CMPT 379
Compilers

Anoop Sarkar

http://www.cs.sfu.ca/~anoop

Code Optimization

- There is no fully optimizing compiler *O*
- Let's assume *O* exists: it takes a program P and produces output **Opt**(P) which is the *smallest* possible
- Imagine a program Q that produces no output and never terminates, then **Opt**(Q) could be:
  ```
  L1: goto L1
  ```
- Then to check if a program P never terminates on some inputs, check if **Opt**(P) is equal to **Opt**(Q)
- Full Employment Theorem for Compiler Writers, see Rice(1953)

Optimizations

- Non-Optimizations
- Types of optimizations
- Correctness of optimizations
  - Optimizations must not change the meaning of the program
- Amdahl’s Law
- Moore’s Law

Non-Optimizations

```
enum { GOOD, BAD };
extern int test_condition();

void check() {
  int rc;

  rc = test_condition();
  if (rc != GOOD) {
    exit(rc);
  }
}
```

```
enum { GOOD, BAD };
extern int test_condition();

void check() {
 int rc;

 if ((rc = test_condition())) {
   exit(rc);
 }
}
```

Which version of check runs faster?

Types of Optimizations

- High-level optimizations
  - function inlining
- Machine-dependent optimizations
  - e.g., peephole optimizations, instruction scheduling
- Local optimizations or Transformations
  - within basic block
- Global optimizations or Data flow Analysis
  - across basic blocks
  - within one procedure (intraprocedural)
  - whole program (interprocedural)

Maintaining Correctness

- What does this program output?

3

Not:

$ decafcc byzero.decaf

Floating exception

```
void main() {
    int x;
    if (false) {
        x = 3/(3-3);
    } else {
        x = 3;
    }
    callout("print_int", x);
}
```

Peephole Optimization

- Redundant instruction elimination
  - If two instructions perform that same function ***and*** are in the same basic block, remove one
  - Redundant loads and stores
  - Unreachable code
- Flow control optimization

  goto _L1

  _L1: goto _L2

Peephole Optimization

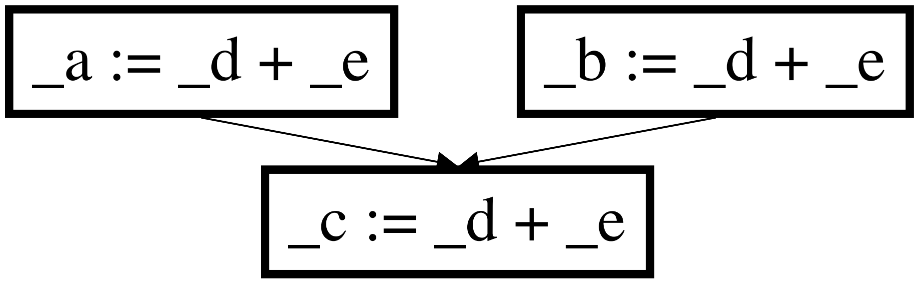
- Algebraic simplification
- Reduction in strength
  - Use faster instructions whenever possible
- Use of Machine Idioms

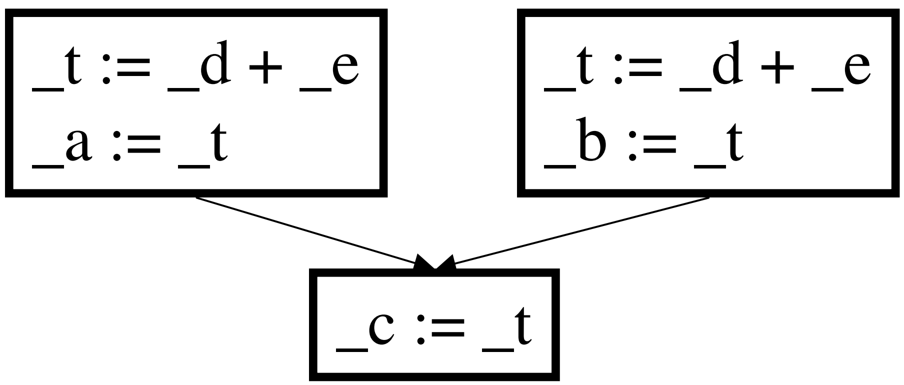
Constant folding & propagation

- Constant folding
  - compute expressions with known values at compile time
- Constant propagation
  - if constant assigned to variable, replace uses of variable with constant unless variable is reassigned

Constant folding & propagation

- Copy Propagation





Transformations

- Structure preserving transformations
  1. Common subexpression elimination

     _a := _b + _c

     _b := _a - _d

     _c := _b + _c

     _d := _a - _d (⇒ _b)
  2. Dead-code elimination
  3. Renaming temporary variables
  4. Interchange of statements

Transformations

- Algebraic transformations

  _d := _a + 0 ($\Rightarrow$ _a)

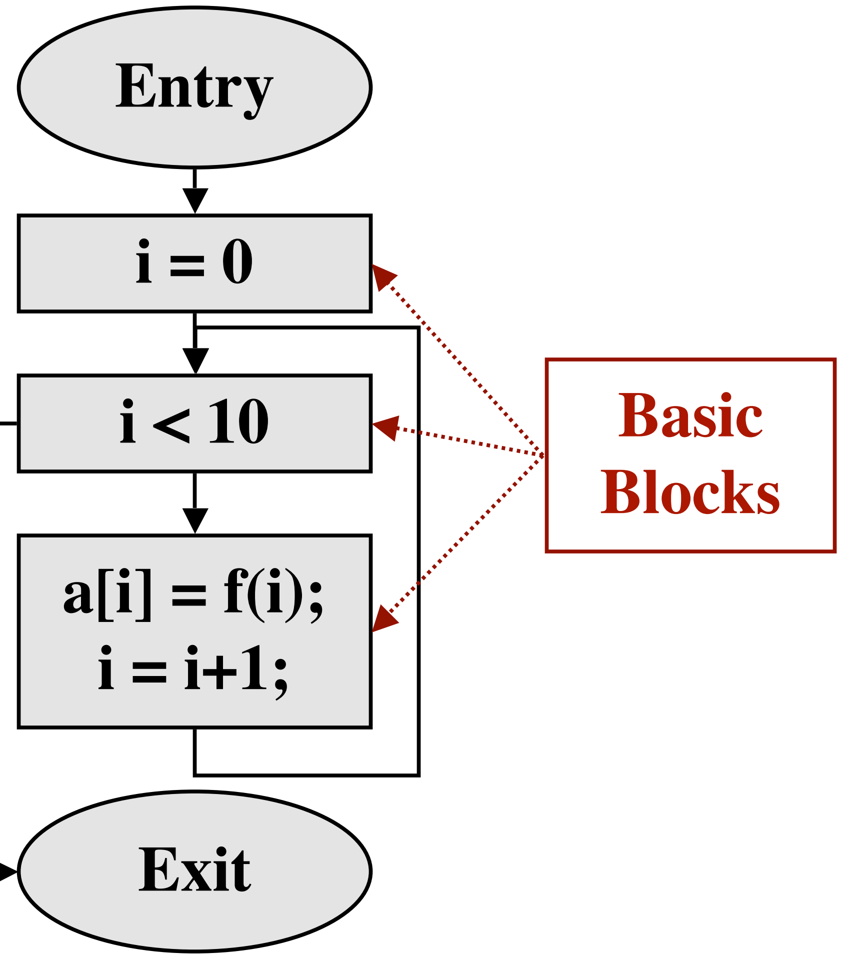
  _d := _d * 1 ($\Rightarrow$ *eliminate*)

- Reduction of strength

  _d := _a ** 2 ($\Rightarrow$ _a * _a)

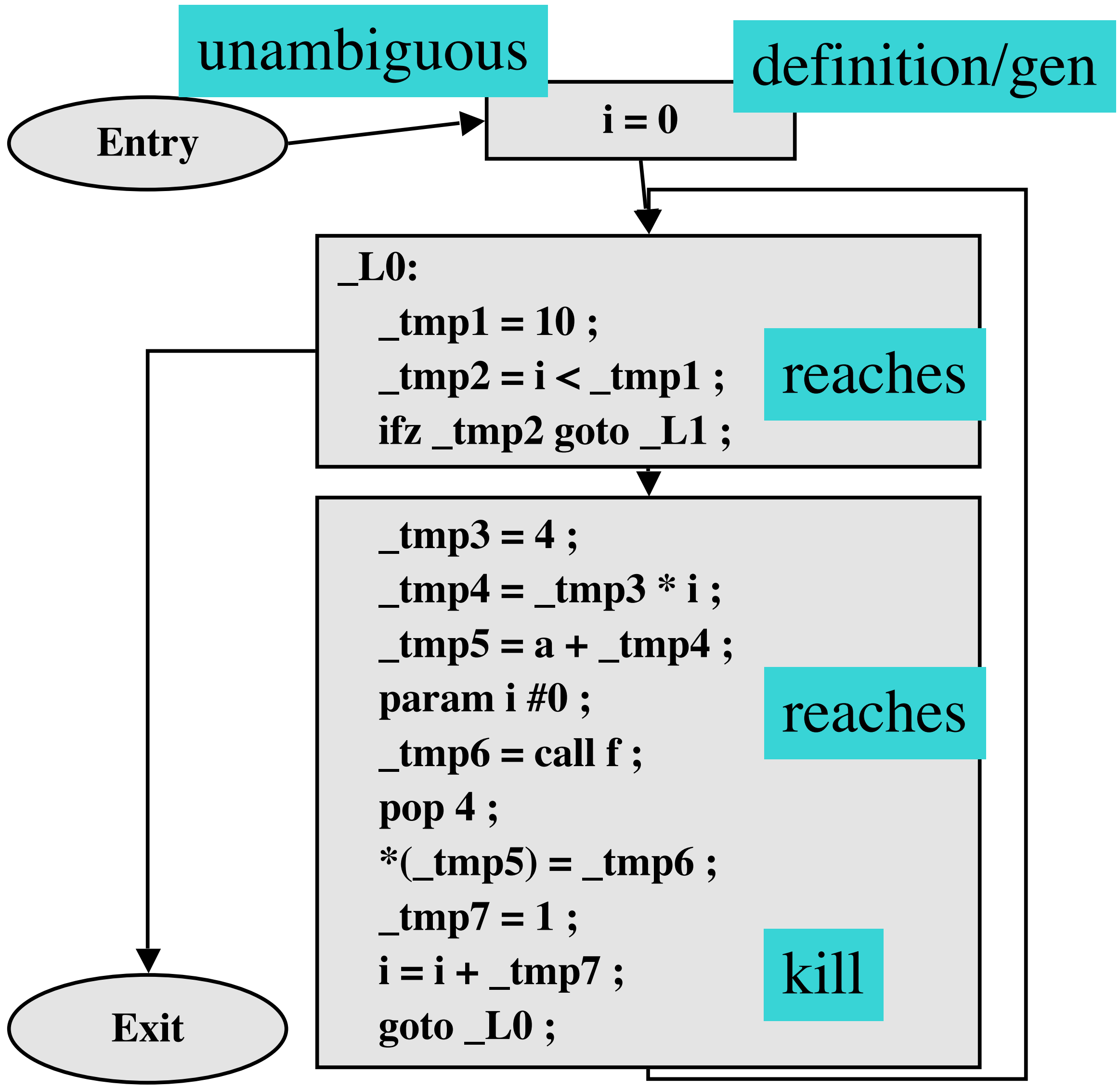
Control Flow Graph (CFG)

```
int main() {
 extern int f(int);
 int i;
 int *a;
 for (i = 0;
         i < 10;
             i = i + 1)
     { a[i] = f(i); }
}
```



Control Flow Graph in TAC

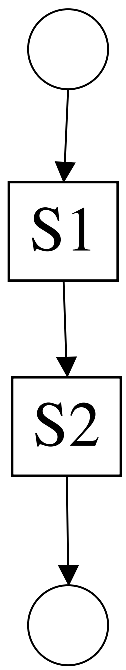
```
main:
   BeginFunc 72 ;
   i = 0 ;
_L0:
   _tmp1 = 10 ;
   _tmp2 = i < _tmp1 ;
   IfZ _tmp2 Goto _L1 ;
   _tmp3 = 4 ;
   _tmp4 = _tmp3 * i ;
   _tmp5 = a + _tmp4 ;
   param i #0 ;
   _tmp6 = call f ;
   pop 4 ;
   *(_tmp5) = _tmp6 ;
   _tmp7 = 1 ;
   i = i + _tmp7 ;
   goto _L0 ;
_L1:
   EndFunc ;
```



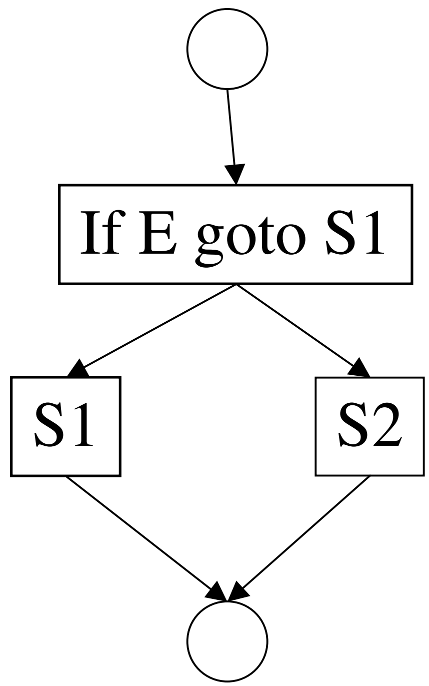
Dataflow Analysis

- $S \rightarrow id := E$
- $S \rightarrow S ; S$
- $S \rightarrow$ if E then S else S
- $S \rightarrow$ do S while E
- $E \rightarrow id + id$
- $E \rightarrow id$

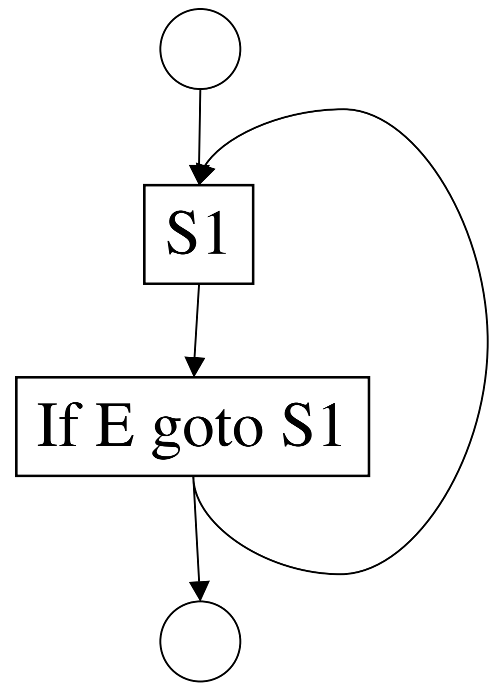
Dataflow Analysis



S ; S

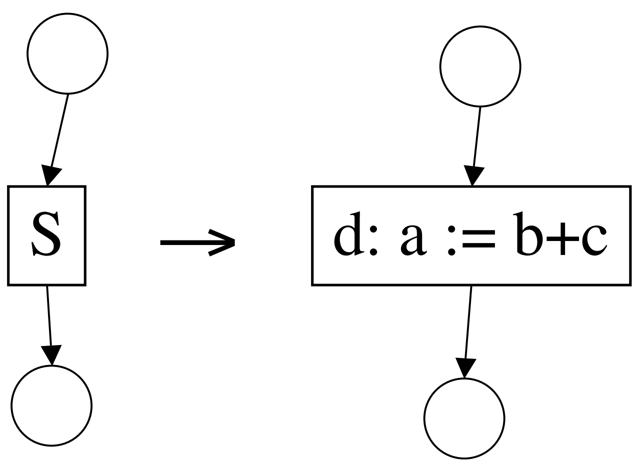


if E then S else S



do S while E

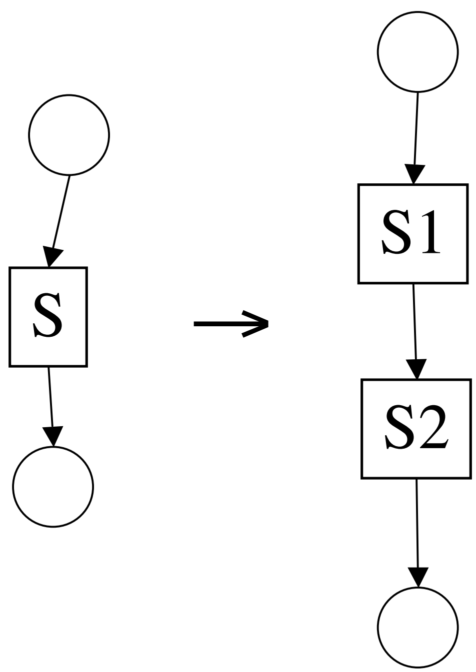
Reaching definitions



gen[S] = { d }
kill[S] = Def(a) - { d }

out[S] = gen[S] ∪ (in[S] - kill[S])

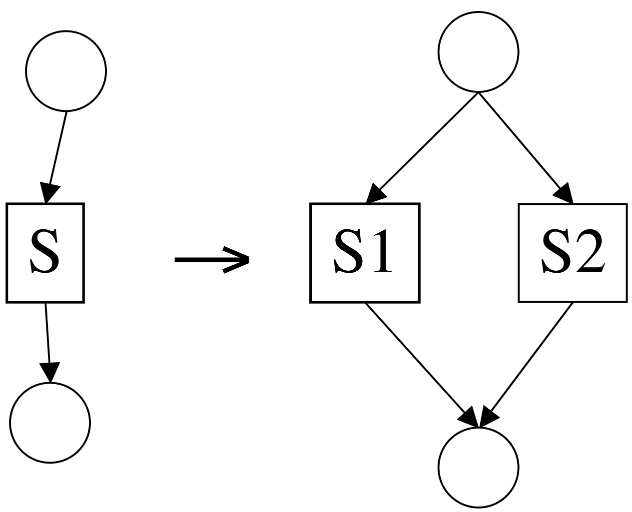
Reaching definitions



$gen[S] = gen[S2] \cup (gen[S1] - kill[S2])$
$kill[S] = kill[S2] \cup (kill[S1] - gen[S2])$

$in[S1] = in[S]$
$in[S2] = out[S1]$
$out[S] = out[S2]$

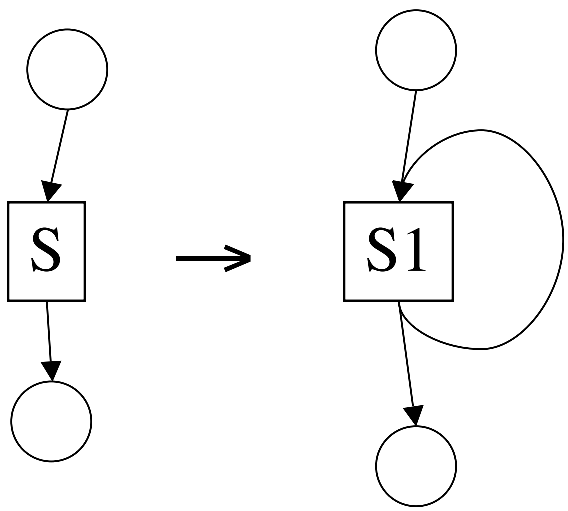
Reaching definitions



gen[S] = gen[S1] ∪ gen[S2]
kill[S] = kill[S1] ∩ (kill[S1] - gen[S2])

in[S1] = in[S]
in[S2] = in[S]
out[S] = out[S1] ∪ out[S2]

Reaching definitions



gen[S] = gen[S1]
kill[S] = kill[S1]

in[S1] = in[S] ∪ gen[S1]
out[S] = out[S1]

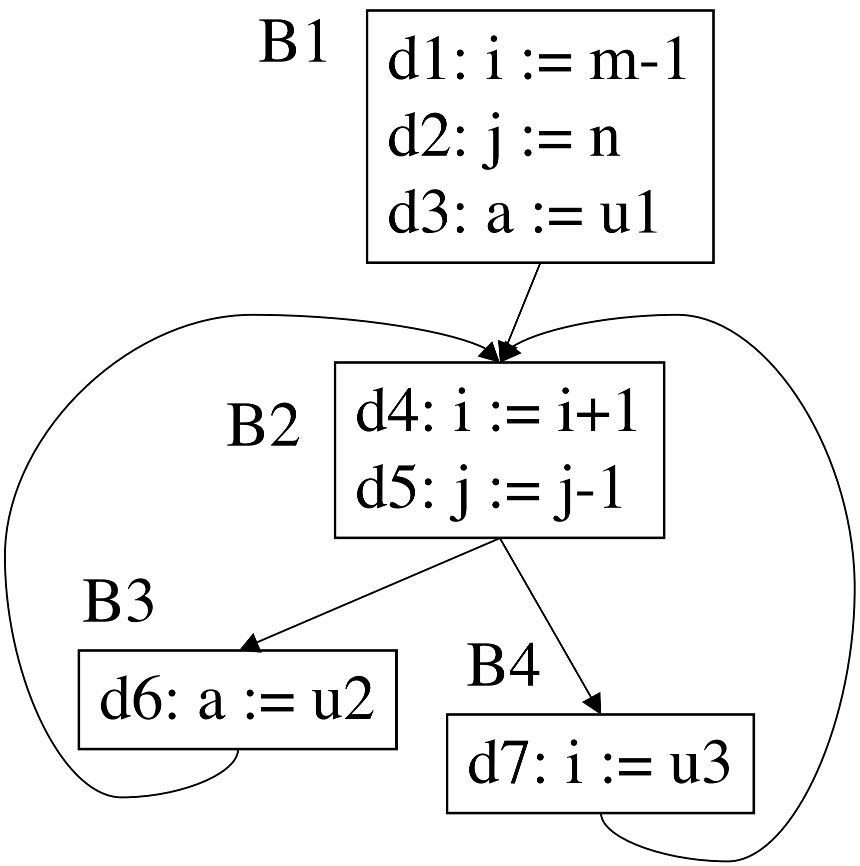
Iteratively find out[S] (fixed point)

out = synthesized attribute

in = inherited attribute

out[S1] = gen[S1] ∪ (in[S1] - kill[S1])

Reaching definitions



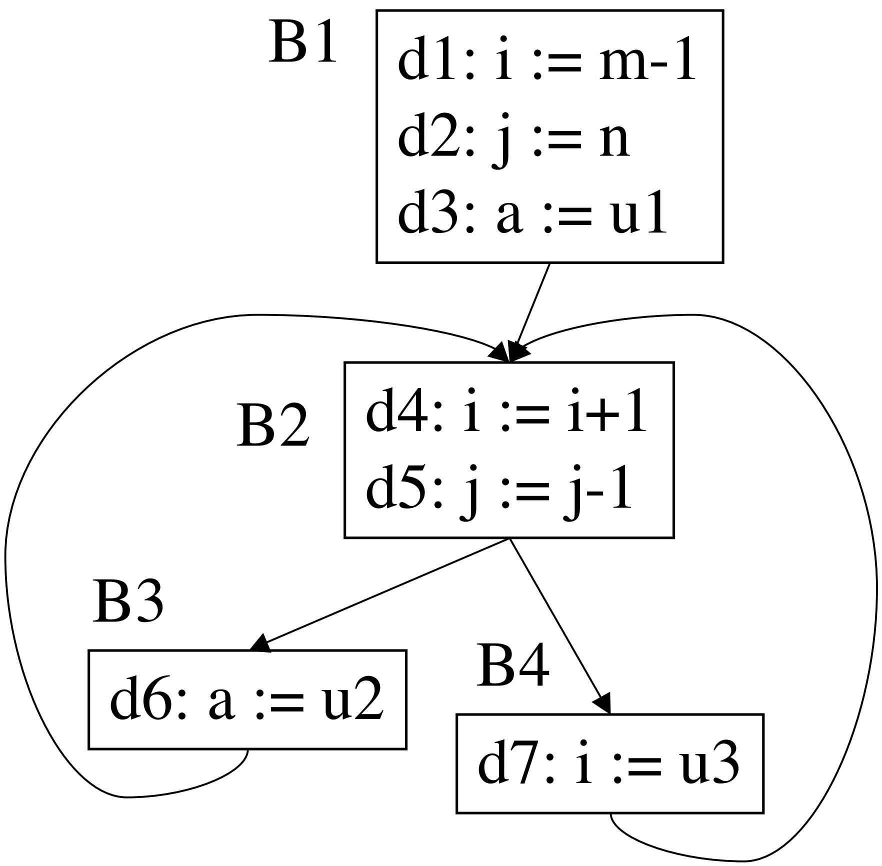
gen[B1] = { d1, d2, d3 }
kill[B1] = { d4, d5, d6, d7 }

gen[B2] = { d4, d5 }
kill[B2] = { d1, d2, d7 }

gen[B3] = { d6 }
kill[B3] = { d3 }

gen[B4] = { d7 }
kill[B4] = { d1, d4 }

Reaching definitions



gen[B1] = { d1, d2, d3 }
kill[B1] = { d4, d5, d6, d7 }

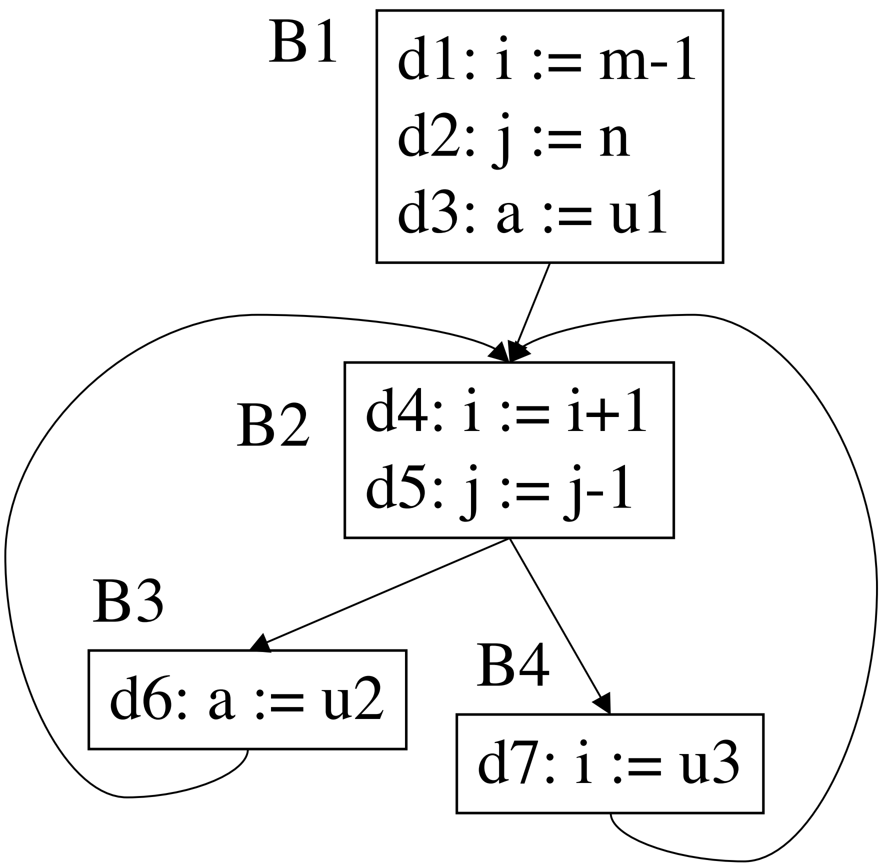
gen[B2] = { d4, d5 }
kill[B2] = { d1, d2, d7 }

gen[B3] = { d6 }
kill[B3] = { d3 }

gen[B4] = { d7 }
kill[B4] = { d1, d4 }

in[B2] = out[B1] ∪ out[B3] ∪ out[B4]

Reaching definitions



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 }

√ out[B2] = gen[B2] ∪ (in[B3] - kill[B2])
out[B2] = gen[B2] ∪ (in[B4] - kill[B2])

Dataflow Analysis

- Compute Dataflow Equations over Control Flow Graph
  - Reaching Definitions (Forward)

    out[BB] := gen[BB] ∪ (in[BB] – kill[BB])

    in[BB] := ∪ out[s] : forall s ∈ pred[BB]
  - Liveness Analysis (Backward)

    in[BB] := use[BB] ∪ (out[BB] – def[BB])

    out[BB] := ∪ in[s] : forall s ∈ succ[BB]
- Computation by fixed-point analysis

Amdahl's Law

- $\text{Speedup}_{\text{total}} =$
  $((1 - \text{Time}_{\text{Fractionoptimized}}) +$
  $\text{Time}_{\text{Fractionoptimized}}/\text{Speedup}_{\text{optimized}})^{-1}$
- Optimize the common case, 90/10 rule
- Requires quantitative approach
  - Profiling + Benchmarking
- Problem: Compiler writer doesn't know the application beforehand

Moore’s Law

- Speed per $ doubles every 18 months
- How long do you have to wait until a new processor obsoletes your +5% performance improvement?
- And how does that feel if the optimization was machine-specific?)

Wrap Up

- Analysis/Synthesis
  - Translation from string to executable
- Divide and conquer
  - Build one component at a time
  - Theoretical analysis will ensure we keep things **simple** and **correct**
  - Create a complex piece of software

Lecture 1: Why take this class?

- To learn parsing techniques that can be applied elsewhere
- To see how theoretical algorithms are applied in practice
- To gain an appreciation for language design
- To observe important aspects of software engineering