Compiler Optimization and Code Generation

Professor: Sc.D., Professor
Vazgen Melikyan

Course Overview

- Introduction: Overview of Optimizations
  - 1 lecture
- Intermediate-Code Generation
  - 2 lectures
- Machine-Independent Optimizations
  - 3 lectures
- Code Generation
  - 2 lectures

Machine-Independent Optimizations

Causes of Redundancy

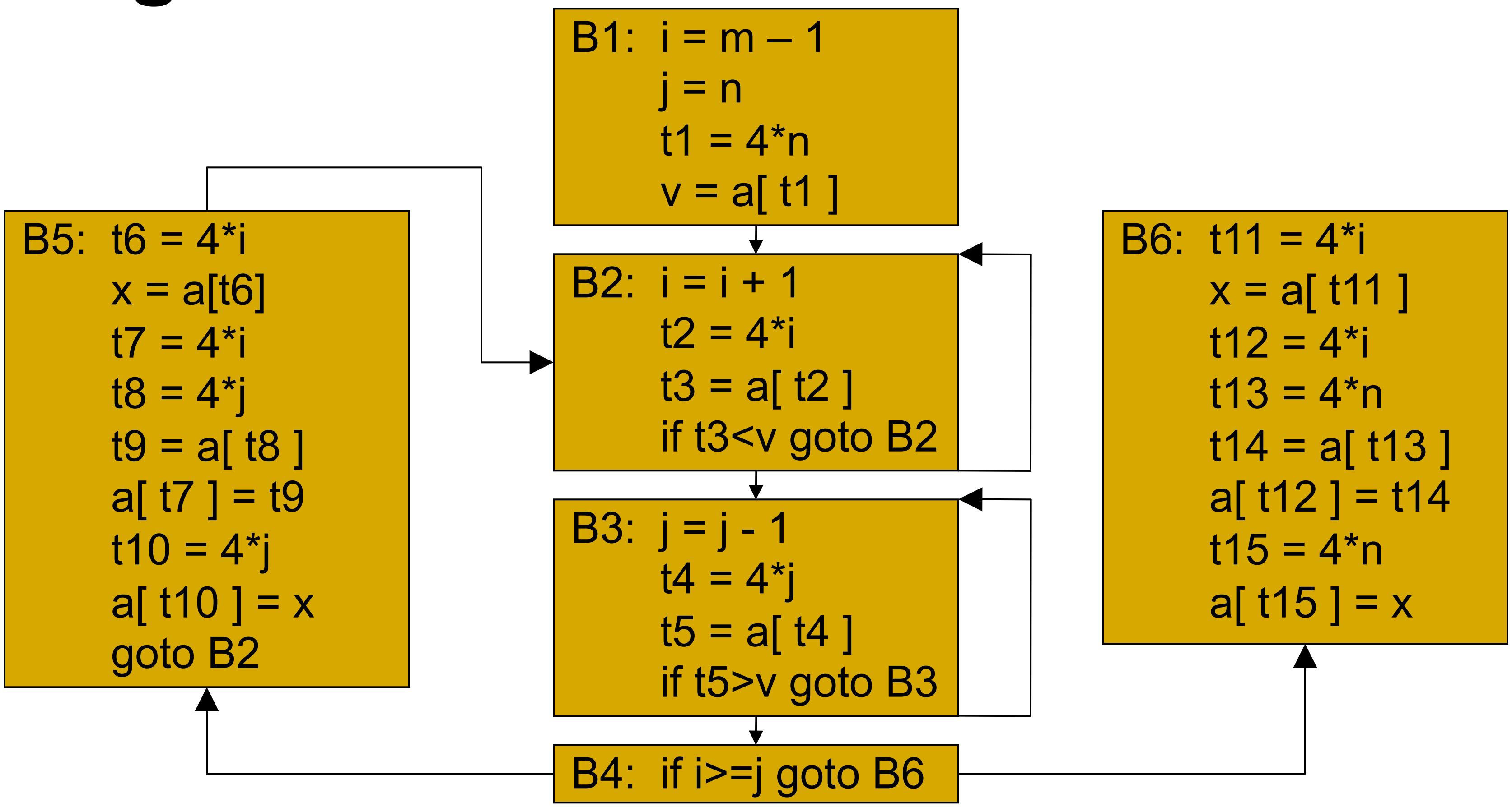
- Redundancy is available at the source level due to recalculations while one calculation is necessary.
- Redundancies in address calculations
  - Redundancy is a side effect of having written the program in a high-level language where referrals to elements of an array or fields in a structure is done through accesses like A[i][j] or X -> f1.

  As a program is compiled, each of these high-level data-structure accesses expands into a number of low-level arithmetic operations, such as the computation of the location of the [i, j]-th element of a matrix A. Accesses to the same data structure often share many common low-level operations.

A Running Example: Quicksort

```
void quicksort (int m, int n)
    /* recursively sorts a[ m ] through a[ n ] */
    {
            int i , j, v, x;
            if (n <= m) return;
            /* fragment begins here */
            i = m - 1; j = n; v = a[ n ];
            while (1) {
                    do i = i + 1; while (a[ i ] < v);
                    do j = j - 1; while (a[ j ] > v);
                    if ( i >= j ) break;
                    x = a [ i ]; a[ i ] = a [ j ] ; a [ j ] = x;    /* swap a [ i ] , a[ j ] */
            }
            x = a [ i ]; a[ i ] = a[ n  ]; a[ n ] = x;  /* swap a[ i ] , a[ n ] */
            /* fragment ends here */
            quicksort ( m, j ); quicksort ( i + 1 ,n );
    }
```

Flow Graph for The Quicksort Fragment



Semantics-Preserving Transformations

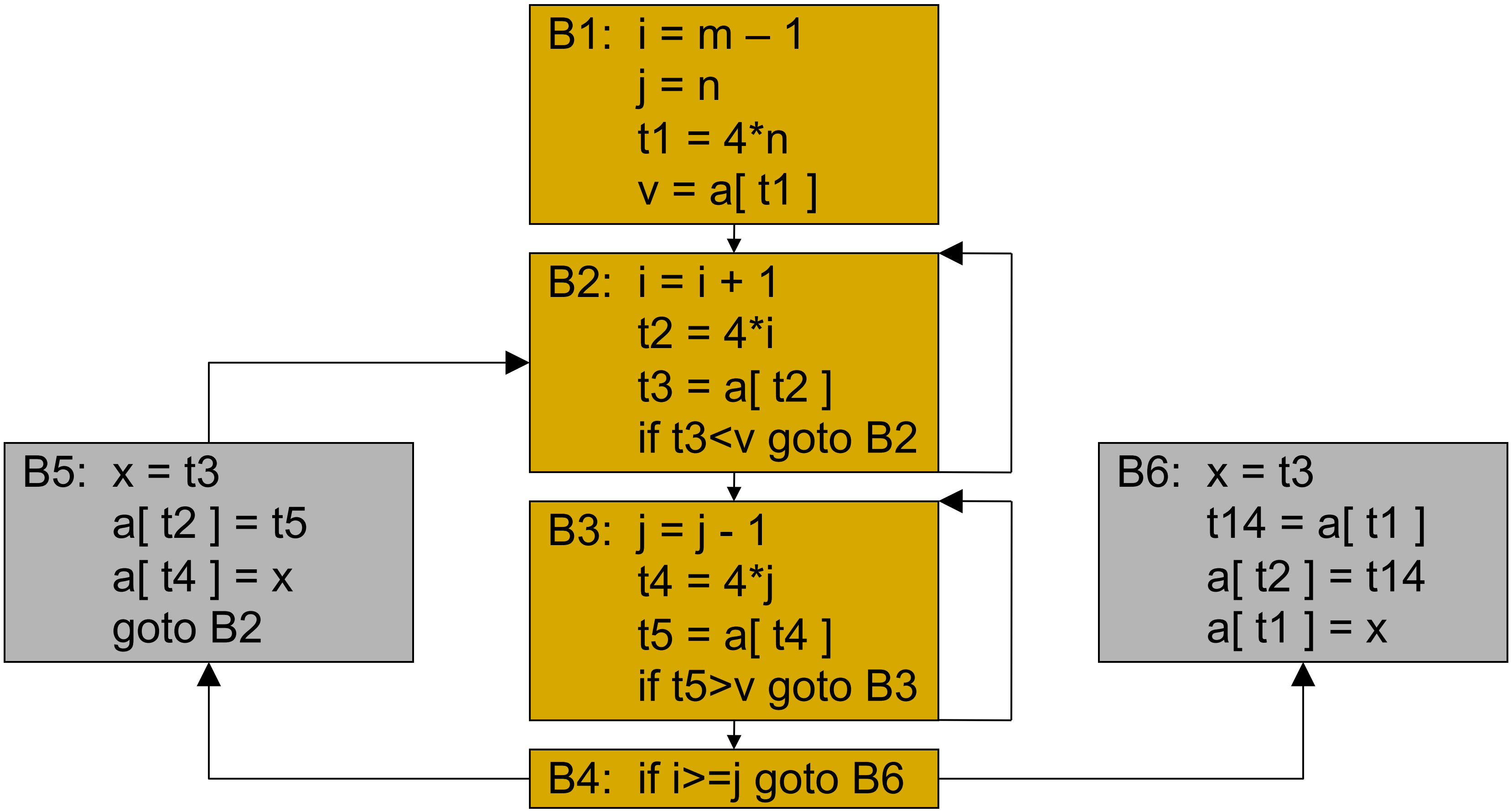
- There are a number of ways in which a compiler can improve a program without changing the function it computes.
  - Common-subexpression elimination
  - Copy propagation
  - Dead-code elimination
  - Constant folding
  - Code motion
  - Induction-variable elimination

Common-Subexpression Elimination

- An occurrence of an expression E is called a common subexpression if E was previously computed and the values of the variables in E have not changed since the previous computation.

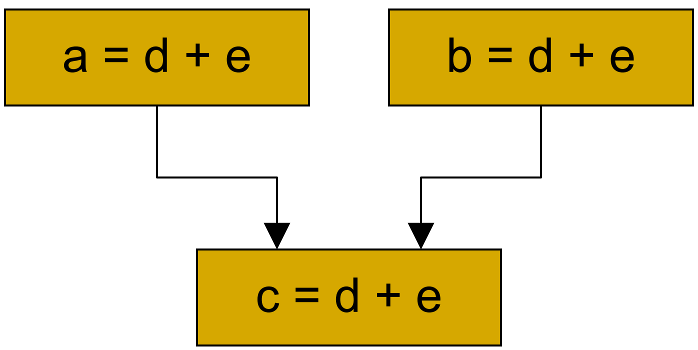
  Avoid recomputing E if can be used its previously computed value; that is, the variable x to which the previous computation of E was assigned has not changed in the interim.

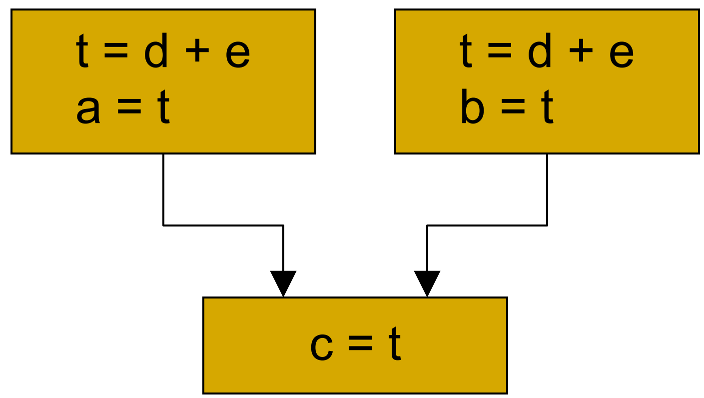
Flow Graph After C.S. Elimination



Copy Propagation (1)

- This optimization concerns assignments of the form u = v called copy statements.
- The idea behind the copy-propagation transformation is to use v for u, wherever possible after the copy statement u = v.
- Copy propagation work example:





Copy Propagation (2)

- The assignment x = t3 in block B5 is a copy.
  Here is the result of copy propagation applied to B5.

```
B5:  x = t3                B5:  x = t3
     a[ t2 ] = t5               a[ t2 ] = t5
     a[ t4 ] = x        ⇒       a[ t4 ] = t3
     goto B2                    goto B2
```

- This change may not appear to be an improvement, but it gives the opportunity to eliminate the assignment to x.
- One advantage of copy propagation is that it often turns the copy statement into dead code.

Dead-code Elimination

- Code that is unreachable or that does not affect the program (e.g. dead stores) can be eliminated.

  While the programmer is unlikely to introduce any dead code intentionally, it may appear as the result of previous transformations.
- Deducing at compile time that the value of an expression is a constant and using the constant instead is known as constant folding.

Dead-code Elimination: Example

- In the example below, the value assigned to i is never used, and the dead store can be eliminated. The first assignment to global is dead, and the third assignment to global is unreachable; both can be eliminated.

int global;
void f ()
{
    int i;
    i = 1; /* dead store */
    global = 1; /* dead store */
    global = 2;
    return;
    global = 3; /* unreachable */
}
	int global;
	void f ()
	{
	    global = 2;
	    return;
	}

Before and After Dead Code Elimination

Code Motion

- Code motion decreases the amount of code in a loop.

  This transformation takes an expression that yields the same result independent of the number of times a loop is executed (**a loop-invariant computation**) and evaluates the expression before the loop.

  Evaluation of limit - 2 is a loop-invariant computation in the following while-statement :

  ```
  while ( i <= limit-2)  /* statement does not change limit */
  ```

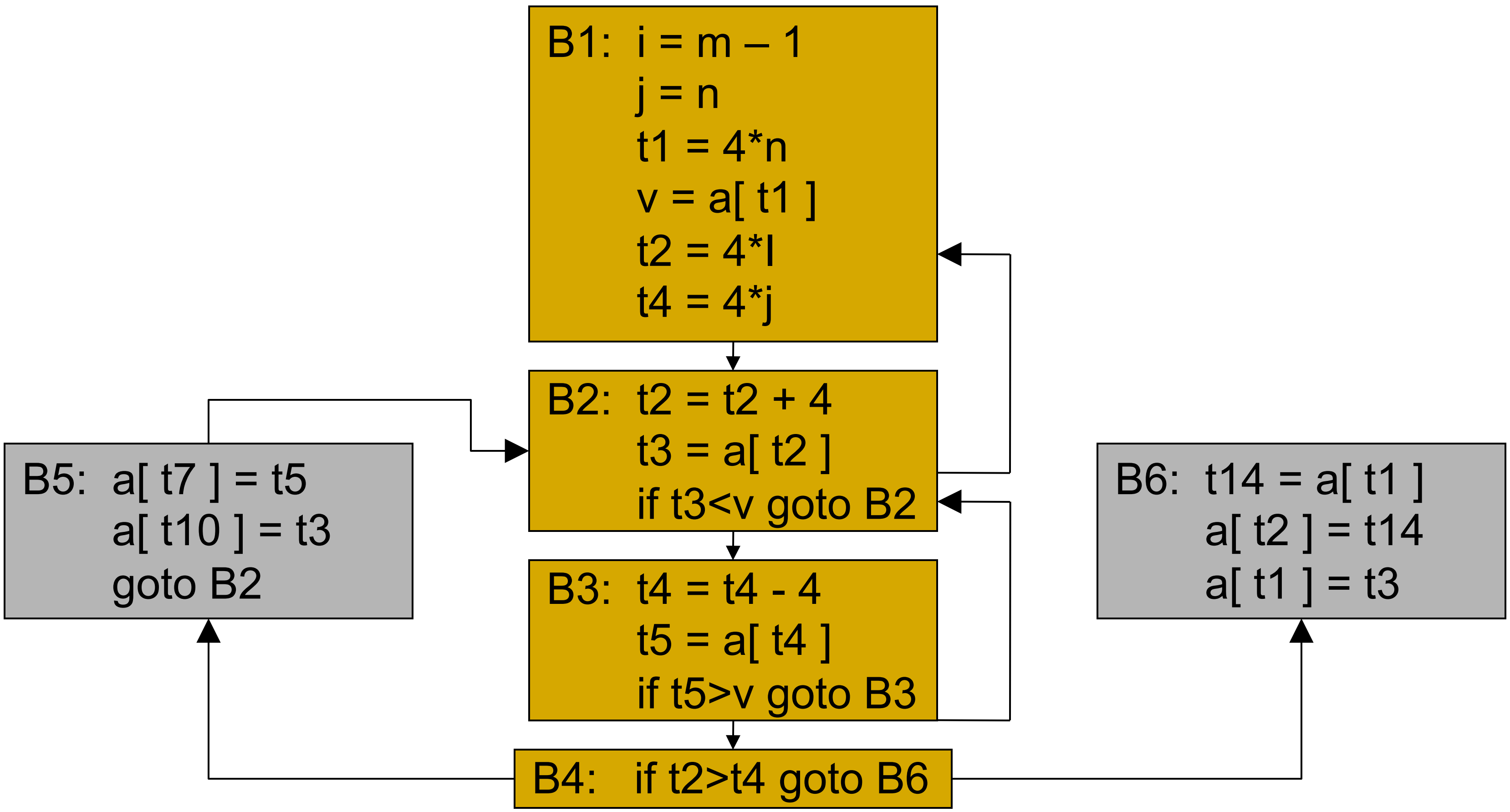
- Code motion will result in the equivalent code:

  ```
  t = limit-2;
  while ( i <= t )        /* statement does not change limit or t */
  ```

Induction-Variable (IV) Elimination

- Variable x is said to be an "induction variable“ if there is a positive or negative constant c such that each time x is assigned, its value increases by c.
- For instance, i and t2 are induction variables in the loop containing B2 of QuickSort example.
- Induction variables can be computed with a single increment (addition or subtraction) per loop iteration. The transformation of replacing an expensive operation, such as multiplication, by a cheaper one, such as addition, is known as strength reduction.

Flow Graph After IV Elimination

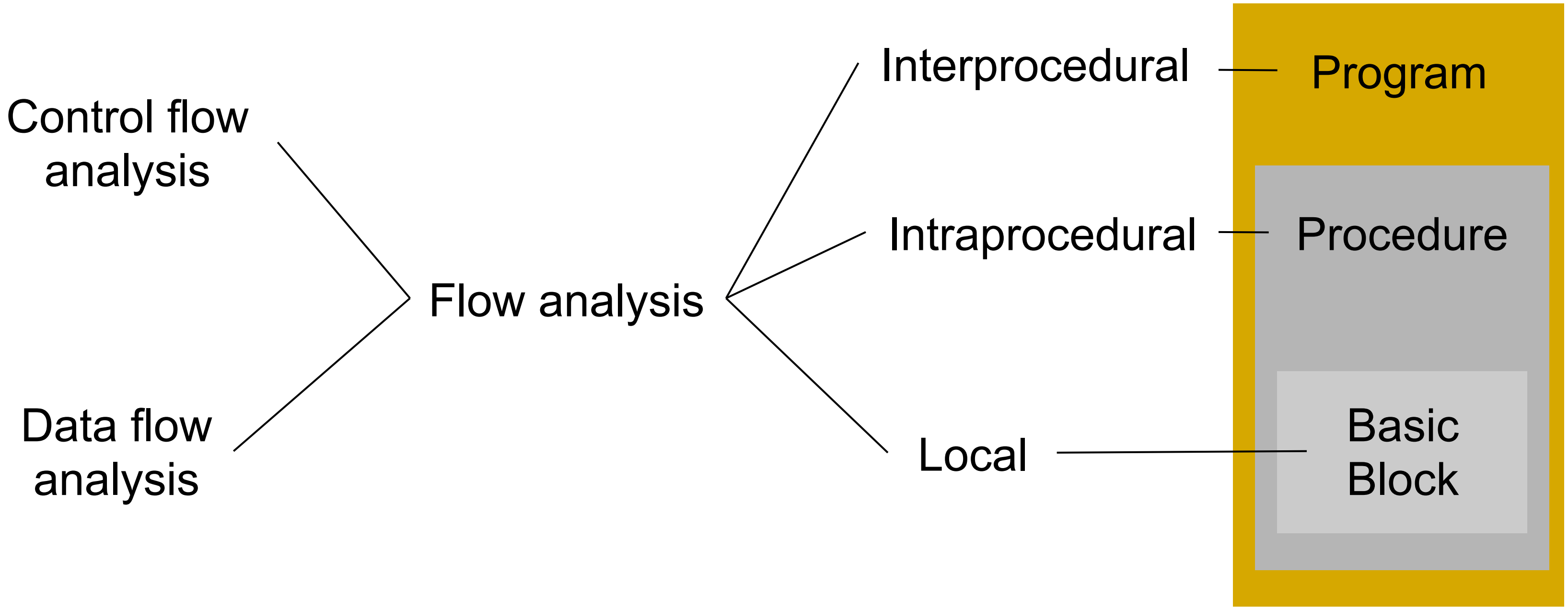


Flow Analysis

- Flow analysis is a fundamental prerequisite for many important types of code improvement.
- Generally control flow analysis precedes data flow analysis.
- Control flow analysis (CFA) represents flow of control usually in form of graphs. CFA constructs:
  - Control flow graph
  - Call graph
- Data flow analysis (DFA) is the process of asserting and collecting information prior to program execution about the possible modification, preservation, and use of certain entities (such as values or attributes of variables) in a computer program.

Classification of Flow Analysis

- Two orthogonal classifications of flow analysis:



- Interprocedural optimizations usually require a call graph.
- In a call graph each node represents a procedure and an edge from one node to another indicates that one procedure may directly call another.

Introduction to Data-Flow Analysis (1)

- "Data-flow analysis" refers to a body of techniques that derive information about the flow of data along program execution paths.
- All the previous optimizations depend on data-flow analysis.
  - Example 1: One way to implement global common subexpression elimination requires us to determine whether two textually identical expressions evaluate to the same value along any possible execution path of the program.
  - Example2: If the result of an assignment is not used along any subsequent execution path, then the assignment can be eliminated as dead code.

Data Flow Problems

- Reaching definitions
  - Determine the set of variable definitions that can reach a CFG node, i.e. the definition occurs at least on one path prior to that node (forward problem).
- Available expressions
  - Determine the set of expressions that are available at a CFG node, i.e. the expression is evaluated on all paths prior to that node (forward problem).
- Live/dead variables
  - Determine the set of variables that are live at a CFG node, i.e. the variable is used after control passes that node at least on one path; if the variable is not used, it is called dead (backward problem).

Introduction to Data-Flow Analysis (2)

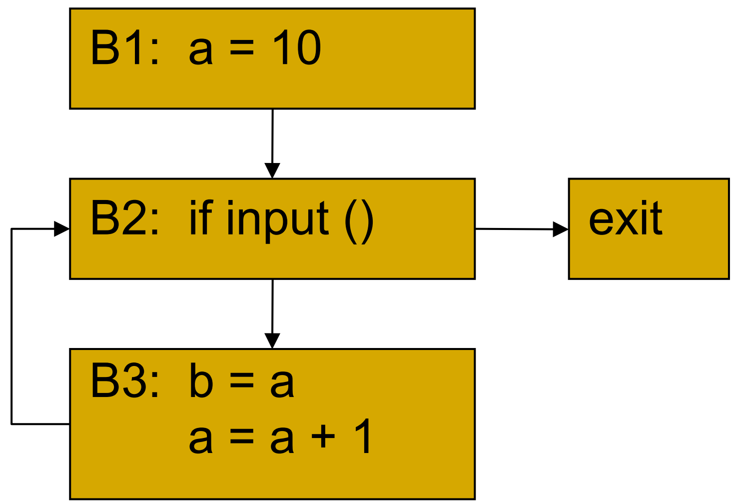
- Local analysis (e.g. value numbering)
  - Analyze effect of each instruction
  - Compose effects of instructions to derive information from beginning of basic block to each instruction
- Data flow analysis
  - Analyze effect of each basic block
  - Compose effects of basic blocks to derive information at basic block boundaries
  - From basic block boundaries, apply local technique to generate information on instructions

The Data-Flow Abstraction

- The execution of a program is viewed as a series of transformations of the program state, which consists of the values of all the variables in the program.
- Each execution of an intermediate-code statement transforms an input state to a new output state.

  The input state is associated with the program point before the statement and the output state is associated with the program point after the statement.
- While analyzing the behavior of a program, all the possible sequences of program points ("paths") through a flow graph must be considered. For the particular data-flow analysis problem an extraction from the possible program states at each point is used.

Static Program vs. Dynamic Execution



- **Statically**: Finite program
- **Dynamically**: Can have infinitely many possible execution paths
- Data flow analysis abstraction:
  - For each point in the program information of all the instances of the same program point are combined.

The Execution Path

- Execution path (or just path) from point p1 to point p2, is a sequence of points $p_1, p_2, p_3, \ldots p_n$ such that for each $i = 1, 2, \ldots n-1$ either
  - p_i is the point immediately preceding a statement and $p_i + 1$ is the point immediately following that same statement, or
  - p_i is the end of some block and $p_i + 1$ is the beginning of a successor block.
- In general, there is an infinite number of possible execution paths through a program, and there is no finite upper bound on the length of an execution path.
- Program analyses summarize all the possible program states that can occur at a point in the program with a finite set of facts.

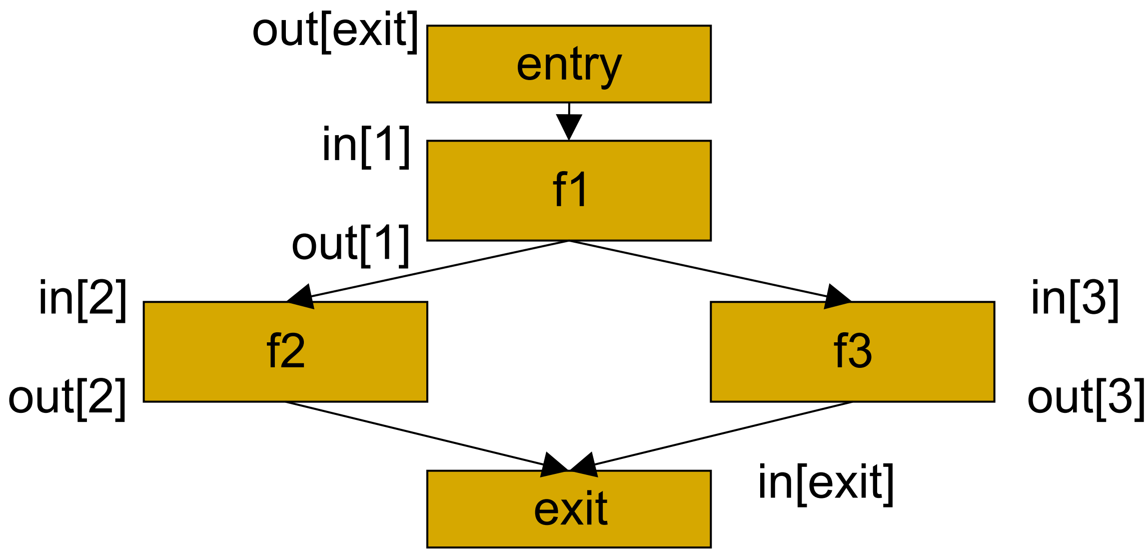
The Data-Flow Analysis Scheme (1)

- In each application of data-flow analysis, every program point is associated with value that represents an abstraction of the set of all possible program states that can be observed for that point. The set of possible data-flow values is the domain for this application.
- A particular data-flow value is a set of definitions. The choice of abstraction depends on the goal of the analysis.
- Data-flow values are denoted before and after each statement s by $IN[s]$ and $OUT[s]$ respectively. The data-flow problem is to find a solution to a set of constraints on the $IN[s]$s and $OUT[s]$'s, for all statements s. There are two sets of constraints: those based on the semantics of the statements and those based on the flow of control.

Synopsys University Courseware

Compiler Optimization and Code Generation
Lecture - 3
Developed By: Vazgen Melikyan

The Data-Flow Analysis Scheme (2)



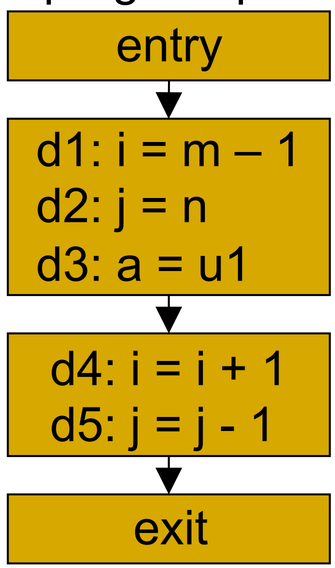
- Build a flow graph (nodes = basic blocks, edges = control flow)
- Set up a set of equations between $\mathrm{in}[b]$ and $\mathrm{out}[b]$ for all basic blocks b
  - Effect of code in basic block:
    - Transfer function f_b relates $\mathrm{in}[b]$ and $\mathrm{out}[b]$ for same b
  - Effect of flow of control:
    - Relates $\mathrm{out}[b1]$, $\mathrm{in}[b2]$ if b_1 and b_2 are adjacent

Effects of a Basic Block

- Effect of a statement: a = b+c
  - Uses variables (b, c)
  - Kills an old definition (old definition of a)
  - New definition (a)
- Compose effects of statements - effect of a basic block
  - Any definition of a data item in the basic block kills all definitions of the same data item reaching the basic block.
  - A locally available definition = last definition of data item in basic block

Reaching Definitions

- A definition d reaches a point p if there exists path from the point immediately following d to p such that d is not killed (overwritten) along that path.
- Problem statement
  - For each point in the program, determine if each definition in the program reaches the point
  - A bit vector per program point vector length = #def



gen (b1) = { d1 , d2 , d3 }

kill (b1) = { d4 , d5 }

gen (b1) = { d4 , d5 }

kill (b1) = { d1 , d2 }

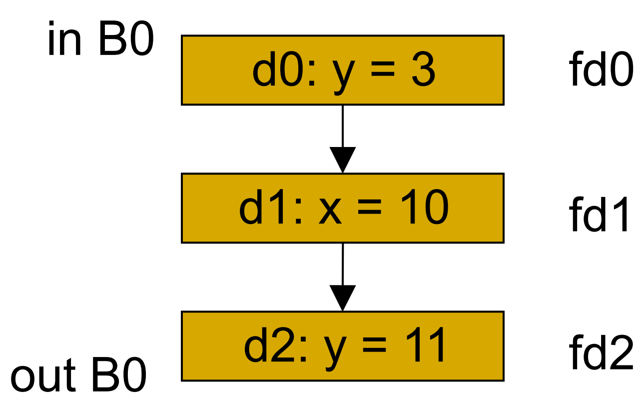
Effects of a Statement

- f_s: A transfer function of a statement
  - Abstracts the execution with respect to the problem of interest
- For a statement s(d: x = y + z)

$$out[s] = f_s(in[s]) = Gen[s] \cup (in[s] - Kill[s])$$

  - Gen[s]: definitions generated: $Gen[s] = \{d\}$
  - Propagated definitions: $in[s] - Kill[s]$,

    where $Kill[s] =$ set of all other defs to x in the rest of program



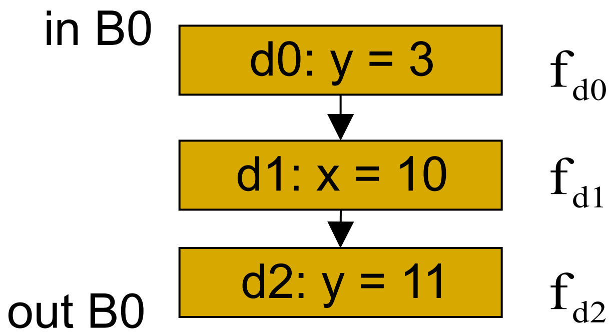
Effects of a Basic Block

- Transfer function of a statement s:
  - $out[s] = f_s(in[s]) = Gen[s] \cup (in[s] - Kill[s])$
- Transfer function of a basic block B:
  - Composition of transfer functions of statements in B
- $out[B] = f_B(in[B]) = f_{d2} f_{d1} f_{d0}(in[B])$

$$= Gen[d_2] \cup (Gen[d_1] \cup (Gen[d_0] \cup (in[B] - Kill[d_0])) - Kill[d_1])) - Kill[d_2]$$

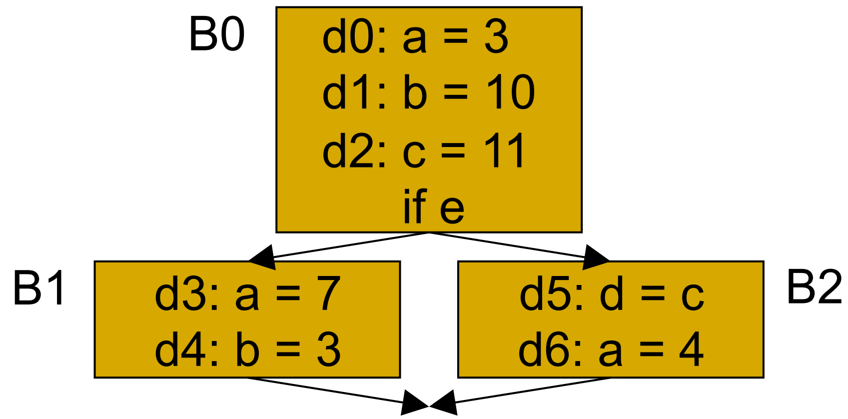
$$= Gen[B] \cup (in[B] - Kill[B])$$

  - Gen[B]: locally exposed definitions (available at end of b.b.)
  - Kill[B]: set of definitions killed by B



$$f_B = f_{d2} f_{d1} f_{d0}$$

Effects of a Basic Block: Example

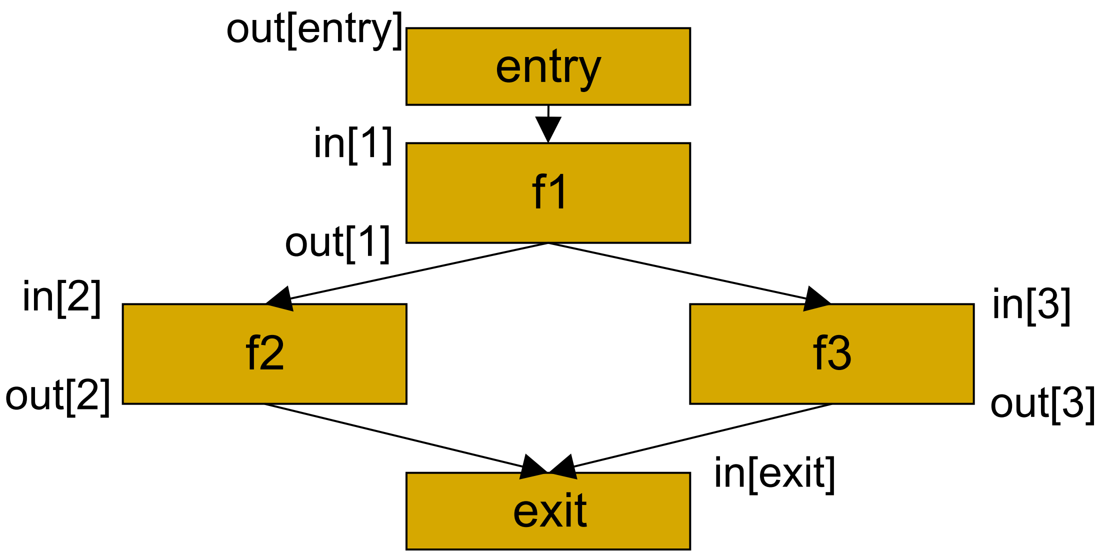


- A transfer function f_b of a basic block b:

  $$out[b] = f(in[b])$$

  incoming reaching definitions $\Longrightarrow$ outgoing reaching definitions
- A basic block b
  - Generates definitions: Gen[b],
    - Set of locally available definitions in b
  - Kills definitions: in[b] - Kill[b],
    - Where Kill[b]=set of defs (in rest of program) killed by defs in b
- $out[b] = Gen[b] \bigcup (in[b] - Kill[b])$

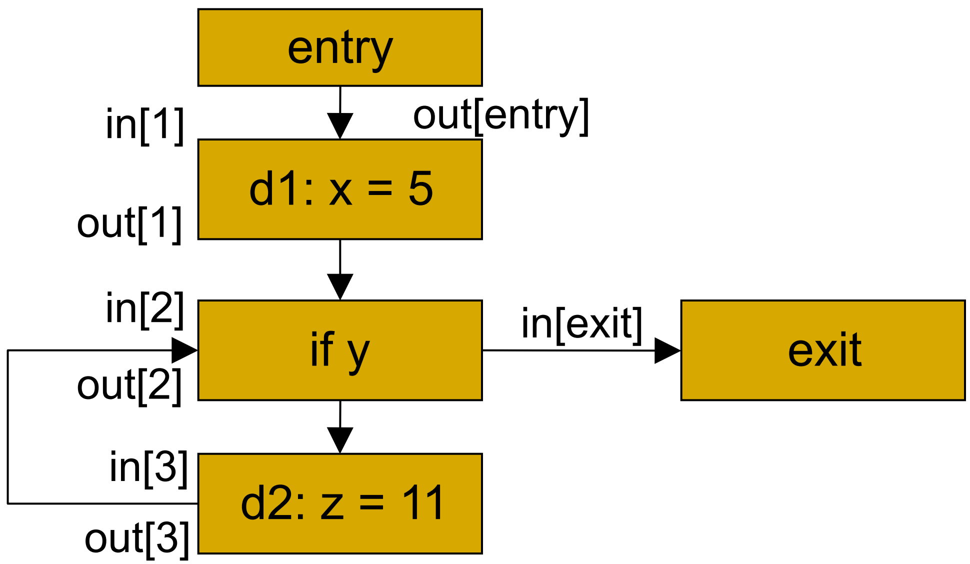
Effects of Edges: Acyclic



f	Gen	Kill
1	{1,2}	{3,4,6}
2	{3,4}	{1,2,6}
3	{5,6}	{1,3}

- $\mathrm{out}[b] = f_b(in[b])$
- Join node: a node with multiple predecessors
- Meet operator:
  - $in[b] = out[p_1] \cup out[p_2] \cup \ldots \cup out[p_n]$, where $p_1, p_2 \ldots p_n$ are all predecessors of b

Cyclic Graphs



- Equations still hold
  - $out[b] = f_b(in[b])$
  - $in[b] = out[p_1] \cup out[p_2] \cup \ldots \cup out[p_n]$
- Find: fixed point solution

Iterative Algorithm to Reach Definitions

Input: control flow graph CFG = (N, E, Entry, Exit)

// Boundary condition
    out[Entry] = $\emptyset$

// Initialization for iterative algorithm
  For each basic block B other than Entry
    out[B] = $\emptyset$

// Iterate
  While (Changes to any out[] occur) {
    For each basic block B other than Entry {
      in[B] = $\bigcup$(out[p]) , for all predecessors p of B
      out[B] = f_B(in[B])   // out[B]=gen[B] $\bigcup$(in[B]-kill[B])
    }
  }

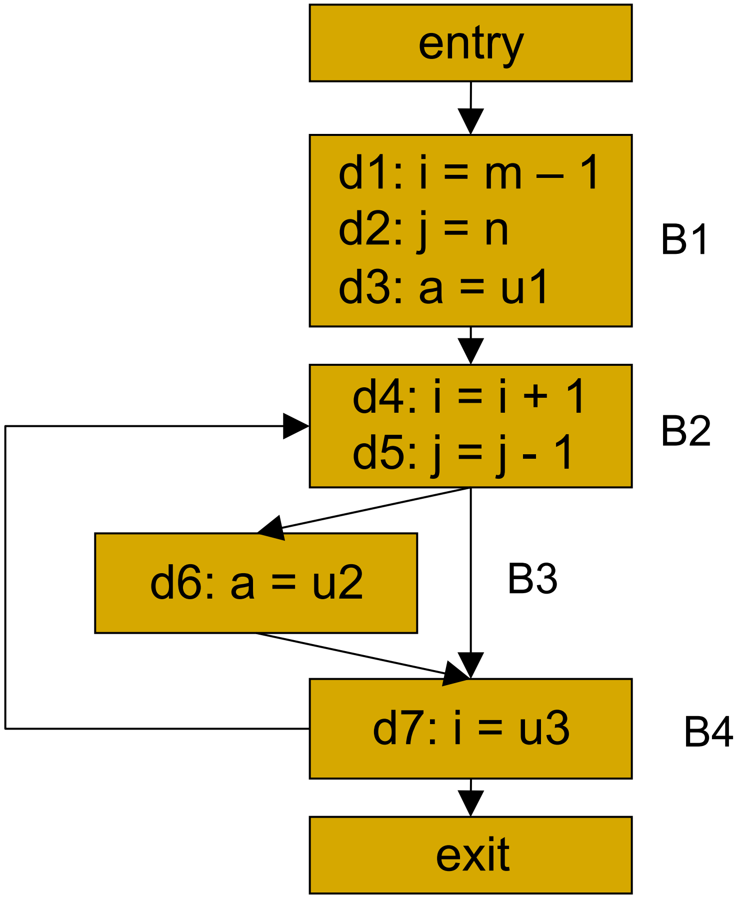
Synopsys University Courseware

Compiler Optimization and Code Generation
Lecture - 3
Developed By: Vazgen Melikyan

Reaching Definitions: Worklist Algorithm

```
Input: control flow graph CFG = (N, E, Entry, Exit)
// Initialize
    out[Entry] = ∅              // can set out[Entry] to special def
                                // if reaching then undefined use
    For all nodes i
        out[i] = ∅              // can optimize by out[i]=gen[i]
        ChangedNodes = N
// Iterate
    While ChangedNodes ≠ ∅  {
        Remove i from Changed Nodes
        in[i] = U (out[p]), for all predecessors p of i
        oldout = out[i]
        out[i] = fi(in[i])      // out[i]=gen[i] U (in[i]-kill[i])
        if (oldout ≠ out[i]) {
            for all successors s of i
                add s to Changed Nodes
        }
    }
```

Reaching Definitions: Example (1)



gen b1 = { d1 , d2 , d3 }

kill b1 = { d4 , d5 , d6, d7 }

gen b2 = { d4 , d5 }

kill b2 = { d1 , d2 , d7 }

gen b3 = { d6 }

kill b3 = { d3 }

gen b4 = { d7 }

kill b4 = { d1 , d4 }

Reaching Definitions: Example (2)

Block B	OUT[B]0	IN[B]1	OUT[B]1	IN[B]2	OUT[B]2
B1	000 0000	000 0000	111 0000	000 0000	111 0000
B2	000 0000	111 0000	001 1100	111 0111	001 1110
B3	000 0000	001 1100	000 1110	001 1110	000 1110
B4	000 0000	001 1110	001 0111	001 1110	001 0111
EXIT	000 0000	001 0111	001 0111	001 0111	001 0111

Computation of IN and OUT

Live Variable Analysis (1)

- **Definition**
  - A variable v is live at point p if the value of v is used along some path in the flow graph starting at p
  - Otherwise, the variable is **dead.**
- **Motivation**
  - E.g. register allocation

```
for i = 0 to n
    … i …
for i = 0 to n
    … i …
```

- **Problem statement**
  - For each basic block determine if each variable is live in each basic block
  - Size of bit vector: one bit for each variable

Live Variable Analysis (2)

control flow

$\mathrm{IN[b]} = \mathrm{f_b}(OUT[b])$

b $\mathrm{f_b}$

OUT[b]

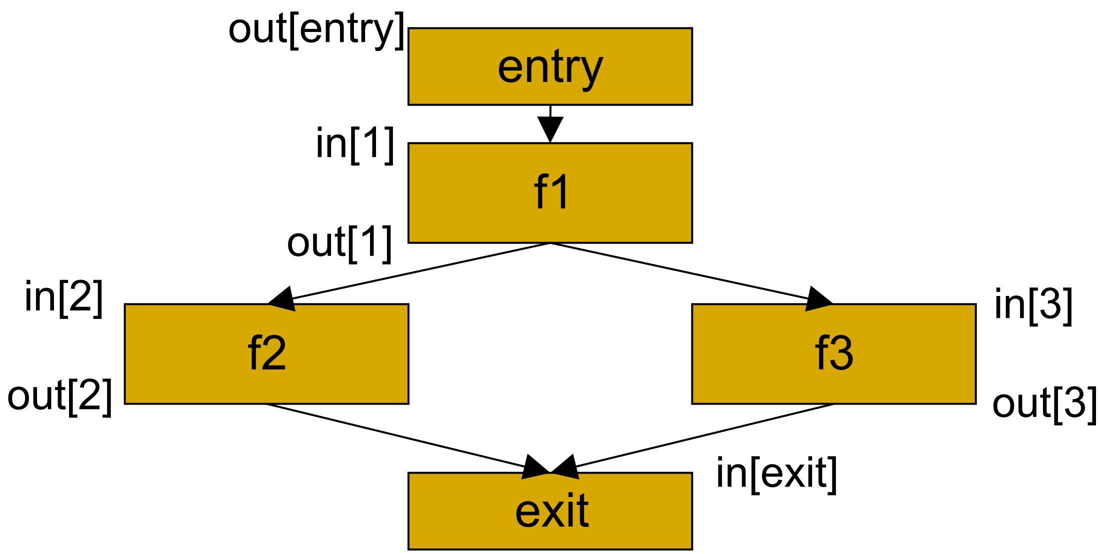
example

d3: a = 1
d4: b = 1

d5: c = a
d6: a = 4

- **A basic block b can**
  - Generate live variables: $Use[b]$
    - Set of locally exposed uses in b
  - Propagate incoming live variables: $OUT[b] - Def[b]$,
    - where def[b]= set of variables defined in basic block
- **Transfer function for block b:**
  - $in[b] = Use[b] \bigcup (out(b) - Def[b])$

Live Variable Analysis: Flow Graph



f	Use	Def
1	{ e }	{a , b}
2	{ }	{a, b}
3	{ a }	{a, c}

- $\mathrm{IN}[b] = f_b(OUT[b])$
- Join node: a node with multiple successors
- **Meet operator:**
  - $\mathrm{out}[b] = in[s_1] \cup in[s_2] \cup \ldots \cup in[s_n]$, where $s_1, \ldots, s_n$ are all successors of b

Live Variable Analysis: Iterative Algorithm

```
input: control flow graph CFG = (N, E, Entry, Exit)
// Boundary condition
    in[Exit] = ∅

// Initialization for iterative algorithm
    For each basic block B other than Exit
        in[B] = ∅

// Iterate
    While (Changes to any in[] occur) {
        For each basic block B other than Exit {
            out[B] = ⋃ (in[s]), for all successors s of B
            in[B] = fb(out[B])          // in[B]=Use[B] ⋃ (out[B]-Def[B])
        }
    }
```

Summary of Two Data-flow Problems

	Reaching Definitions	**Live Variables**
Domain	Sets of definitions	Sets of variables
Direction	forward: $\mathrm{out}[\mathrm{b}] = \mathrm{f}_{\mathrm{b}}(in[b])$ $in[b] = \wedge out[pred(b)]$	backward: $\mathrm{in}[\mathrm{b}] = \mathrm{f}_{\mathrm{b}}(out[b])$ $\mathrm{out}[\mathrm{b}] = \wedge in[succ(b)]$
Transfer function	$\mathrm{f}_{\mathrm{b}}(x) = \mathrm{Gen}_{\mathrm{b}} \cup (x - \mathrm{Kill}_{\mathrm{b}})$	$\mathrm{f}_{\mathrm{b}}(x) = \mathrm{Use}_{\mathrm{b}} \cup (x - \mathrm{Def}_{\mathrm{b}})$
Meet Operation (^)	$\cup$	$\cup$
Boundary Condition	$\mathrm{out}[\mathrm{entry}] = \varnothing$	$\mathrm{in}[\mathrm{exit}] = \varnothing$
Initial interior points	$\mathrm{out}[\mathrm{b}] = \varnothing$	$\mathrm{in}[\mathrm{b}] = \varnothing$

Predictable Success