Lecture Overview

Aside: Loop induction variables

Dataflow

- Using bit sets for LiveOut
- Reverse Postorder Traversal
- Dominance
- Other Dataflow Problems
- Interprocedural Dataflow

[Chapter 9]

Loop induction variables

An **induction variable** is a variable that takes on a value that is linearly related to the iteration number that a loop is executing.

We'll follow the convention of counting loop iterations from 0. Thus, the first iteration is iteration 0, the second is iteration 1, etc.

Suppose we have source code that looks like

```
i = 0
do {
        c = src[i]
        dst[i] = c
        i++
} while (c != 0);
```

and *src* and *dst* are both pointers to arrays of 2-byte characters. This code would translate as

```
        i = 0
loop:   t0 = 2 * i
        t1 = src + t0
        c = load t1
        t2 = 2 * i
        t3 = dst + t2
        store t3, c
        i = i + 1
        branch c != 0 loop
```

We'll make passes through this code to search for induction variables. On each pass, we examine each instruction to see if it computes what we know to be an induction variable.

In the first pass, t_0 is not flagged as an induction variable, as we don't know that i is an induction variable. Similarly, t_1, c, t_2, and t_3 are not flagged. The variable i, however, is seen to be an induction variable, as it has an assignment of the form i = i + k. Analysis of its starting value gives i a value of 1*iterationNum + 0 before the loop assignment to i, and 1*iterationNum + 1 after that assignment.

In the second pass, we see that $t_0 = 2 * i$, so it is an induction variable. Since i was 1*iterationNum+0 at this point in the loop, t_0 takes the value 2*iterationNum + 2*0.
Next we see that $t_1 = src + t_0$, so t_1 is an induction variable with value 2*iterationNum + src.
Likewise we find that t_2 and t_3 are induction variables, and we can summarize this as follows, using λ for the iteration number:

variable	formula
i	1λ + 0
t_0	2λ + 0
t_1	2λ + src
t_2	2λ + 0
t_3	2λ + dst

The third pass through yields no more induction variables, so we stop.

Now, for each induction variable that is kλ + m, we initialize that variable before the loop to

m – k , and then increment it by k where it was previously assigned.

```
        i = 0
        t0 = -2
        t1 = src - 2
        t2 = 0 - 2
        t3 = dst - 2
loop:   t0 = t0 + 2
        t1 = t1 + 2
        c = load t1
        t2 = t2 + 2
        t3 = t3 + 2
        store t3, c
        i = i + 1
        branch c != 0 loop
```

Next we can eliminate any variables whose values are not used except perhaps to compute themselves. The three variables we find are i, t_0 and t_2.

```
        t1 = src - 2
        t3 = dst - 2
loop:   t1 = t1 + 2
        c = load t1
        t3 = t3 + 2
        store t3, c
        branch c != 0 loop
```

That's pretty fast code compared with the original. There are now no multiplies in the loop (which is a good thing, because multiplies in general take longer than additions).

If src and dst are not used outside the loop, the compiler can discover that the temporaries are not needed and reduce this code further to:

```
        src = src – 2
        dst = dst – 2
loop:   src = src + 2
        c = load src
        dst = dst + 2
        store dst, c
        branch c != 0 loop
```

Bit Sets

Bit sets are a fast way to encode and operate on relatively small sets. By "relatively small," we mean that the universe for the sets is below 32, 64, or 128 elements. We'll use 32 elements for illustration purposes.

Each element of the universe is assigned a bit from a 32-bit word. Then an integer word represents a set that consists of exactly those elements whose bit is 1 in the word.

Suppose our universe is {a, b, c, d, e}. We could assign a to bit 0, b to bit 1, c to bit 2, d to bit 3, and e to bit 4. The higher bits are unused.

b_4	b_3	b_2	b_1	b_0
e	d	c	b	a

Then, for instance, the integer 9 would represent the set {a, d} because $9_{10} = 01001_2$, so there are 1's in the bits for a and d.

Then, set-theoretic operations become bitwise operations on the integers representing the sets.

Union $\rightarrow$ bitwise 'or'

$$\{d, a\} \cup \{c, b\} = \{d, c, b, a\}$$

$$01001 \mid 00110 = 01111$$

Intersection $\rightarrow$ bitwise 'and'

$$\{d, c, a\} \cap \{c, b\} = \{c\}$$

$$01101 \,\&\, 00110 = 00100$$

Set Subtraction $\rightarrow$ bitwise 'and-not'

$$\{d, c, a\} \setminus \{c, b\} = \{d, a\}$$

$$01101 \,\&\, \sim 00110 = 01001$$

Symmetric Difference → bitwise 'xor'

$$\{d, c, a\} \triangle \{c, b\} = \{d, b, a\}$$

$$01101 \text{ ^ } 00110 = 01011$$

Complement → bitwise complement

$$\overline{\{d, c, a\}} = \{e, b\}$$

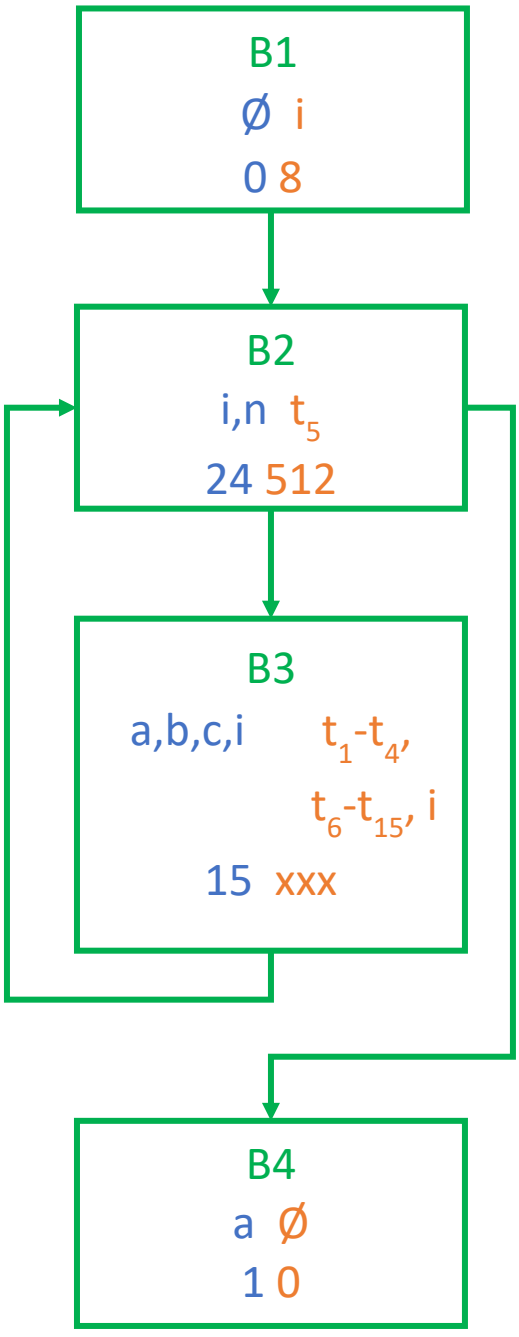
$$\sim 01101 = 10010$$

Our example program that we ran LiveOut analysis on last lecture had variables {a, b, c, i, n, t_1, t_2, …, t_{16}}. Let's use this as a universe for a bitset.

b_{21} t_{16}	…	b_7 t_3	b_6 t_2	b_5 t_1	b_4 n	b_3 i	b_2 c	b_1 b	b_0 a

$$\text{LiveOut}(n) = \bigcup_{m \in \text{succ}(n)} (\text{UEVar}(m) \cup (\text{LiveOut}(m) \cap \overline{\text{VarKill}(m)}))$$

UEVar(m) | (LiveOut(m) &~ VarKill(m))



LiveOut

Ø = 0

i,n = 24 | (0 &~512) = 24

Ø = 0

a,b,c,i = 15 | (0 &~ xxx)
| 1 | (0 &~ 0) = 15

Ø = 0

a,b,c,i,n = 24 | (15 &~ 512)
= 31

Ø

Ø

Reverse Postorder Traversals

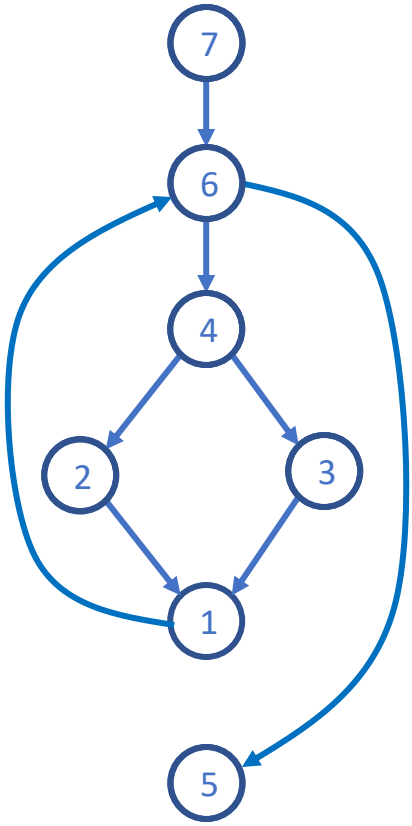
A postorder traversal of a graph or digraph can be done with the following algorithm. All vertices start unmarked.

```
POSTORDER(G, v)
mark v
for each (out-)neighbor n of v
    if n is unmarked
        POSTORDER(G, n)
visit v
```

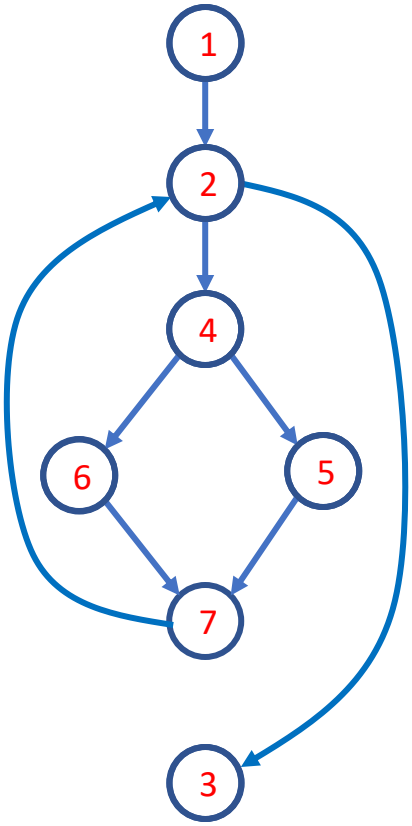
Suppose that "visit v" denotes the operation

```
v.num = count++
```

where count is a global variable that starts at 1. Then the number assigned to v is called its postorder number. If that number is subtracted from n+1, it is called the reverse postorder number.



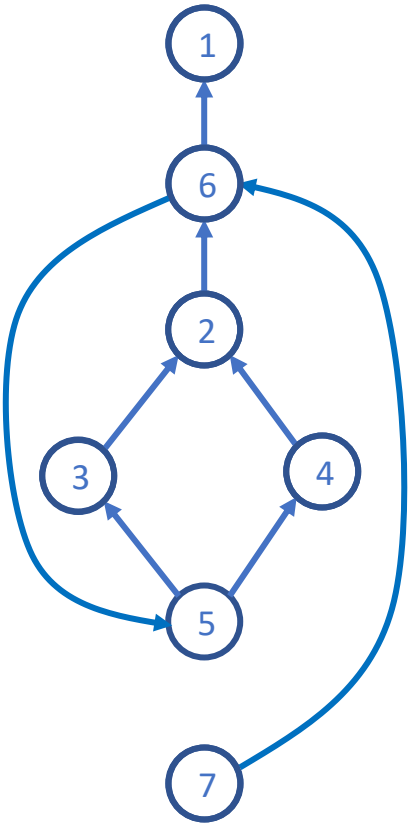
Postorder



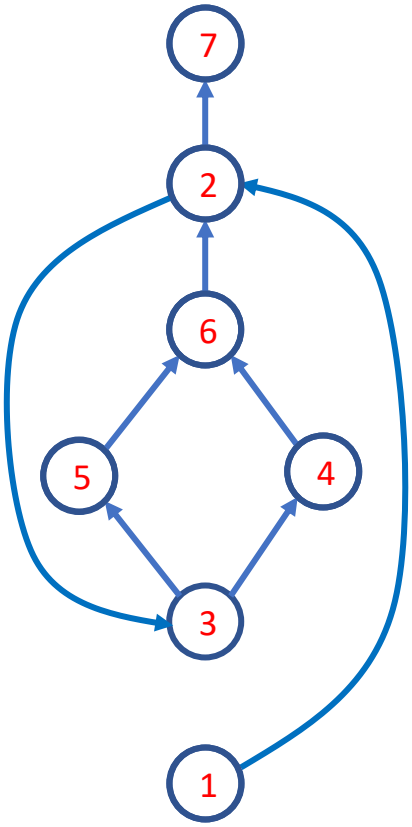
Reverse Postorder

A forward dataflow problem (where the data flows from the predecessors of a node to that node) should visit the nodes in Reverse Postorder (RPO). This allows as many nodes as possible to use the just-computed dataflow values rather than the dataflow values from the last round.

A backward dataflow problem (where the data flows from the successors of a node to that node) should visit the nodes in Reverse Postorder (RPO) on the transpose (reverse) of the graph.



Postorder

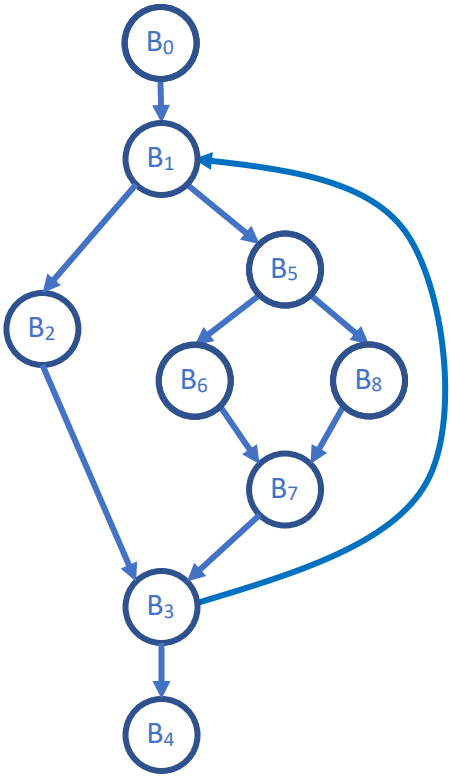


Reverse Postorder

Dominance

A key tool that compilers use to reason about the shape and structure of the CFG is the notion of dominators. A node B_i in the CFG is a dominator of node B_j (written $B_i \gg B_j$) if B_i lies on every path from the entry node to B_j. By definition, each B_i dominates itself.

A simple dataflow problem can annotate each node B_j in the CFG with a set DOM(B_j) that contains all nodes which dominate B_j.



Node	Dom(.)
B_0	{0}
B_1	{0, 1}
B_2	{0, 1, 2}
B_3	{0, 1, 3}
B_4	{0, 1, 3, 4}
B_5	{0, 1, 5}
B_6	{0, 1, 5, 6}
B_7	{0, 1, 5, 7}
B_8	{0, 1, 5, 8}

The dataflow equation for computing dominators is:

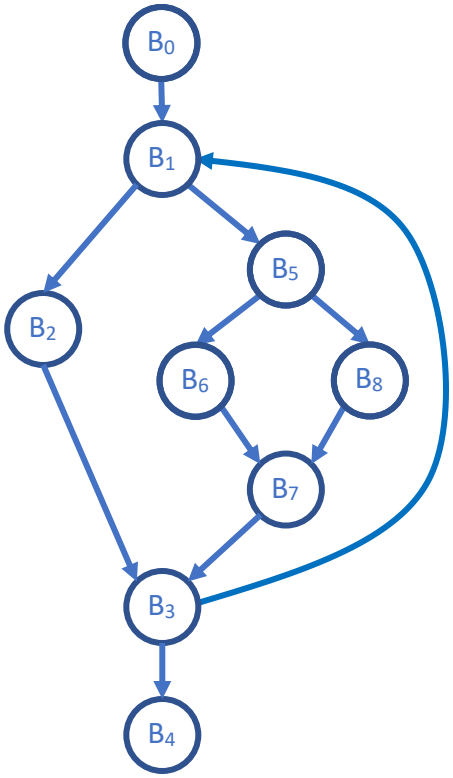
$$\mathrm{Dom}(n) = \{n\} \cup \bigcap_{m \in \mathrm{pred}(n)} \mathrm{Dom}(m)$$

With the initial conditions that Dom(n_0) = {n_0}, and otherwise Dom(n) = N, the set of all nodes in the CFG.

Because the intersection is taken over all predecessors, this is a forward dataflow problem.

```
// CFG has block set N, blocks numbered 0 to |N|-1
Dom(0) = {0}
for i = 1 to |N|-1
      Dom(i) = N

changed = true;
while (changed)
   changed = false
   for i = 1 to |N|-1
        recompute Dom(i) according to formula
        if Dom(i) changed
                changed = true
```



$$\mathrm{Dom}(n) = \{n\} \cup \bigcap_{m \in \mathrm{pred}(n)} \mathrm{Dom}(m)$$

Dom(n) computation:

	B_0	B_1	B_2	B_3	B_4	B_5	B_6	B_7	B_8
	0	N	N	N	N	N	N	N	N
1	0	01	012	0123	01234	015	0156	01567	0158
2	0	01	012	013	0134	015	0156	0157	0158
3	0	01	012	013	0134	015	0156	0157	0158

With an RPO traversal, we could stop after 2 iterations.

Other Dataflow Problems

Dataflow analyses are used to prove the safety of applying code transformations in particular situations. Thus, many dataflow problems have been proposed, each to drive a particular optimization.

Available Expressions

In a compiler using 3AC, an expression is the right-hand side of an instruction. To identify redundant expressions, we can compute information about the *availability* of expressions.

An expression e is available at point p in a procedure iff on every path from the procedure's entry to p, e is evaluated and none of its constituent subexpressions has been redefined inbetween that evaluation and p.

This analysis computes AvailIn(n) for each node n in a CFG, which contains all expressions in the procedure which are available on entry to the block n.

The initial conditions are AvailIn(n_0) = Ø, and otherwise AvailIn(n) = {all expressions}.

The dataflow equations are:

$$\text{AvailIn}(n) = \bigcap_{m \in \text{pred}(n)} (\text{DEExpr}(m) \cup (\text{AvailIn}(m) \cap \overline{\text{ExprKill}(m)}))$$

Where DEExpr(m) is the set of downward exposed expressions in m. An expression e is in DEExpr(m) means block m evaluates e and none of e's operands is defined between the last evaluation of e and the end of m.
ExprKill(m) contains all those expressions "killed" by a definition in m. An expression is killed if one or more of its operands is redefined in the block.

For a block m, one can compute DEExpr(m) and ExprKill(m) in a single bottom-up pass through the instructions of the block. One must do this for every block before starting the dataflow solver.

```
for each block b
    init(b)

init(b)
      DEExpr(b) = Ø
      ExprKill(b) = Ø
      for i = b.numInstr downto 1   // instruction i is
                                    // x[i] = y[i] op z[i]
          if y[i] op z[i] ∉ ExprKill(b)
              add y[i] op z[i] to DEExpr(b)
          for every expression e containing x[i]
              add e to ExprKill(b)
```

To do this computation, one needs a data structure holding all expressions in the procedure, with easy lookup to find expressions containing a given variable. A (hash?)table of expressions indexed by variable will suffice. Each entry of the table would be a list of expressions.

Reaching Definitions

Oftentimes a compiler will need to know where an operand is defined. If multiple paths in the CFG lead to an operation, then multiple definitions may provide the value of the operand.

We say a definition d of v reaches operation i iff i reads the value of v and there exists a path from d to i that does not define v.
For CFG node n, we let Reaches(n) be defined as all definitions d of v for which there is a path to the start of n that does not define v. Note there is no requirement for n to read the value of v. The dataflow equations for Reaches are:

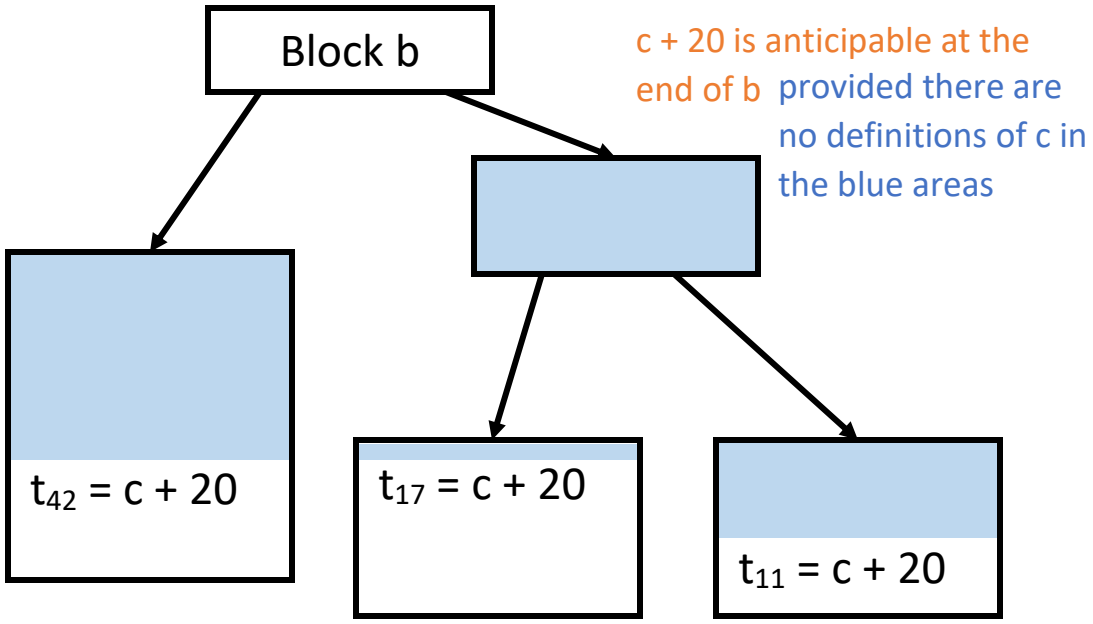
$$\text{Reaches}(n) = \bigcup_{m \in pred(n)} (\text{DEDef}(m) \cup \left(\text{Reaches}(m) \cap \overline{\text{DefKill}(m)}\right))$$

where we start with Reaches(n) = Ø for all n.

DEDef(m) is the set of downward-exposed definitions in m—those for which the defined name is not subsequently redefined in m. DefKill(m) contains all the definition points that are obscured by a definition of the same name in m. In other words, a definition point d is in DefKill(m) if d defines some name and m contains a definition of the same name.
DEDef and DefKill are defined over the set of definition points (instructions) and require a mapping from names to definition points. Again, a (hash?) table indexed by names suffices. Here the table would have lists of definition points as entries.

Anticipable Expressions

An expression e is considered anticipable (or very busy) on exit from block b iff (1) every path that leaves b evaluates and subsequently uses e, and (2) evaluating e at the end of b would produce the same result as the first evaluation of e along each of those paths.



The dataflow equations are:

$$\text{AntOut}(n) = \bigcap_{m \in \text{succ}(n)} (\text{UEExpr}(m) \cup (\text{AntOut}(m) \cap \overline{\text{ExprKill}(m)}))$$

where the initial conditions are AntOut(n_f) = Ø and otherwise AntOut(n) = {all expressions}.

The results of this analysis are used in code motion to decrease execution time and shrink the size of the compiled code.

Interprocedural Summary Problems

When taking procedure calls into account in dataflow analysis, to be conservative, we assume that procedures modify every variable they can access—they count as definitions for these variables. This includes parameters, module-level and global/static variables.

We can reduce these assumptions by summarizing which variables a procedure (and its subprocedures) actually modify. The result is the interprocedural may modify problem. It is one of the simplest interprocedural analyses. It is posed as a set of dataflow equations over the program's call graph (not the CFG!).

$$\text{MayMod}(p) =$$

$$\text{LocalMod(p)} \cup \bigcup_{e=(p,q)} unbind_{\text{e}}(\text{MayMod(q)})$$

Where each MayMod(p) is initialized to LocalMod(p). $unbind_{\text{e}}$ maps names from q's name space to p's name space.

Given the MayMod sets for each procedure, a compiler can compute the set of names that might be modified at any procedure call in procedure p to procedure q by computing S = $unbind_e$(MayMod(q)) and then adding to S any names that are aliased inside p to names in S.

There's a similar procedure summary MayRef(p) that is the set of variables that a procedure and its subprocedures may reference. It has a similar dataflow formulation.