Static Program Analysis

CS4430/7430

Introduction to Compilers

Static Program Analysis

- Analyzes the source code of the program and reasons about the run-time program behavior
- Many uses
  - Traditionally in compilers in order to perform optimizing, semantics-preserving transformations
  - Recently in software tools for testing and validation: our focus

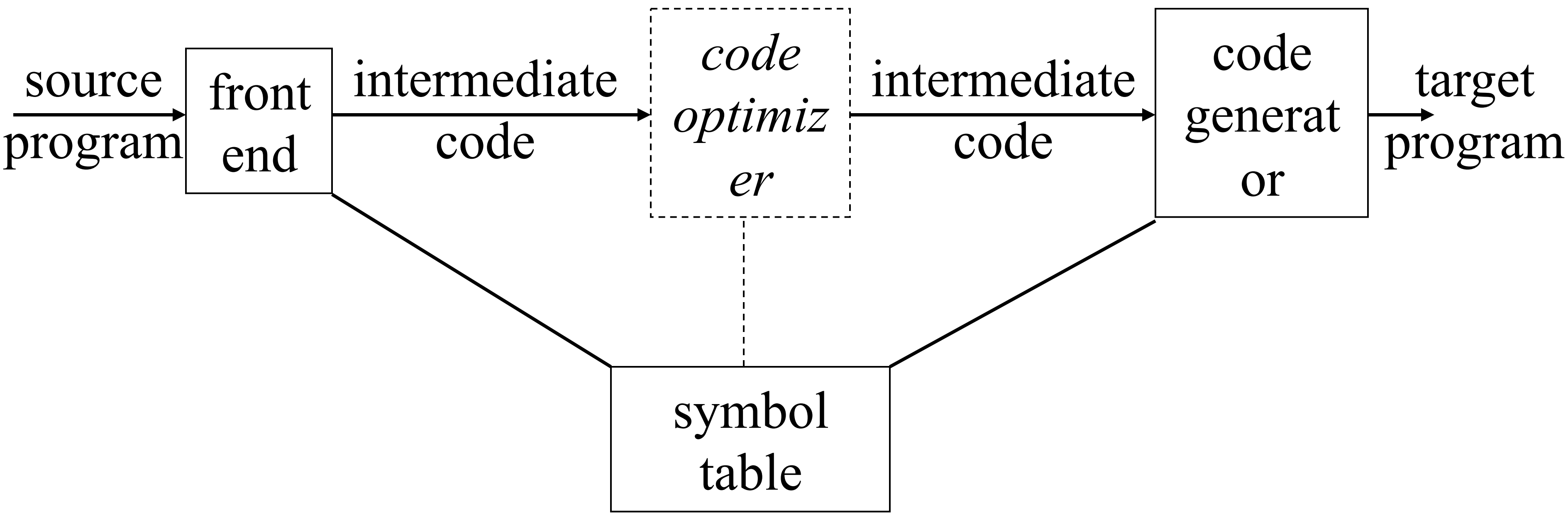
Static vs. Dynamic program properties

- **Static** properties
  - any property that may be determined through analysis of <u>program text</u>
    - e.g., for some languages, the type of a program may be determined entirely through analysis of program source
      - e.g., ML, Java, & Pascal have "static type inference"
- **Dynamic** properties
  - any property that may only be discovered through execution of the program
    - e.g., "the final result of program p is 42" – can't be discovered in general without some form of execution
- Compilation involves many forms of "static analysis"
  - e.g., type checking, the definition and use of variables, information of data and control flow, …

Outline

- Basic Blocks
- Control flow graphs
- Local optimizations --- within basic blocks
- Global optimizations --- across basic blocks

Compilation



An optimization is a semantics-preserving transformation

Definition: A Basic Block (BB) is

1. Sequence of consecutive instructions, starting with a unique entry point
   – aka, the "header" or "leader" and
2. ending with exit instruction(s) that transfer to another BB or ends the program (e.g. a Halt instruction)

Defining Basic Blocks (cont'd)

- Possible to have single-instruction BBs.
- Leaders are *explicitly* created by being the destination of branch- and call destinations.
- Leaders are *implicitly* created by the
  - previous instruction branching away (via jump or call), or
  - by fall-though e.g. in the case of conditional branches

Sample: Basic Block

A basic block is a sequence of 1 or more consecutive operations whose first is the sole entry and whose last is the sole exit point.

- Only the first statement can be a label or target of a jump. But being the first operation of a BB via fall-though is also possible
- Only the last statement can be a jump statement. But non-control-flow operations can also be exits points, for example, when the next one happens to be target of a branch.

(1)-(4) form Basic Block

```
(0)  L1:
(1)  i := m-1
(2)  j := n
(3) t1 := 4*n
(4)  v := a[t1]
(5)  L2: ...
```

(0) is Basic Block

```
(0)  L3: goto foo
```

Multiple Basic Blocks

```
(0)  i := m-1
(1)  j := n
(2)  L4:
(3) t2 := 4 * i
(4)  goto bar
(5) t3 := t3 -j
```

Identifying Basic Blocks

1.) Identify "leaders", i.e. the first statements of basic blocks. Leaders are:

- The first statement of the program; e.g. first instruction of main() function
- The target of a call, conditional, or unconditional branch
- Operation following a control-transfer instruction; this operation is an implicit target by fall-through; note that successor of unconditional branch is candidate for unreachable code

2.) For each leader: its basic block consists of the leader itself plus all 0 or more operations up to and excluding the next leader or up to the halt instruction

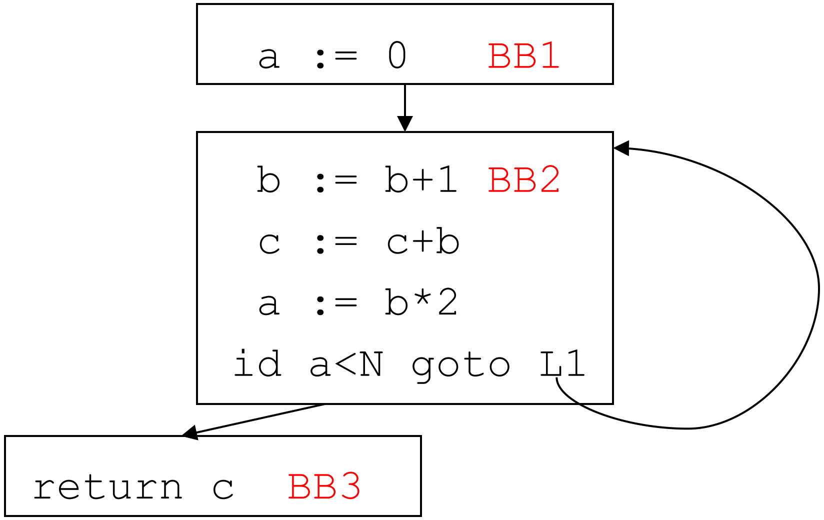
Example:	***Leaders:***	***Basic Blocks:***
`L0:(1) a := 0`	`(1)`	`(1)`
`L1:(2) b := b+1`	`(2)`	`(2) (3) (4) (5)`
`   (3) c := c+b`		
`   (4) a := b*2`		
`   (5) if a<N goto L1`		
`   (6) return c`	`(6)`	`(6)`

Control Flow Graph (CFG)

A program's Control Flow Graph is a directed graph, whose nodes are Basic Blocks, and whose vertices are program-defined flows of control from Basic Blocks to others

Example

```
(1)  a  :=  0
L1:
(2)  b  :=  b+1
(3)  c  :=  c+b
(4)  a  :=  b*2
(5)  if a<N goto L1
(6)  return c
```



Sample: Quicksort

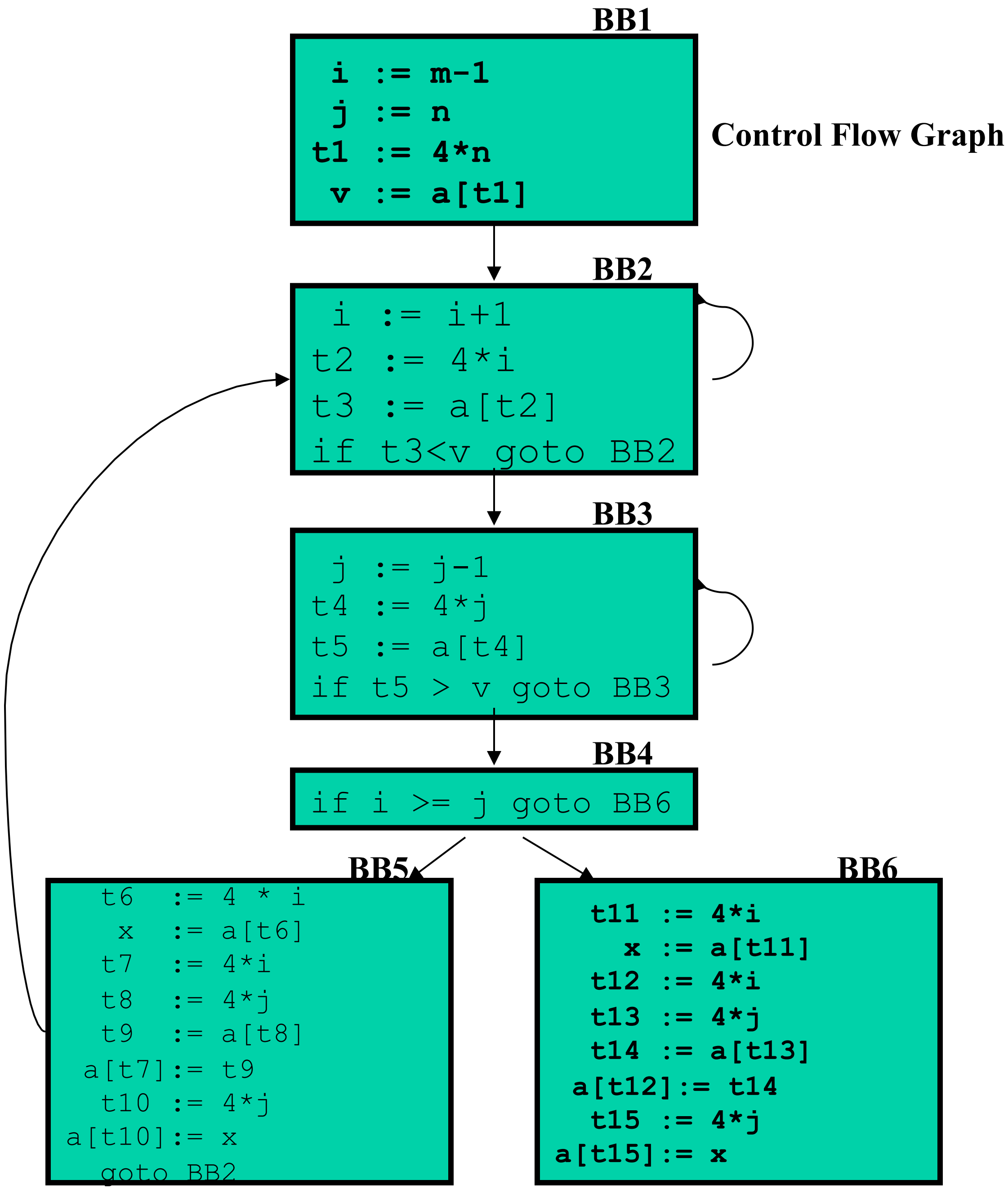
```
// assume an external input-output array: int a[]
void quicksort( int m, int n ) {
   int i, j, v, x; // temps
   if ( n <= m ) return;
   i = m-1;
   j = n;
   v = a[n];
   while(1) {
       do i=i+1; while( a[i] < v );
       do j=j-1; while( a[j] > v );
       if ( i >= j ) break;
       x = a[i]; a[i] = a[j]; a[j] = x;
   } //end while
   x = a[i]; a[i] =a [n]; a[n] = x;
   quicksort( m, j );
   quicksort( i+1, n );
} //end quicksort
```

Quicksort IR Code

```
  (1) i := m-1
  (2) j := n
  (3) t1 := 4*n
  (4) v := a[t1]
L0: L1:
  (5) i := i+1
  (6) t2 := 4*i
  (7) t3 := a[t2]
  (8) if t3<v goto L1
L2:
  (9) j := j-1
  (10) t4 := 4*j
  (11) t5 := a[t4]
  (12) if t5>v goto L2
  (13) if i>=j goto L3
  (14) t6 := 4*i
  (15) x := a[t6]
  (16) t7 := 4*i
  (17) t8 := 4*j
  (18) t9 := a[t8]
  (19) a[t7] := t9
  (20) t10 := 4*j
  (21) a[t10] := x
  (22) goto L0
L3:
  (23) t11 := 4*i
  (24) x := a[t11]
  (25) t12 := 4*i
  (26) t13 := 4*j
  (27) t14 := a[t13]
  (28) a[t12] := t14
  (29) t15 := 4*j
  (30) a[t15] := x
  (31) 2 calls ...
```

Quicksort CFG

```
BB1:  (1)--(4)
BB2:  (5)--(8)
BB3:  (9)--(12)
BB4:  (13)
BB5:  (14)--(22)
BB6:  (23)--(30)
```



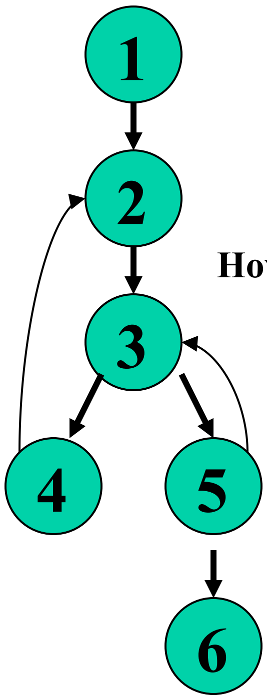
What is a "Loop"?

- Since CFG is a graph, it may contain loops,
  - AKA strongly-connected-components (SCC)
  - Generally, a loop is a directed graph, whose nodes can reach all other nodes along some path
  - This includes “unstructured” loops, with multiple entry and multiple exit points
- A structured loop (proper loop) has just 1 entry point, and (generally) a single point of exit
- Remark: Loops created by mapping high-level source programs to IR or assembly code are proper, unless disturbed by Goto (and Break) statements
- Goto can create any loop; break creates additional exits

Loops, Cont’d

Unstructured

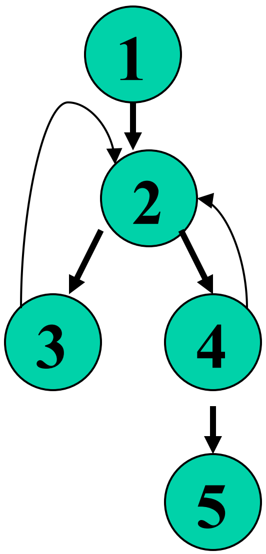
Loop: 2, 3, 4, 5



How many loops?

2 proper loops, one unstructured loop

Loop1: 2, 3; Loop2: 2, 4; Loop3: 2, 3, 4



We will return to this loop concept when we discuss loop optimization later in the semester

Another Example

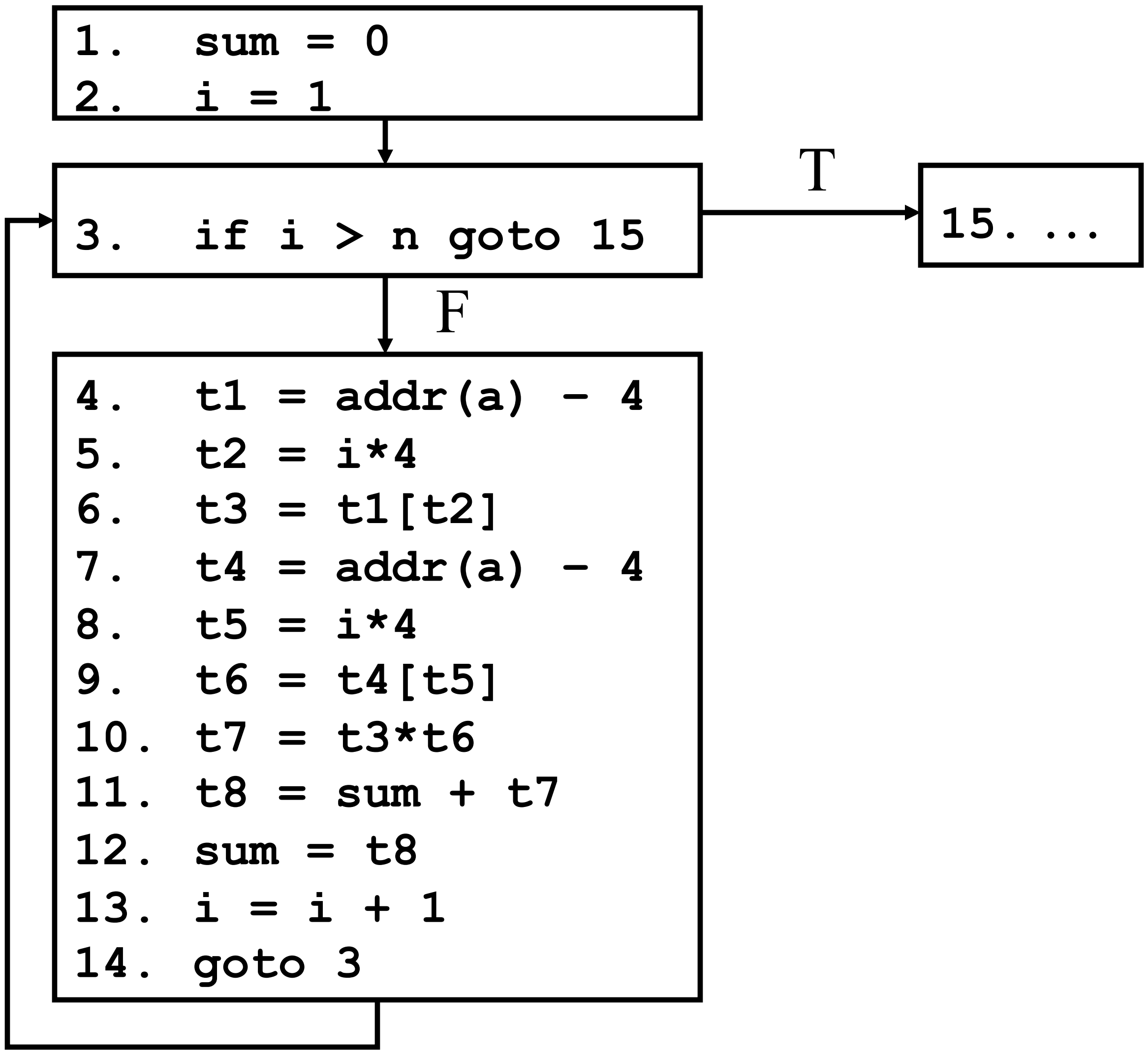
- Define classical optimizations using an example Fortran loop
- Opportunities result from table-driven code generation

```
…
sum = 0
do 10 i = 1, n
10  sum = sum + a[i]*a[i]
…
```

Three Address Code

1.	sum = 0	initialize sum
2.	i = 1	initialize loop counter
3.	if i > n goto 15	loop test, check for limit
4.	t1 = addr(a) – 4	
5.	t2 = i * 4	a[i]
6.	t3 = t1[t2]	
7.	t4 = addr(a) – 4	
8.	t5 = i * 4	a[i]
9.	t6 = t4[t5]	
10.	t7 = t3 * t6	a[i]*a[i]
11.	t8 = sum + t7	
12.	sum = t8	increment sum
13.	i = i + 1	increment loop counter
14.	goto 3	
15.	…	

Control Flow Graph (CFG)



Building Control Flow Graph

How to Partition into basic blocks

1. Determine the *leader* statements
   - (i) First program statement
   - (ii) Targets of conditional or unconditional goto's
   - (iii) Any statement following a goto
2. For each leader, its basic block consists of the leader and all statements up to but not including the next leader or the end of the program

Building Control Flow Graph

- Add flow-of-control information
- There is a directed edge from basic block B_1 to block B_2 if B_2 can immediately follow B_1 in some execution sequence

(i) B_2 immediately follows B_1 and B_1 does not end in an unconditional jump

(ii) There is a jump from the last statement in B_1 to the first statement in B_2

Leader Statements and Basic Blocks

```
1.  sum = 0
2.  i = 1
3.  if i > n goto 15
4.  t1 = addr(a) - 4
5.  t2 = i*4
6.  t3 = t1[t2]
7.  t4 = addr(a) - 4
8.  t5 = i*4
9.  t6 = t5[t5]
10. t7 = t3*t6
11. t8 = sum + t7
12. sum = t8
13. i = i + 1
14. goto 3
15. …
```

Analysis and optimizing transformations

- *Local optimizations* – performed by local analysis of a basic block
- *Global optimizations* – requires analysis of statements outside a basic block
- Local optimizations are performed first, followed by global optimizations

Local optimizations --- optimizing transformations of a basic blocks

- Local common subexpression elimination
- Dead code elimination
- Copy propagation
- Constant propagation
- Renaming of compiler-generated temporaries to share storage

Example 1: Local Common Subexpression Elimination

```
1. t1 = 4 * i
2. t2 = a [ t1 ]
3. t3 = 4 * i
4. t4 = b [ t3 ]
5. t5 = t2 * t4
6. t6 = prod * t5
7. prod = t6
8. t7 = i + 1
9. i = t7
10.if i <= 20 goto 1
```

Example 2: Local Dead Code Elimination

```
1. a = y + 2
2. z = x + w
3. x = y + 2
4. z = b + c
5. b = y + 2
```

```
1'. a = y + 2
2'. x = a
3'. z = b + c
4'. b = a
```

Example 3: Local Constant Propagation

```
1.   t1 = 1
2.   a = t1
3.   t2 = 1 + a
4.   k = t2
5.   t3 = cvttoreal(k)
6.   t4 = 6.2 + t3
7.   t3 = t4
```

Assuming a, k, t3, and t4 are used beyond:

```
1'. a = 1
2'. k = 2
3'. t4 = 8.2
4'. t3 = 8.2
```

D. Gries' algorithm:

- Process 3-address statements in order
- Check if operand is constant; if so, substitute
- If all operands are constant,
  - Do operation, and add value to table associated with LHS
- If not all operands constant
  - Delete any table entry for LHS

Problems

- Troubles with arrays and pointers. Consider:

```
x = a[k];
a[j] = y;
z = a[k];
```

Can we transform this code into the following?

```
x = a[k];
a[j] = y;
z = x;
```

Next Time

- Continue with Static Analysis Overview
- HW1 due tomorrow at 11:59pm

Global optimizations --- require analysis outside of basic blocks

- Global common subexpression elimination
- Dead code elimination
- Constant propagation
- Loop optimizations
  - Loop invariant code motion
  - Strength reduction
  - Induction variable elimination

Global optimizations --- depend on data-flow analysis

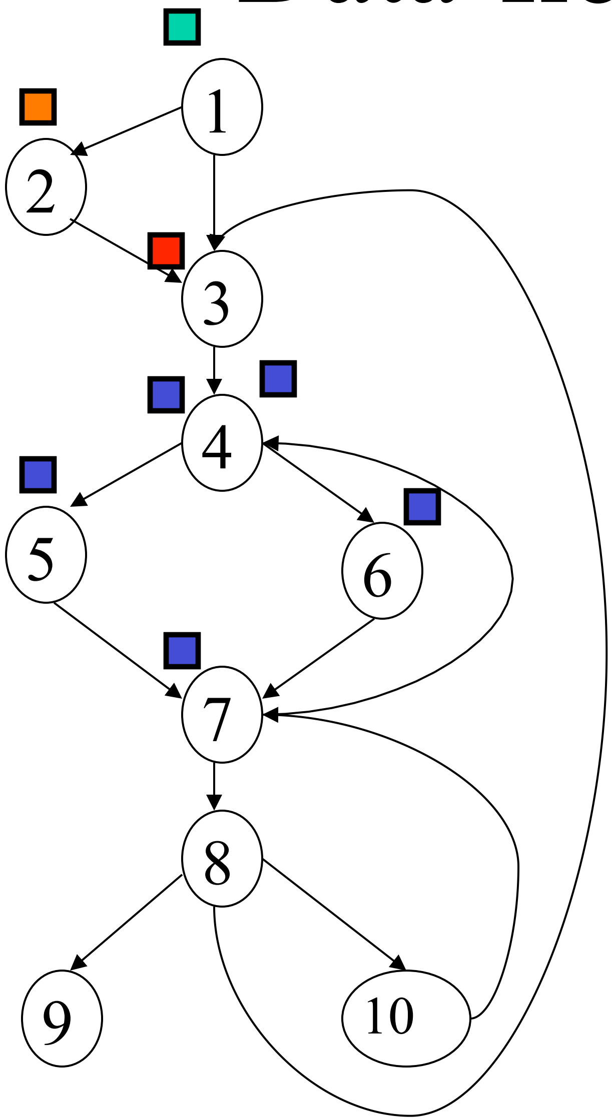
- Data-flow analysis refers to a body of techniques that derive information about the flow of data along program execution paths

- For example, in order to perform global subexpression elimination we need to determine that 2 textually identical expressions evaluate to the same result along any possible execution path

Introduction to Data-flow Analysis

- Collects information about the flow of data along execution paths
  - E.g., at some point we needed to know where a variable was last defined
- *Data-flow information*
- *Data-flow analysis*

Data-flow Analysis



- $G = (N, E, \rho)$
- Data-flow equations (also referred as transfer functions):

$$\text{out}(i) = \text{gen}(i) \cup (\text{in}(i) - \text{kill}(i))$$

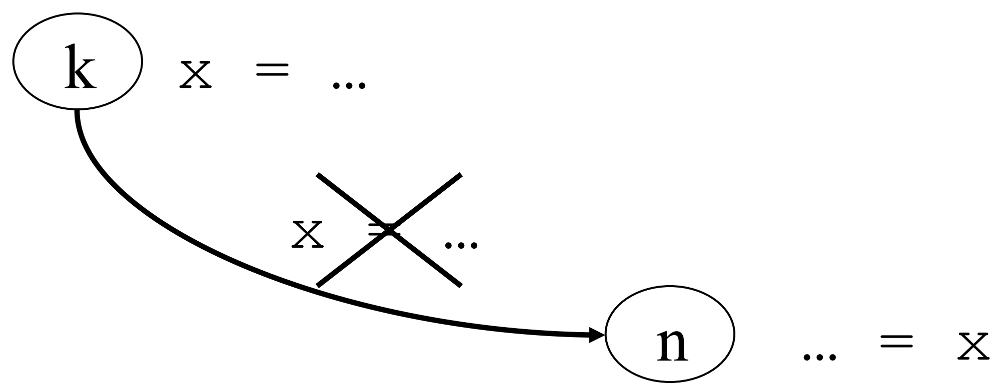
- Equations can be defined over basic blocks or over single statements. We will use equations over single statements

Four Classical Data-flow Problems

- Reaching definitions (*Reach*)
- Live uses of variables (*Live*)
- Available expressions (*Avail*)
- Very Busy Expressions (*VeryB*)
- Def-use chains built from *Reach*, and the dual Use-def chains, built from *Live*, play role in many optimizations
- *Avail* enables global common subexpression elimination
- *VeryB* is used for conservative code motion

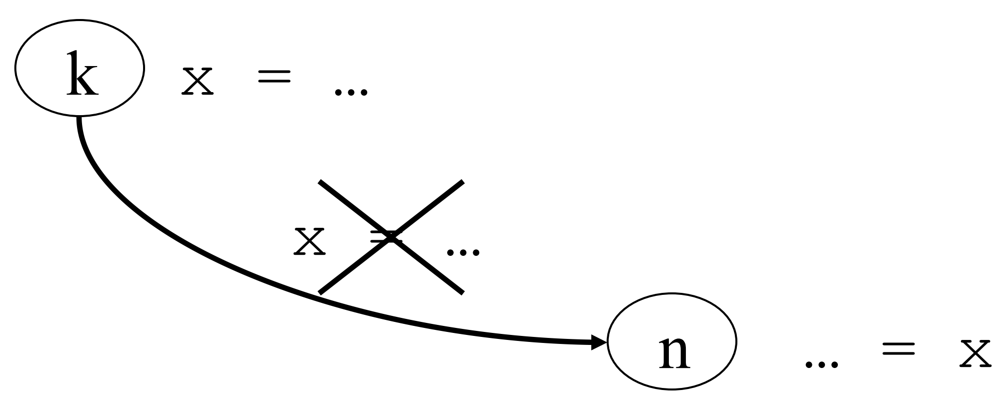
Reaching Definitions

- ***Definition*** A statement that may change the value of a variable (e.g., x = i+5)
- A definition of a variable *x* at node *k* *reaches* node *n* if there is a path clear of a definition of *x* from *k* to *n*.



Live Uses of Variables

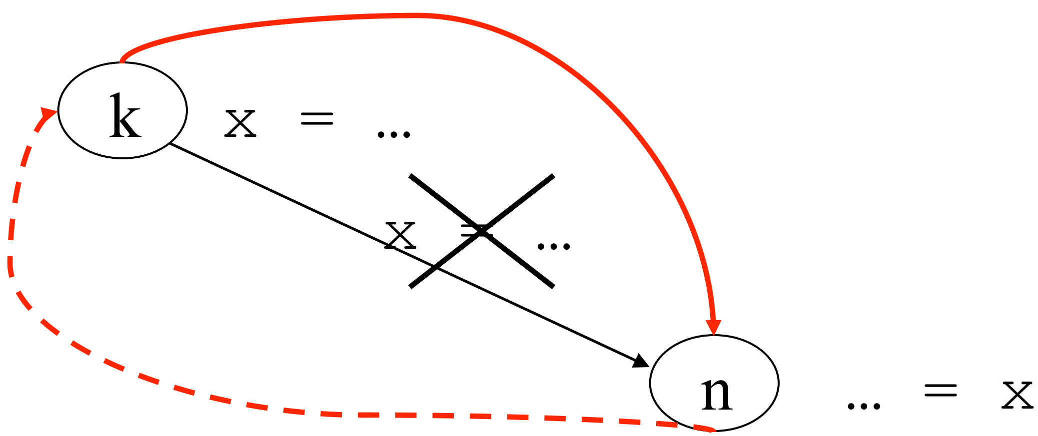
- ***Use*** Appearance of a variable as an operand of a 3-address statement (e.g., y=x+4)
- A use of a variable *x* at node *n* is *live on exit* from *k* if there is a path from *k* to *n* clear of definition of *x*.



Def-use Relations

- *Use-def chain* links an use to a definition that reaches that use

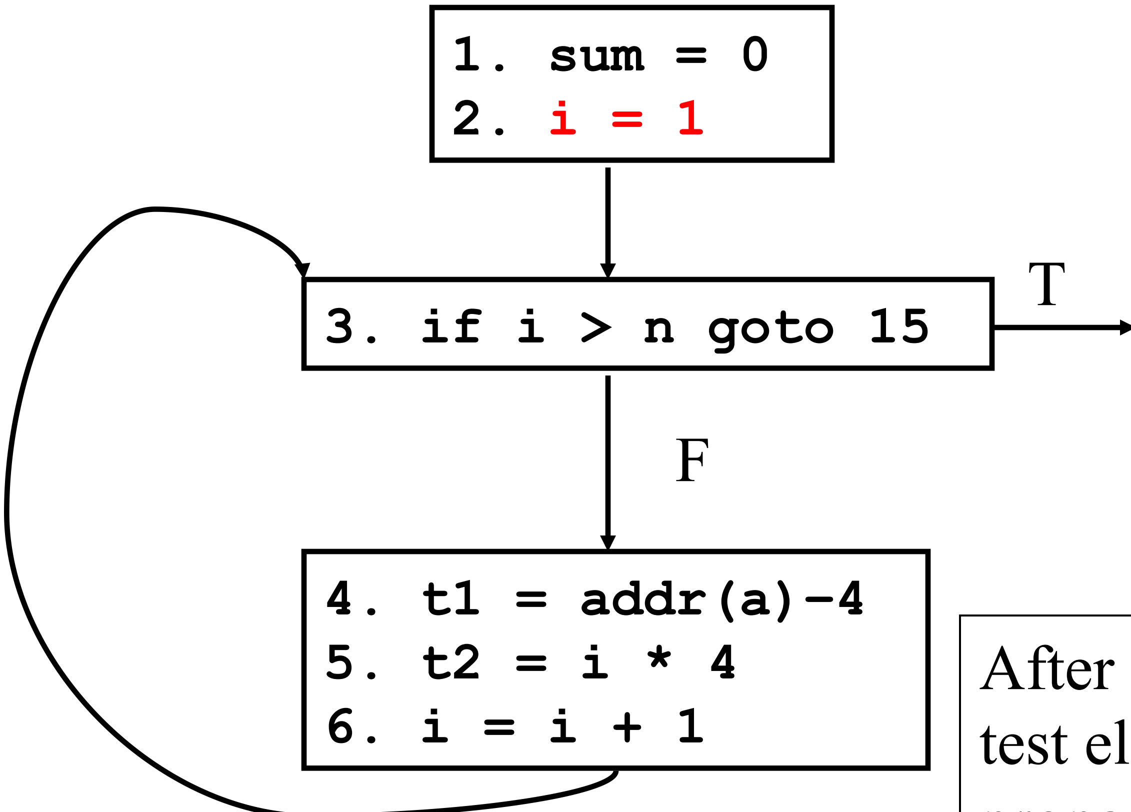
- *Def-use chain* links a definition to an use that it reaches



Optimizations Enabled

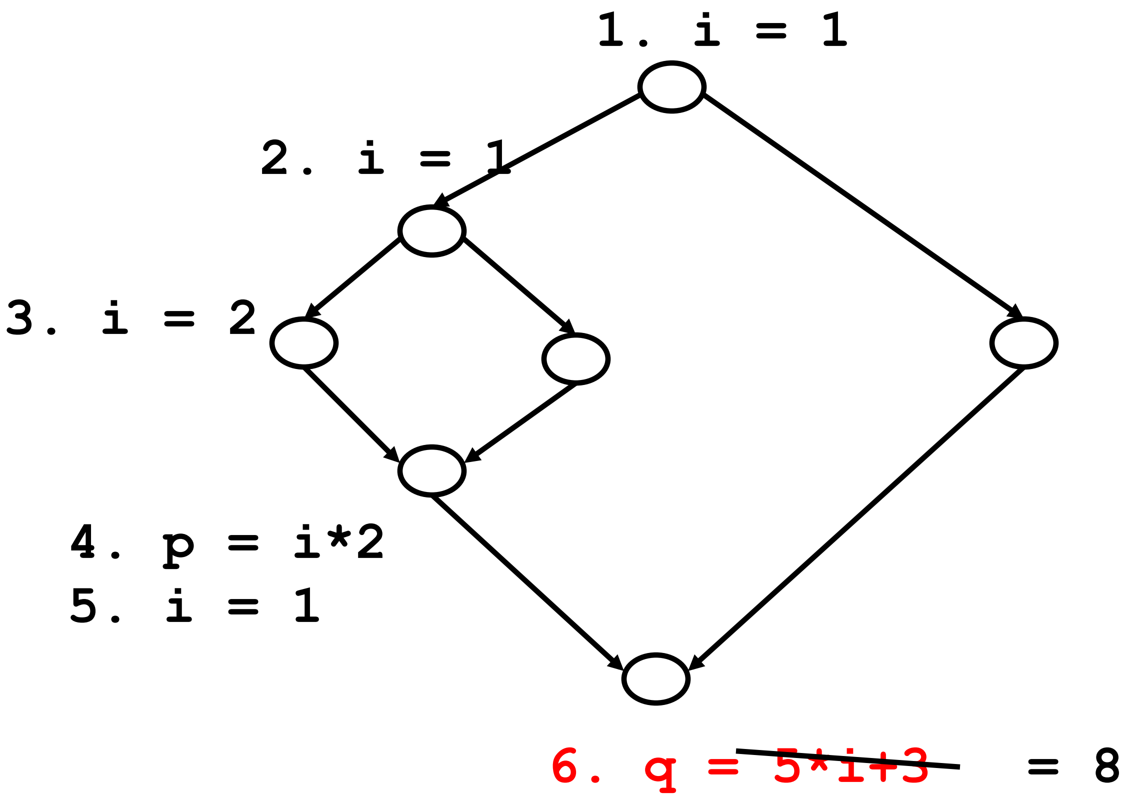
- Dead code elimination (Def-use)
- Code motion (Use-def)
- Constant propagation (Use-def)
- Strength reduction (Use-def)
- Test elision (Use-def)
- Copy propagation (Def-use)

Dead Code Elimination



After strength reduction, test elision and constant propagation, the def-use links from i=1 disappear. It becomes dead code.

Constant Propagation



Terms

- Control flow graph (CFG)
- Basic block
- Local optimization
- Global optimization
- Data-flow analysis

Common Subexpression Elimination

```
1.   sum = 0
2.   i = 1
3.   if i > n goto 15
4.   t1 = addr(a) - 4
5.   t2 = i*4
6.   t3 = t1[t2]
7.   t4 = addr(a) - 4
8.   t5 = i*4
9.   t6 = t4[t5]
10.  t7 = t3*t6
11.  t8 = sum + t7
12.  sum = t8
13.  i = i + 1
14.  goto 3
15.  …
```

```
1.   sum = 0
2.   i = 1
3.   if i > n goto 15
4.   t1 = addr(a) - 4
5.   t2 = i*4
6.   t3 = t1[t2]
7.   t4 = addr(a) - 4
8.   t5 = i*4
9.   t6 = t4[t5]
10.  t7 = t3*t6
10a  t7 = t3*t3
11.  t8 = sum + t7
11a  sum = sum + t7
12.  sum = t8
13.  i = i + 1
14.  goto 3
```

Invariant Code Motion

```
1.   sum = 0
2.   i = 1
3.   if i > n goto 15
4.   t1 = addr(a) - 4
5.   t2 = i * 4
6.   t3 = t1[t2]
10a  t7 = t3 * t3
11a  sum = sum + t7
13.  i = i + 1
14.  goto 3
15.  …
```

```
1.   sum = 0
2.   i = 1
2a   t1 = addr(a) - 4
3.   if i > n goto 15
4.   t1 = addr(a) - 4
5.   t2 = i * 4
6.   t3 = t1[t2]
10a  t7 = t3 * t3
11a  sum = sum + t7
13.  i = i + 1
14.  goto 3
15.  …
```

Strength Reduction

```
1.   sum = 0
2.   i = 1
2a   t1 = addr(a) - 4
3.   if i > n goto 15
5.   t2 = i * 4
6.   t3 = t1[t2]
10a  t7 = t3 * t3
11a  sum = sum + t7
13.  i = i + 1
14.  goto 3
15.  …
```

```
1.   sum = 0
2.   i = 1
2a   t1 = addr(a) - 4
2b   t2 = i * 4
3.   if i > n goto 15
5.   t2 = i * 4
6.   t3 = t1[t2]
10a  t7 = t3 * t3
11a  sum = sum + t7
11b  t2 = t2 + 4
13.  i = i + 1
14.  goto 3
15.  …
```

Test Elision and Induction Variable Elimination

```
1.   sum = 0
2.   i = 1
2a   t1 = addr(a) - 4
2b   t2 = i * 4
3.   if i > n goto 15
6.   t3 = t1[t2]
10a  t7 = t3 * t3
11a  sum = sum + t7
11b  t2 = t2 + 4
13.  i = i + 1
14.  goto 3
15.  …
```

```
1.   sum = 0
2.   i = 1
2a   t1 = addr(a) - 4
2b   t2 = i * 4
2c   t9 = n * 4
3.   if i > n goto 15
3a   if t2 > t9 goto 15
6.   t3 = t1[t2]
10a  t7 = t3 * t3
11a  sum = sum + t7
11b  t2 = t2 + 4
13.  i = i + 1
14.  goto 3a
15.  …
```

Constant Propagation and Dead Code Elimination

```
1.   sum = 0
2.   i = 1
2a   t1 = addr(a) - 4
2b   t2 = i * 4
2c   t9 = n * 4
3a   if t2 > t9 goto 15
6.   t3 = t1[t2]
10a  t7 = t3 * t3
11a  sum = sum + t7
11b  t2 = t2 + 4
14.  goto 3a
15.  …
```

```
1.   sum = 0
2.   i = 1
2a   t1 = addr(a) - 4
2b   t2 = i * 4
2c   t9 = n * 4
2d   t2 = 4
3a   if t2 > t9 goto 15
6.   t3 = t1[t2]
10a  t7 = t3 * t3
11a  sum = sum + t7
11b  t2 = t2 + 4
14.  goto 3a
15.  …
```

New Control Flow Graph