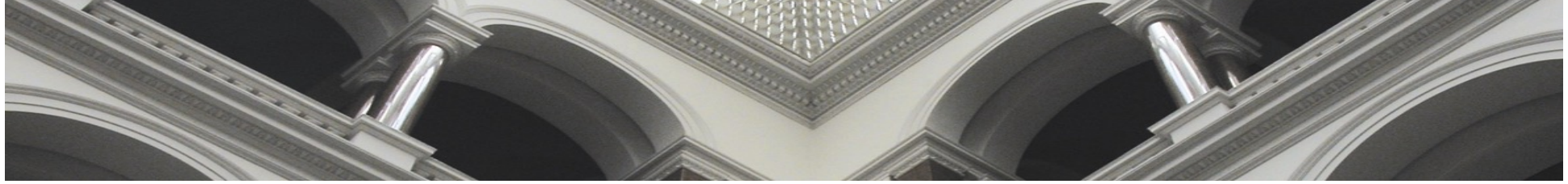




Traversal Order

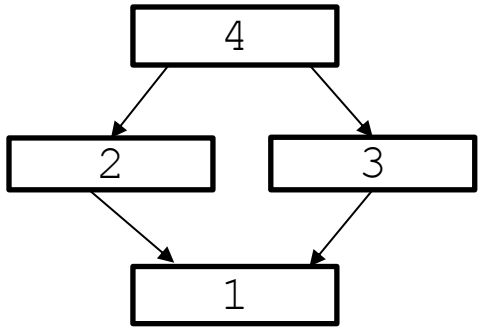
Ordering affects performance
(not convergence!)

- Random order
  - not aware whether the data-flow equations solve a forward or backward data-flow problem
  - performance is relatively poor compared to specialized iteration orders
- Postorder
  - iteration order for backward data-flow problems
  - a node is visited after all its successor nodes have been visited
  - typically implemented with the depth-first strategy
- Reverse postorder (RPO)
  - iteration order for forward data-flow problems
  - a node is visited before any of its successor nodes has been visited, except when the successor is reached by a back edge
    - Note that this is not the same as preorder!

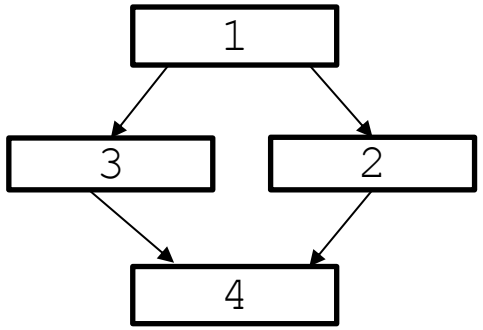


Traversal Order: RPO

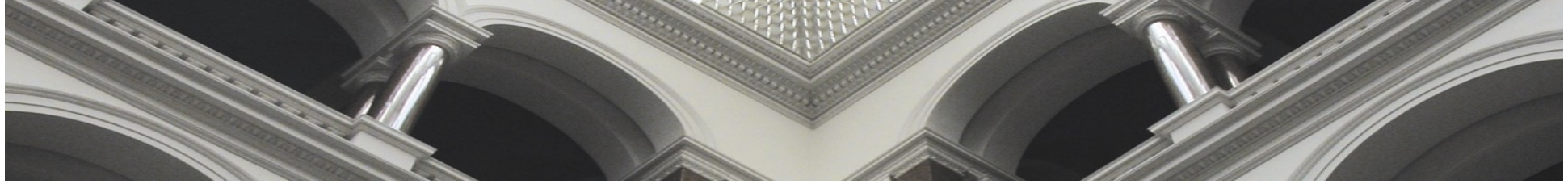
- Reverse CFG: The CFG with its edge reversed
- Reverse postorder is the postorder on the reverse CFG



Postorder



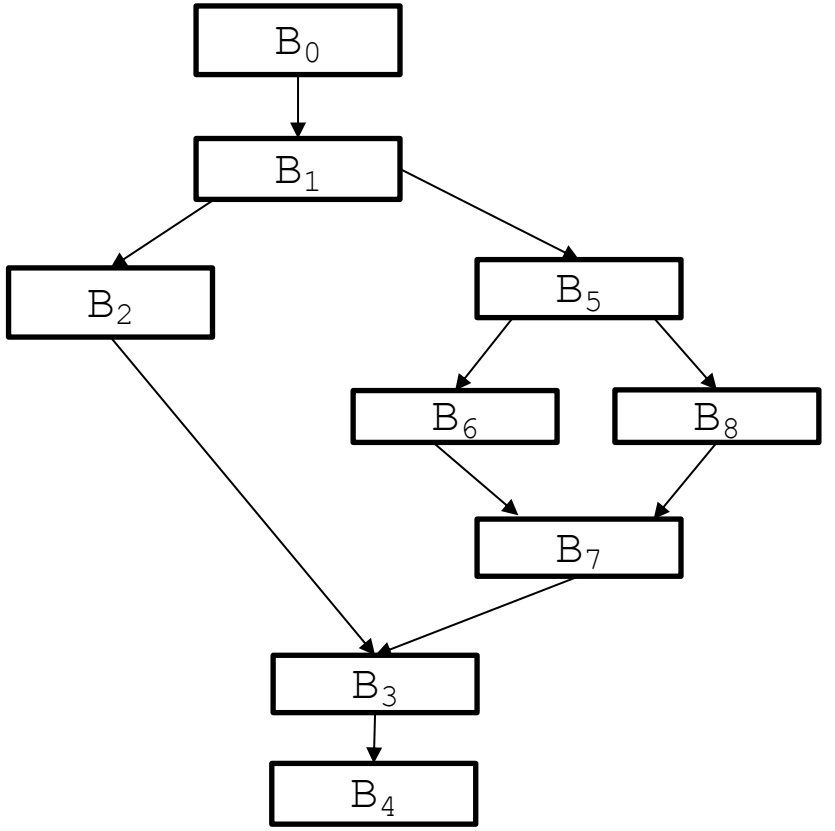
Reverse Postorder



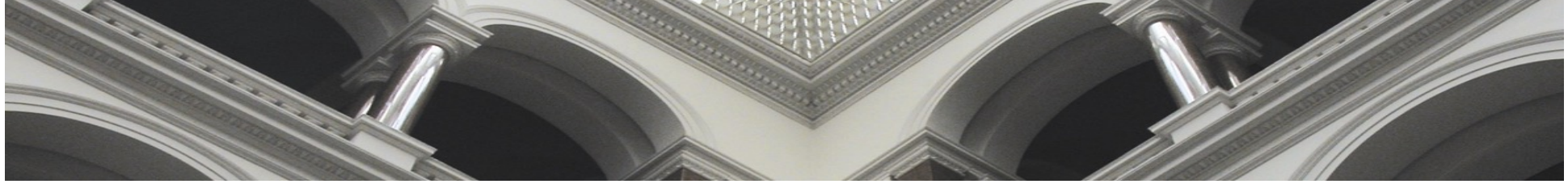
Exercise

- Enumerate the BB of the CFG by

1. Post order
2. Reverse Postorder



Source: Engineering a Compiler, K. Cooper & L. Torczon, p. 481

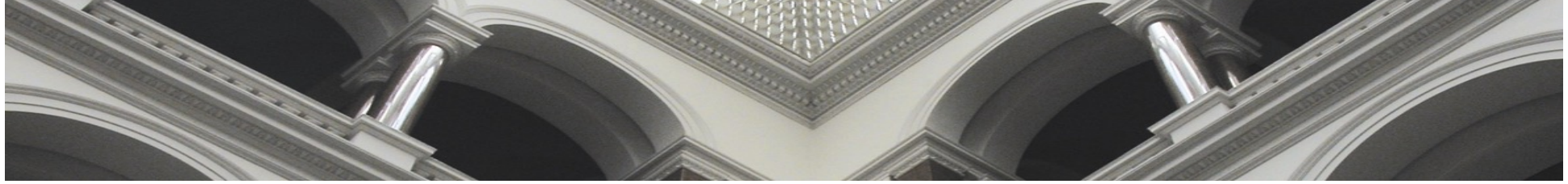


Towards a General Data-Flow Framework

- A data-flow analysis framework $(D, V, \wedge, F)$

1. Direction D (forwards or backwards)
2. Semilattice, which includes a domain of values V and a meet operator $\wedge$
3. A family of transfer functions F from V to V

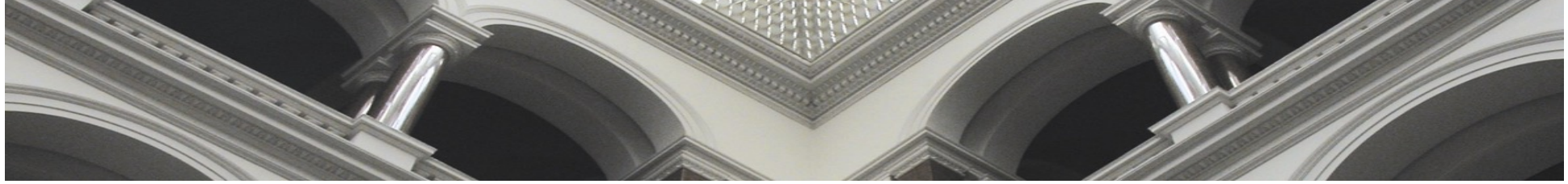
   ➢ must include functions suitable for boundary conditions, which are constant transfer functions for the special nodes ENTRY and EXIT in any flow graph



Semilattice (or *Meet Semilattice*)

- A semilattice is a set V and a binary meet operator $\wedge$ such that for all x, y and z:
  1. $x \wedge x = x$ (meet is idempotent)
  2. $x \wedge y = y \wedge x$ (meet is commutative)
  3. $x \wedge (y \wedge z) = (x \wedge y) \wedge z$ (meet is associative)
- A semilattice has a top element denoted as T, such that for all x in V, $\top \wedge x = x$
- (Optionally) a bottom element, denoted as such that for all x in V, $\bot \wedge x = \bot$

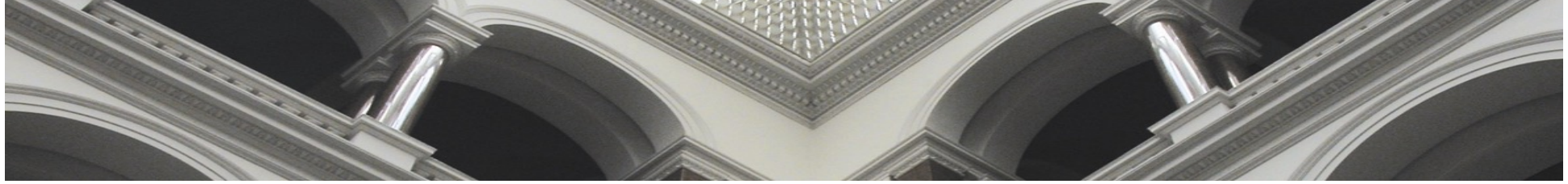
What about union and intersection?



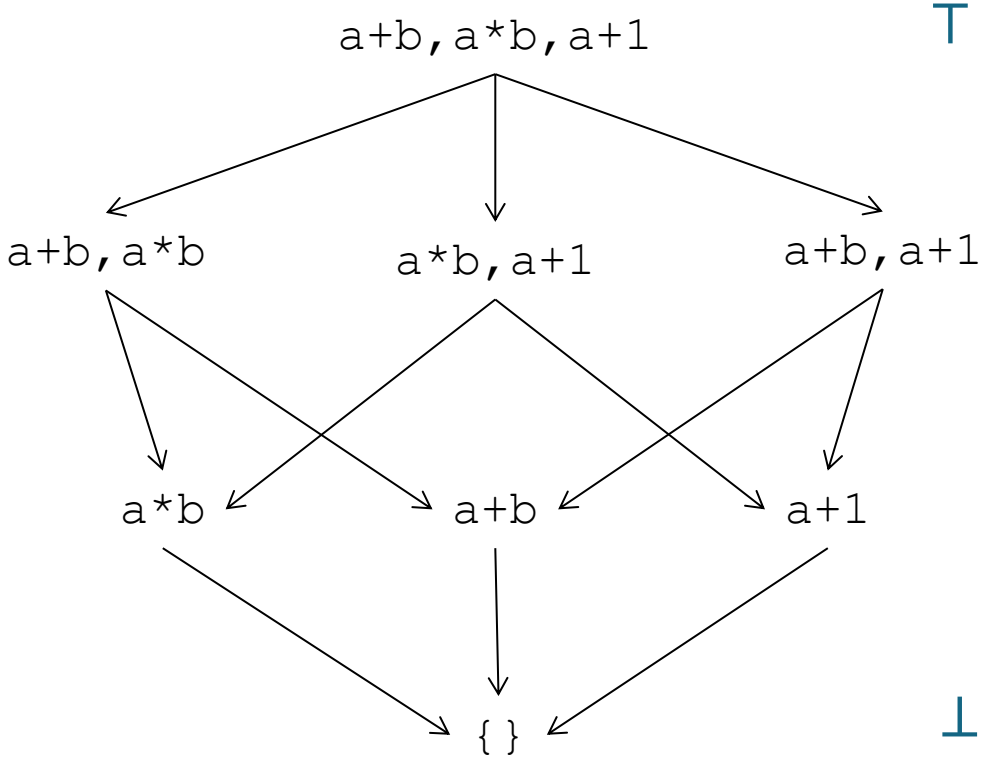
Partial Orders for Semilattice

- The meet operator defines a partial order on the values of the domain
- A relation ≤ is a partial order on a set V if for all x, y and z in V:
  - $x \leq x$ (the partial order is reflexive)
  - if $x \leq y$ and $y \leq x$, then $x = y$ (the partial order is antisymmetric)
  - if $x \leq y$ and $y \leq z$, then $x \leq z$ (the partial order is transitive)
- We can define a partial order $\leq$ for a semilattice $(V, \wedge)$
  - for all x and y in V we define

$$x \leq y \text{ if and only if } x \wedge y = x$$

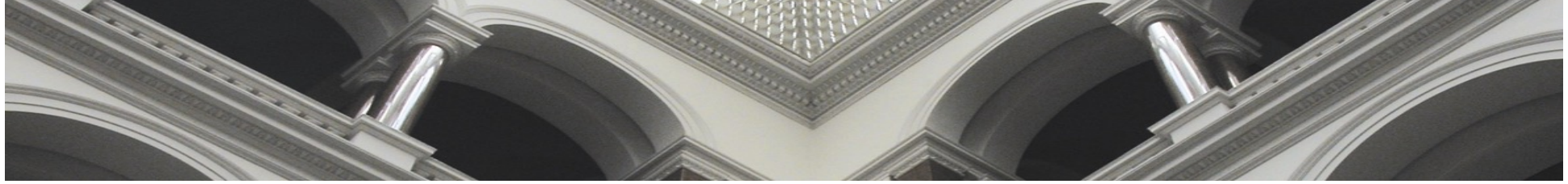


Example: Lattice Diagram for Available Expressions



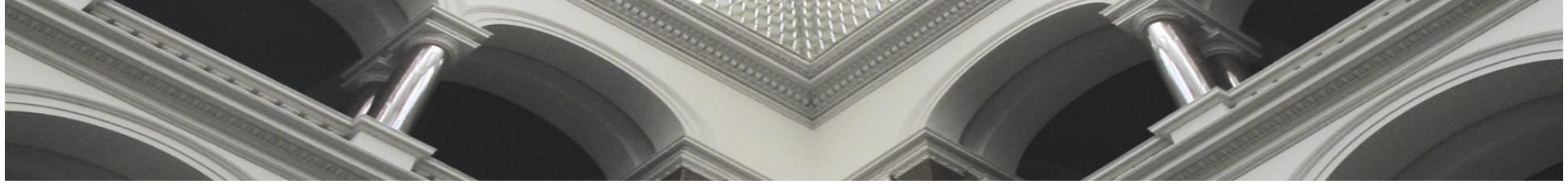
⊤ top

⊥ bottom



Transfer Function

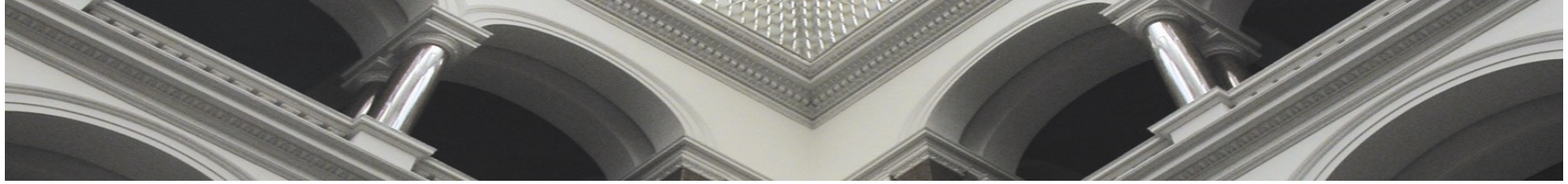
- Property of a family of transfer functions $F: V \rightarrow V$
  - F has an identity function I such that $I(x)=x$ for all x in V
  - F is closed under composition
    - for any other two function f and g in F, the function h defined by $h(x)=g(f(x))$ is in F
  - Example: in reaching definitions, when Gen and $Kill$ are both empty set
- Monotone framework $(D, V, \wedge, F)$
  - For all x and y in V and f in F, $x \leq y$ implies $f(x) \leq f(y)$
- Distributive framework (it applies the distributivity condition)
  - For all x and y in V and f in F, $f(x \wedge y) = f(x) \wedge f(y)$
  - Distributivity implies monotonicity, although the converse is not true



Technische Universität Berlin

Properties

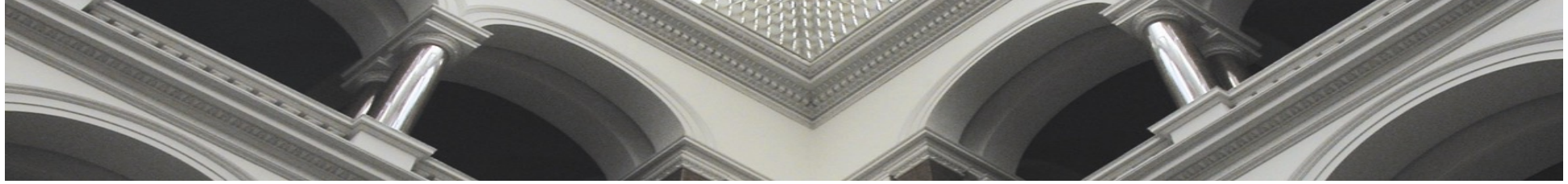
1. When the algorithm converges, the result is a solution to the data-flow equations
2. If the framework is monotone, then the solution found is the maximum fixedpoint (MFP) of the data-flow equations
   - a MFP is a solution with the property that in any other solution, the values of `IN[B]` and `OUT[B]` are $\leq$ the corresponding values of MFP
   - monotonicity ensures that on each iteration the value will either stay the same or will grow larger, while finite height ensures that it cannot grow indefinitely
3. If the semilattice of the framework is monotone and of finite height, then the algorithm is guaranteed to converge
   - e.g., there are no infinite ascending chains $x_1 \leq x_2 \leq \ldots$
   - the combination of the transfer function and the join operation should be monotonic with respect to this partial order



Technische Universität Berlin

Iterative Algorithm for General Frameworks

- INPUT
  1. A data-flow graph, with specially labelled ENTRY and EXIT nodes
  2. A direction of the data-flow `D`
  3. A set of values `V`
  4. A meet operator Λ
  5. A set of functions F, where f_B in F is the transfer function for block B
  6. A constant value v_{entry} or v_{exit} in V, representing the boundary condition for forward and backward frameworks, respectively
- OUTPUT: values for `In[B]` and `Out[B]` for each block B in the data-flow graph



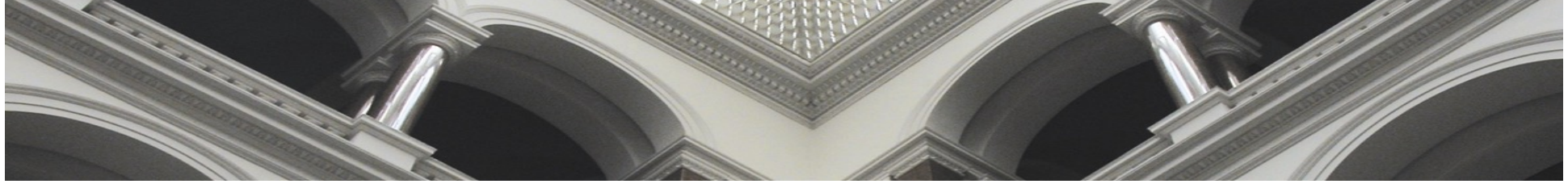
Iterative Algorithm for General Frameworks

- Forward data-flow problem

```
OUT[ENTRY] = v_ENTRY
for (each basic block B other than ENTRY)
  OUT[B] = T
while (changes to any OUT occur)
  for(each basic block B other than ENTRY)
  {
    IN[B] = ⋀_{P a predecessor of B} OUT[P]
    OUT[B] = f_B(IN[B])
  }
```

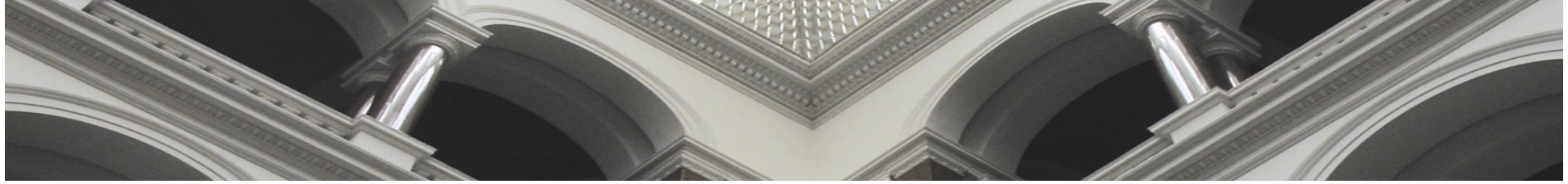
- Backward data-flow problem

```
IN[EXIT] = v_EXIT
for (each basic block B other than ENTRY)
  IN[B] = T
while (changes to any IN occur)
  for(each basic block B other than EXIT)
  {
    OUT[B] = ⋀_{S a successor of B} IN[S]
    IN[B] = f_B(OUT[B])
  }
```



Summary of Three Data-flow Problems

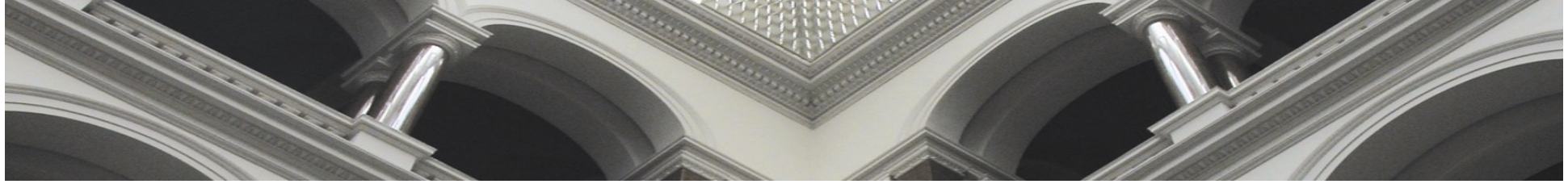
	Reaching definitions	Live Variables	Available Expressions
Domain	Sets of definitions	Sets of variables	Sets of expressions
Direction	Forward	Backwards	Forward
Transfer function	$gen_B \bigcup (x - kill_B)$	$\text{use}_B \bigcup (x - def_B)$	$e\_gen_B \bigcup (x - \text{e}\_kill_B)$
Boundary	$OUT[Entry] = \emptyset$	$IN[Exit] = \emptyset$	$OUT[Entry] = \emptyset$
Meet (∧)	Union	Union	Intersection
Equations	$OUT[B] = \text{f}_B(IN[B])$ $IN[B] = \bigwedge_{P\ in\ pred(B)} OUT[P])$	$IN[B] = \text{f}_B(OUT[B])$ $OUT[B] = \bigwedge_{P\ in\ succ(B)} IN[P]$	$OUT[B] = \text{f}_B(IN[B])$ $IN[B] = \bigwedge_{P\ in\ pred(B)} OUT[P]$
Initialize	$\text{OUT}[B] = \emptyset$	$\text{IN}[B] = \emptyset$	$OUT[B] = U$



Technische Universität Berlin

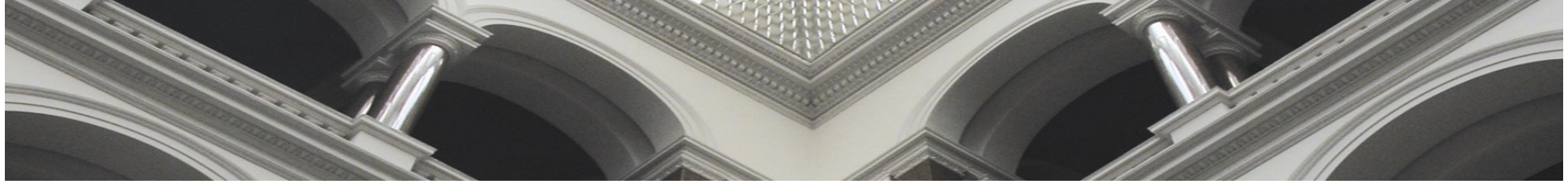
Other Data-Flow Problems

- Constant propagation
- Code motion
- Anticipated expressions
- Post-ponable expressions
- Used expressions
- Very busy expression



Exercise: Questions

1. Under what circumstances is the solution (of data-flow analysis) the fixed point?
2. How precise is the solution obtained by iterative algorithm?
3. Will the iterative algorithm converge?
4. What is the meaning of the solution to the equations?



Technische Universität Berlin

LLVM Passes

- We can build compiler passes at different levels
  - Basic block
  - Function
  - Loop
  - Region
  - Strongly-connected components
- Warning
  - LLVM Entry (and Exit) block is not a separate block, such as in [ALSU]
    - Entry (and Exit) block is **NOT** empty

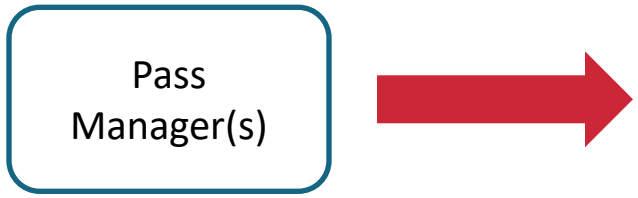
LLVM BasicBlock Pass

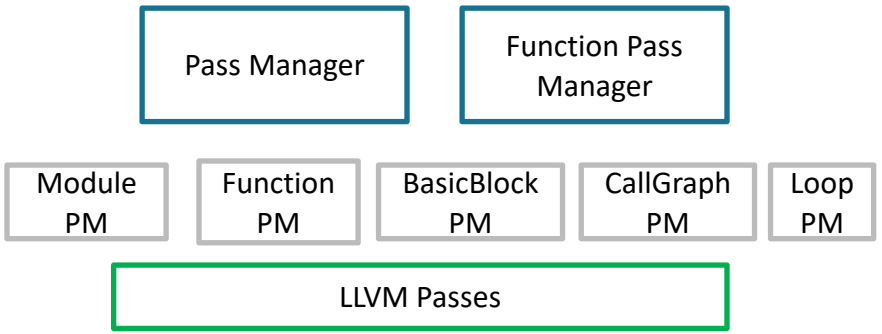
```
#include <llvm/IR/Type.h>
#include <llvm/Pass.h>
#include <llvm/IR/BasicBlock.h>
#include <llvm/Support/raw_ostream.h>
using namespace llvm;
namespace {
class MyBlockPass : public BasicBlockPass {
public:
    static char ID;
    MyBlockPass() : BasicBlockPass(ID) {}

    virtual bool runOnBasicBlock(BasicBlock &BB) {
      errs().write_escaped(BB.getName()) << '\n';
      // iterating instructions in the current BasickBlock
      for(Instruction &i : BB){
          errs() << " - "<< i.getOpcodeName() << " ";
          Type *type = i.getType();
          type->print(errs());
          errs() << '\n';
      }
      errs() << '\n';
      return false;
    }
  };
} // namespace

char MyBlockPass ::ID = 0;
static RegisterPass<MyBlockPass > X("block-pass", "Block Pass");
```

LLVM Function Pass

```
#include <llvm/Pass.h>
#include <llvm/IR/Function.h>
#include <llvm/Support/raw_ostream.h>
using namespace llvm;
namespace {
class MyFunctionPass  : public FunctionPass {
public:
    static char ID;
    MyFunctionPass  () : FunctionPass(ID) {}
    virtual bool runOnFunction(Function &F) {
        errs().write_escaped(F.getName()) << '\n';
        // iterate arguments
        errs() << " - args:" << '\n';
        for(Argument &a : F.getArgumentList()){
            errs() << "   - ";
            errs().write_escaped(a.getName()) << '\n';
        }
        // iterate BB in a function
        errs() << " - blocks:" << '\n';
        for(BasicBlock &b : F.getBasicBlockList()){
            errs() << "   * ";
            errs().write_escaped(b.getName()) << '\n';
        }
        errs() << '\n';
        return false;
    }
};
} // namespace
char MyFunctionPass::ID = 0;
static RegisterPass<MyFunctionPass> X("function-pass", "FunctionPass");
```

LLVM Loop Pass






```
#include <llvm/Analysis/LoopPass.h>
#include <llvm/Support/raw_ostream.h>
using namespace llvm;

namespace {

class MyLoopPass : public LoopPass {
public:
    static char ID;
    MyLoopPass() : LoopPass(ID) {}

    bool runOnLoop(Loop *L, LPPassManager &LPPM) {
      L->print(errs());
      for(BasicBlock *b : L->getBlocks()) {
          errs() << " - ";
          errs().write_escaped(b->getName()) << '\n';
      }
      errs() << '\n';
      return false;
    }
};
}// namespace

char MyLoopPass::ID = 0;
static RegisterPass<MyLoopPass> X("loop-pass", "Loop Pass");
```


AES
Embedded Systems Architecture

LLVM Region Pass

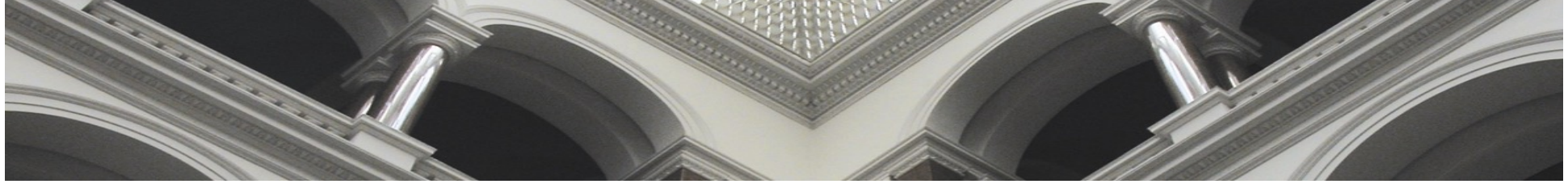
The analysis does not transform the input at all

```
#include <llvm/Analysis/RegionPass.h>
#include <llvm/Support/raw_ostream.h>
using namespace llvm;
namespace {
class MyRegionPass : public RegionPass {
public:
    static char ID;
    MyRegionPass() : RegionPass(ID) {}
    virtual void getAnalysisUsage(AnalysisUsage &au) const {
        au.setPreservesAll();
    }
    virtual bool runOnRegion(Region *R, RGPassManager &RGM){
        errs() << "region ";
        R->print(errs());
        // iterating region nodes in the current region
        errs() << " - subregions:";
        errs() << "\n";
        for(Region *sub : *R){
            errs() << " * ";
            sub->print(errs());
        }
        errs() << '\n';
        return false;
    }
};
}// namespace
char MyRegionPass::ID = 0;
static RegisterPass<MyRegionPass> X("region-pass", "Region Pass");
```

LLVM SCC Pass

For what optimization SCC can be useful?

```
#include <llvm/Pass.h>
#include <llvm/Analysis/CallGraphSCCPass.h>
#include <llvm/Support/raw_ostream.h>
using namespace llvm;
namespace {
class MySCCPass : public CallGraphSCCPass {
public:
    static char ID;
    int sccNum;
    MySCCPass() : CallGraphSCCPass(ID), sccNum(0) {}
    virtual bool runOnSCC(CallGraphSCC &SCC){
        errs() << "SCC " << sccNum++ << '\n';
        // iterating graph nodes in the current SCC
        for(CallGraphNode *cgn : SCC){
            errs() << " - ";
            cgn->print(errs());
        }
        return false;
    }
};
}// namespace
char MySCCPass::ID = 0;
static RegisterPass<MySCCPass> X("scc-pass", "SCC Pass");
```



Technische Universität Berlin

Summary

- Control Flow Graph, Basic block
- Data-flow analysis
- Available expressions, live variables, reaching definitions
- Abstraction of dataflow analysis
- Framework for dataflow analysis: algorithm and properties
- LLVM passes

- Readings
  - ALSU 9.1, 9.2, 9.3
- Suggested course
  - Sabine Glesner: Analysis and Optimisation of Embedded Systems [SoSe]