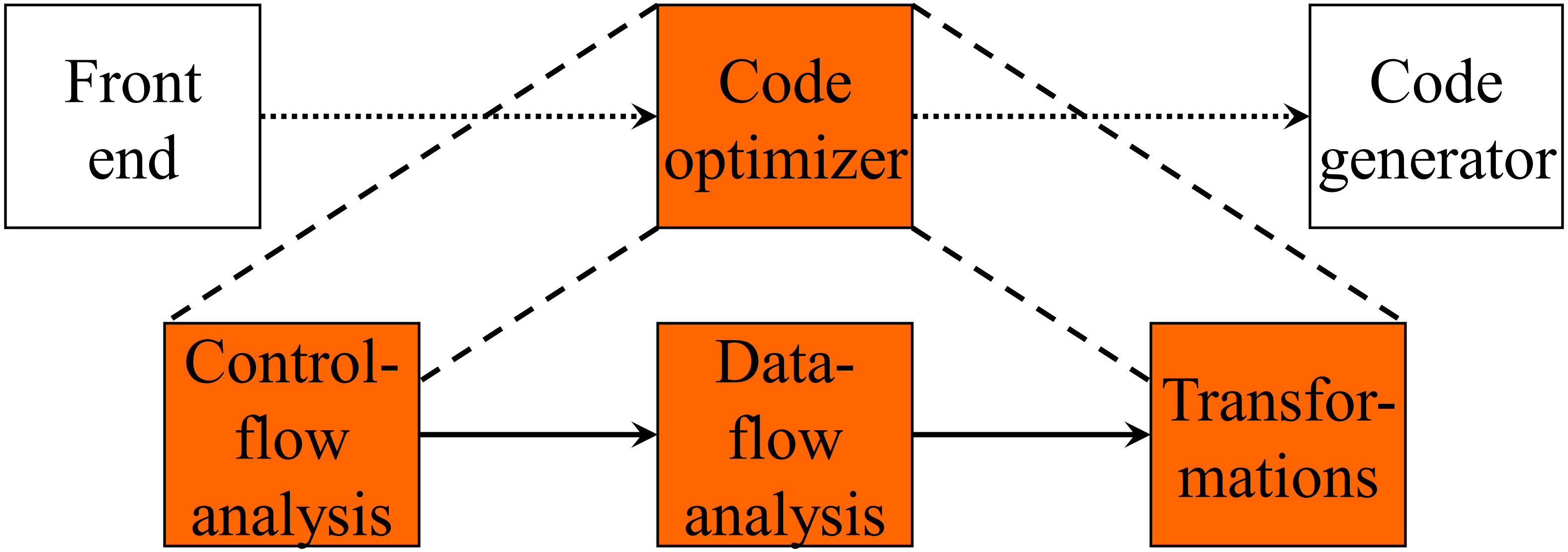
Code Optimization

Bart Kienhuis
Computer Systems Group
University Leiden (LIACS)

The Code Optimizer

- Control flow analysis: CFG (Ch. 9)
- Data-flow analysis
- Transformations



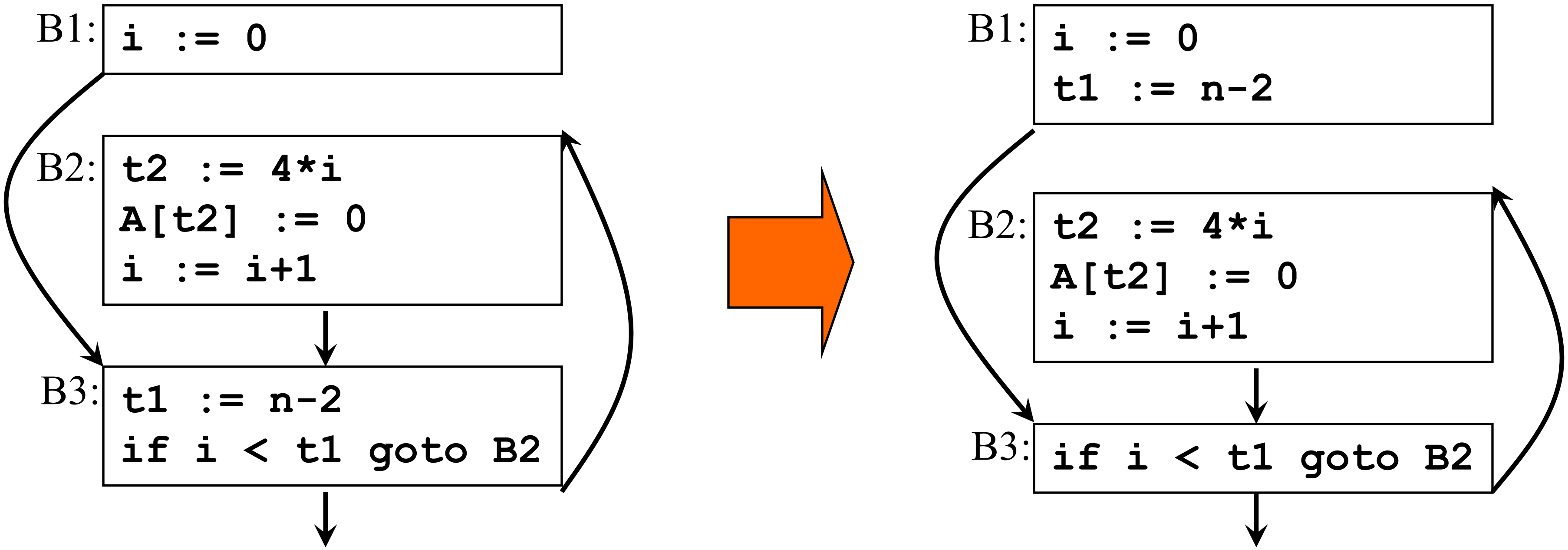
Code Optimizations

- Local/global common subexpression elimination
- Dead-code elimination
- Instruction reordering
- Constant folding
- Algebraic transformations
- Copy propagation
- *Loop optimizations*

Loop Optimizations

- Code motion
- Induction variable elimination
- Reduction in strength
- … lots more

Code Motion



B1:
```
i  := 0
```

B2:
```
t2  := 4*i
A[t2]  := 0
i  := i+1
```

B3:
```
t1  := n-2
if i < t1 goto B2
```

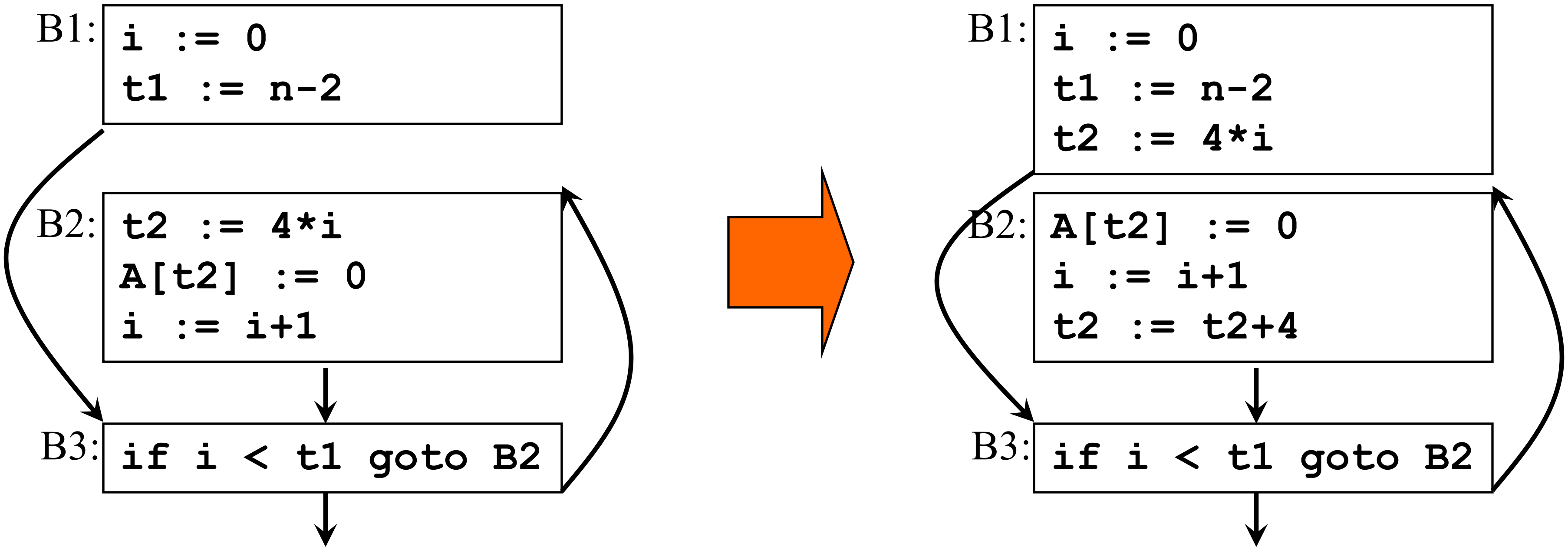
B1:
```
i  := 0
t1  := n-2
```

B2:
```
t2  := 4*i
A[t2]  := 0
i  := i+1
```

B3:
```
if i < t1 goto B2
```

Move *loop-invariant computations* before the loop

Strength Reduction



```
B1: i  := 0
    t1  := n-2

B2: t2  := 4*i
    A[t2]  := 0
    i  := i+1

B3: if i < t1 goto B2
```

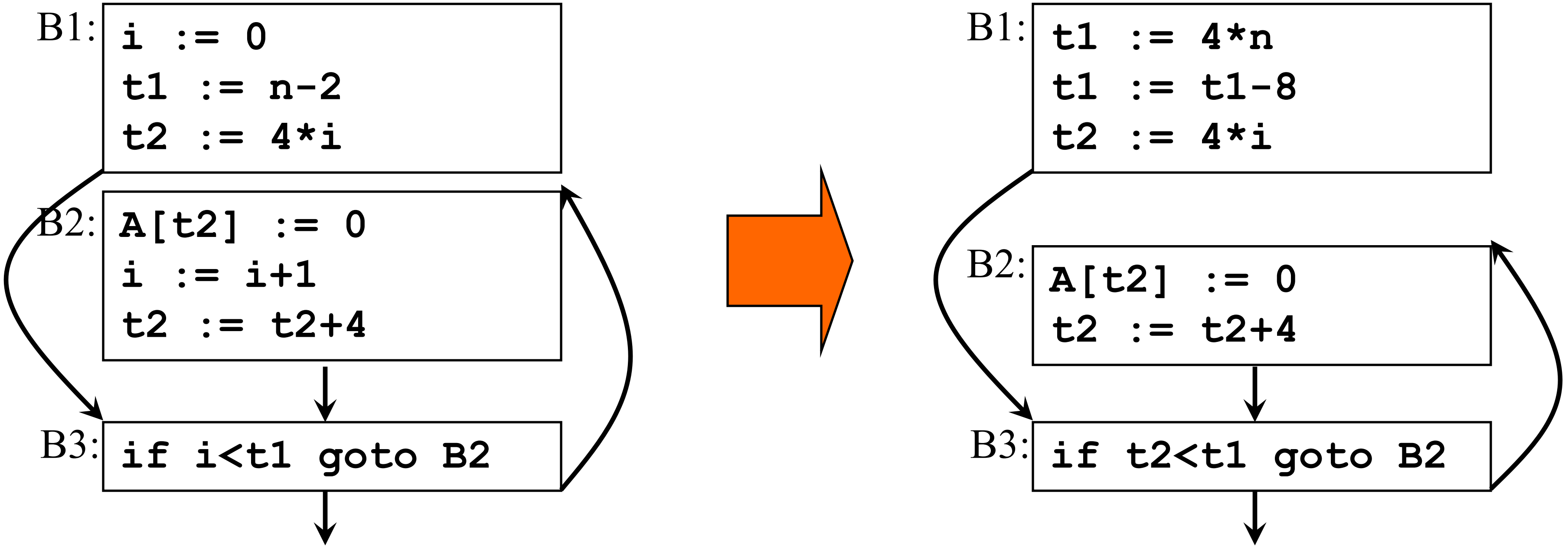
```
B1: i  := 0
    t1  := n-2
    t2  := 4*i

B2: A[t2]  := 0
    i  := i+1
    t2  := t2+4

B3: if i < t1 goto B2
```

Replace expensive computations with *induction variables*

Reduction Variable Elimination



B1:
```
i  := 0
t1 := n-2
t2 := 4*i
```

B2:
```
A[t2] := 0
i  := i+1
t2 := t2+4
```

B3:
```
if i<t1 goto B2
```

B1:
```
t1 := 4*n
t1 := t1-8
t2 := 4*i
```

B2:
```
A[t2] := 0
t2 := t2+4
```

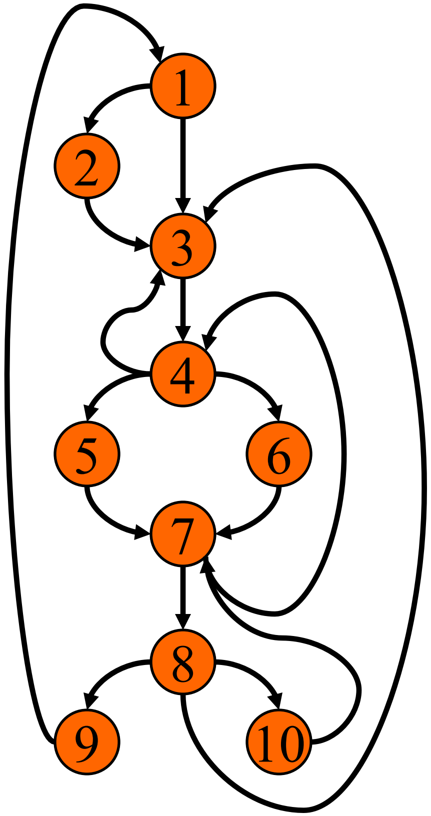
B3:
```
if t2<t1 goto B2
```

Replace induction variable in expressions with another

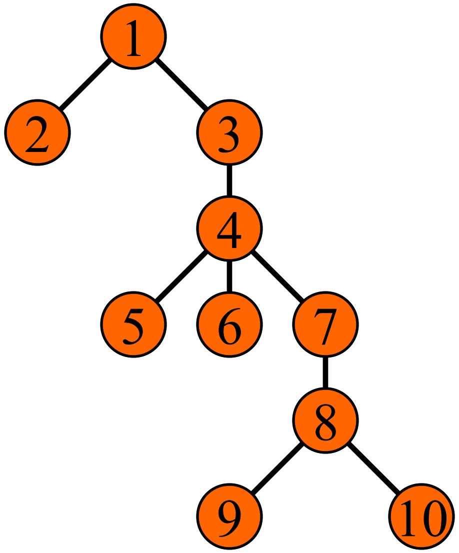
Determining Loops in Flow Graphs: Dominators

- Dominators: *d dom n*
  - Node *d* of a CFG *dominates* node *n* if *every* path from the initial node of the CFG to *n* goes through *d*
  - The loop entry dominates all nodes in the loop
- The *immediate dominator m* of a node *n* is the last dominator on the path from the initial node to *n*
  - If $d \neq n$ and *d dom n* then *d dom m*

Dominator Trees



CFG

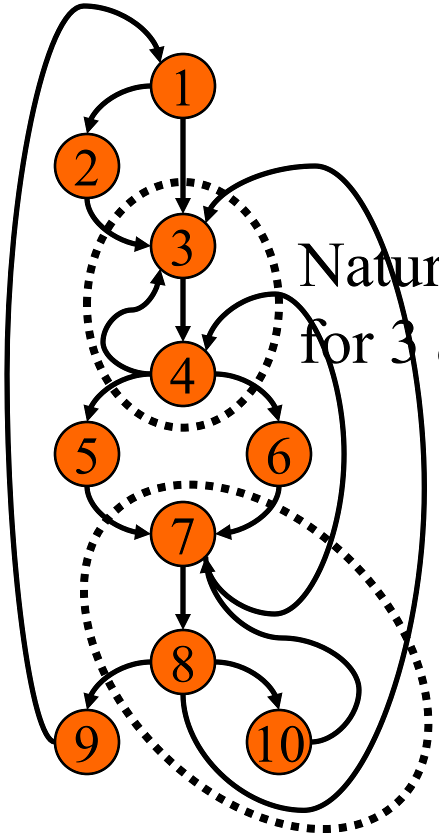


Dominator tree

Natural Loops

- A *back edge* is is an edge $a \rightarrow b$ whose head b dominates its tail a
- Given a back edge $n \rightarrow d$
  - The *natural loop* consists of d plus the nodes that can reach n without going through d
  - The *loop header* is node d
- Unless two loops have the same header, they are disjoint or one is nested within the other
  - A nested loop is an *inner loop* if it contains no other loops

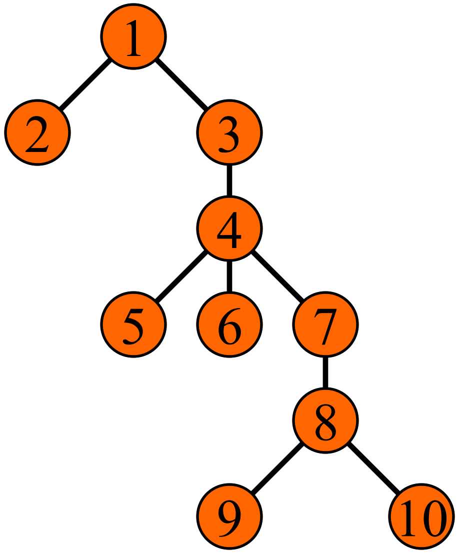
Natural (Inner) Loops Example



Natural loop
for 3 *dom* 4

Natural loop
for 7 *dom* 10

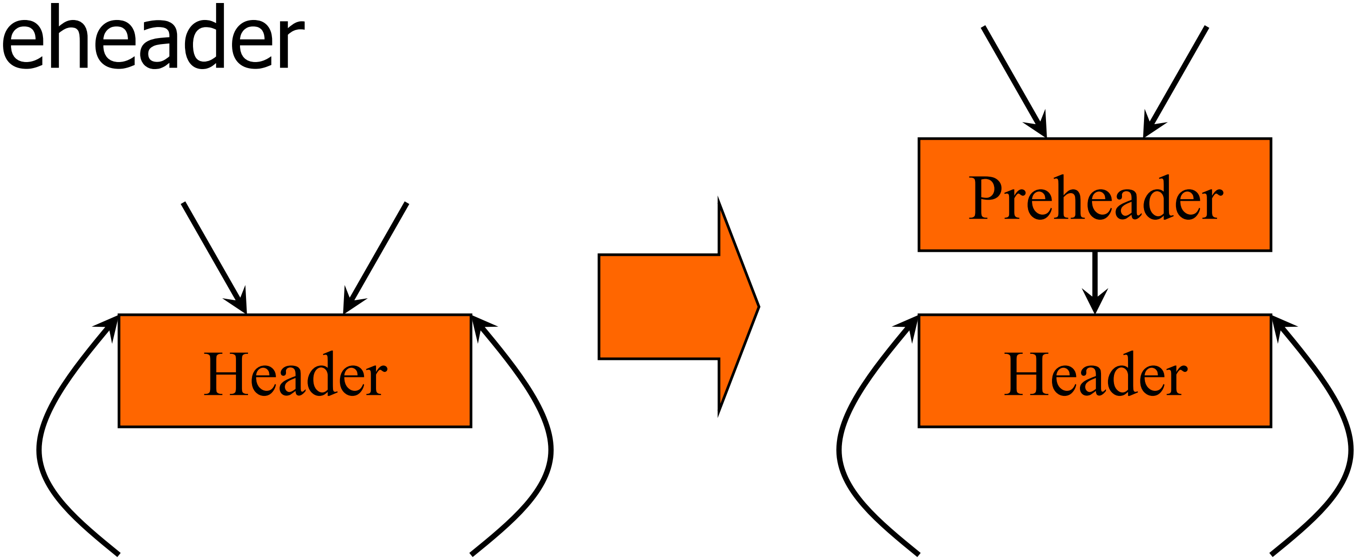
CFG



Dominator tree

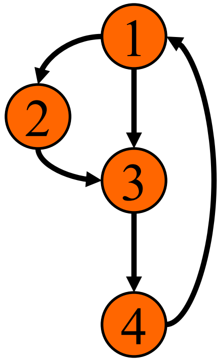
Pre-Headers

- To facilitate loop transformations, a compiler often adds a *preheader* to a loop
- Code motion, strength reduction, and other loop transformations populate the preheader

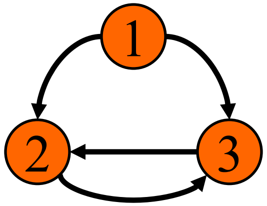


Reducible Flow Graphs

- *Reducible graph* = disjoint partition in forward and back edges such that the forward edges form an acyclic (sub)graph



Example of a reducible CFG

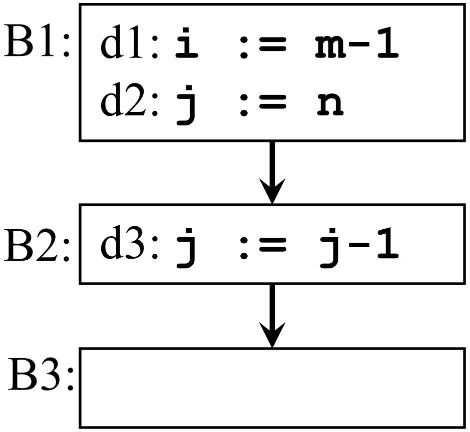


Example of a nonreducible CFG

Global Data-Flow Analysis

- To apply global optimizations on basic blocks, *data-flow information* is collected by solving systems of *data-flow equations*
- Suppose we need to determine the *reaching definitions* for a sequence of statements *S*

$$out[S] = gen[S] \cup (in[S] - kill[S])$$



B1: d1: `i := m-1`
d2: `j := n`

B2: d3: `j := j-1`

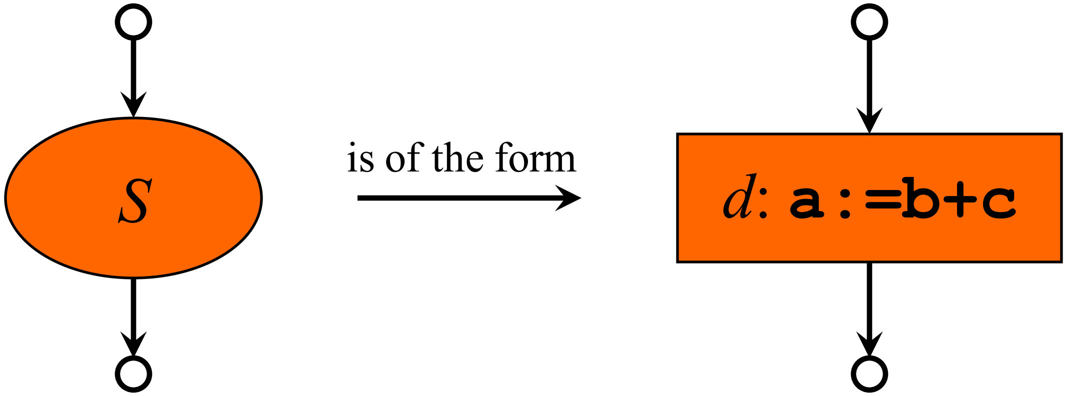
B3:

$out[\mathrm{B1}] = gen[\mathrm{B1}] = \{\mathrm{d1}, \mathrm{d2}\}$

$out[\mathrm{B2}] = gen[\mathrm{B2}] \cup \{\mathrm{d1}\} = \{\mathrm{d1}, \mathrm{d3}\}$

d1 reaches B2 and B3 and
d2 reaches B2, but not B3
because d2 is killed in B2

Reaching Definitions


S

is of the form

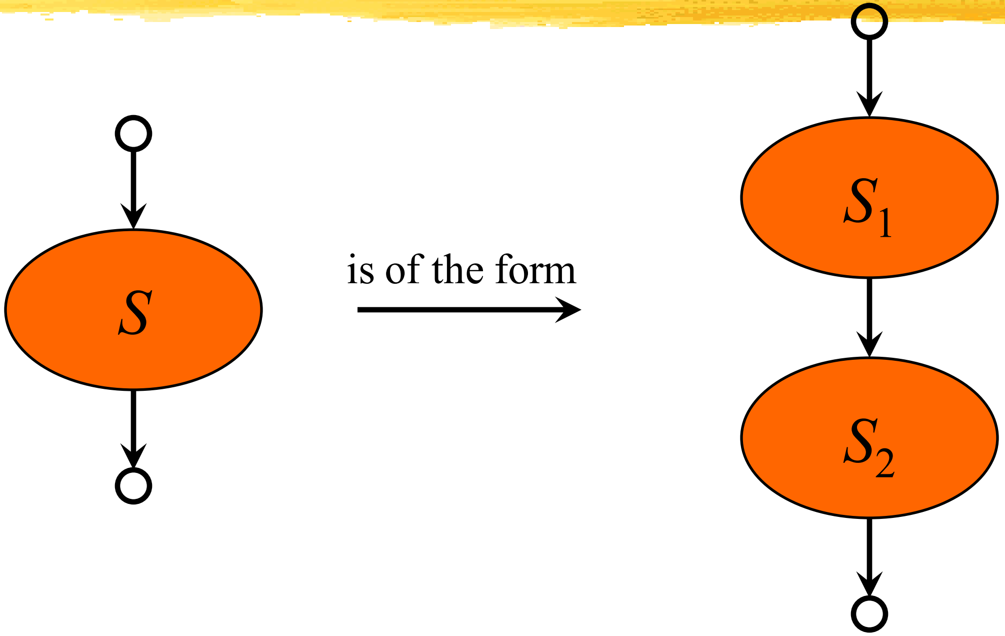
d: `a:=b+c`

Then, the data-flow equations for S are:

$gen[S] = \{d\}$
$kill[S] = D_{\mathtt{a}} - \{d\}$
$out[S] = gen[S] \cup (in[S] - kill[S])$

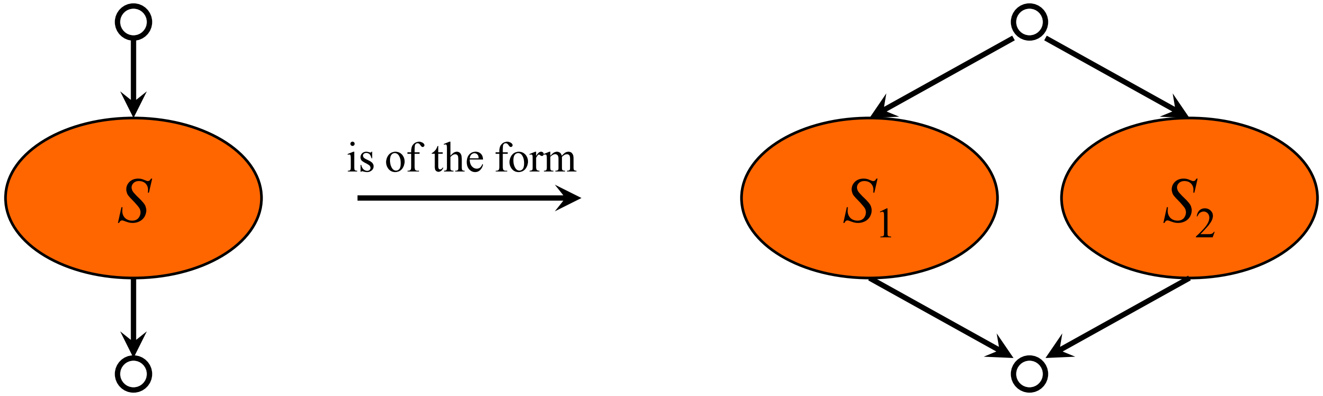
where $D_{\mathtt{a}}$ = all definitions of **`a`** in the region of code

Reaching Definitions



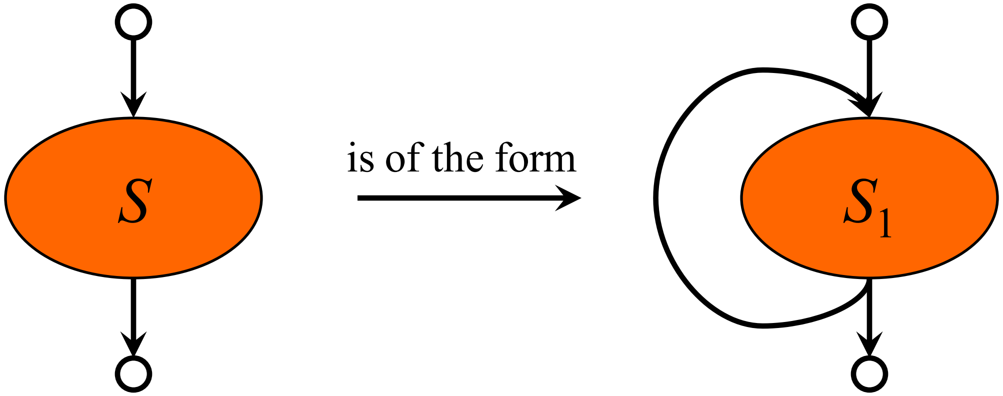
$$\begin{aligned}
gen[S] &= gen[S_2] \cup (gen[S_1] - kill[S_2]) \\
kill[S] &= kill[S_2] \cup (kill[S_1] - gen[S_2]) \\
in[S_1] &= in[S] \\
in[S_2] &= out[S_1] \\
out[S] &= out[S_2]
\end{aligned}$$

Reaching Definitions



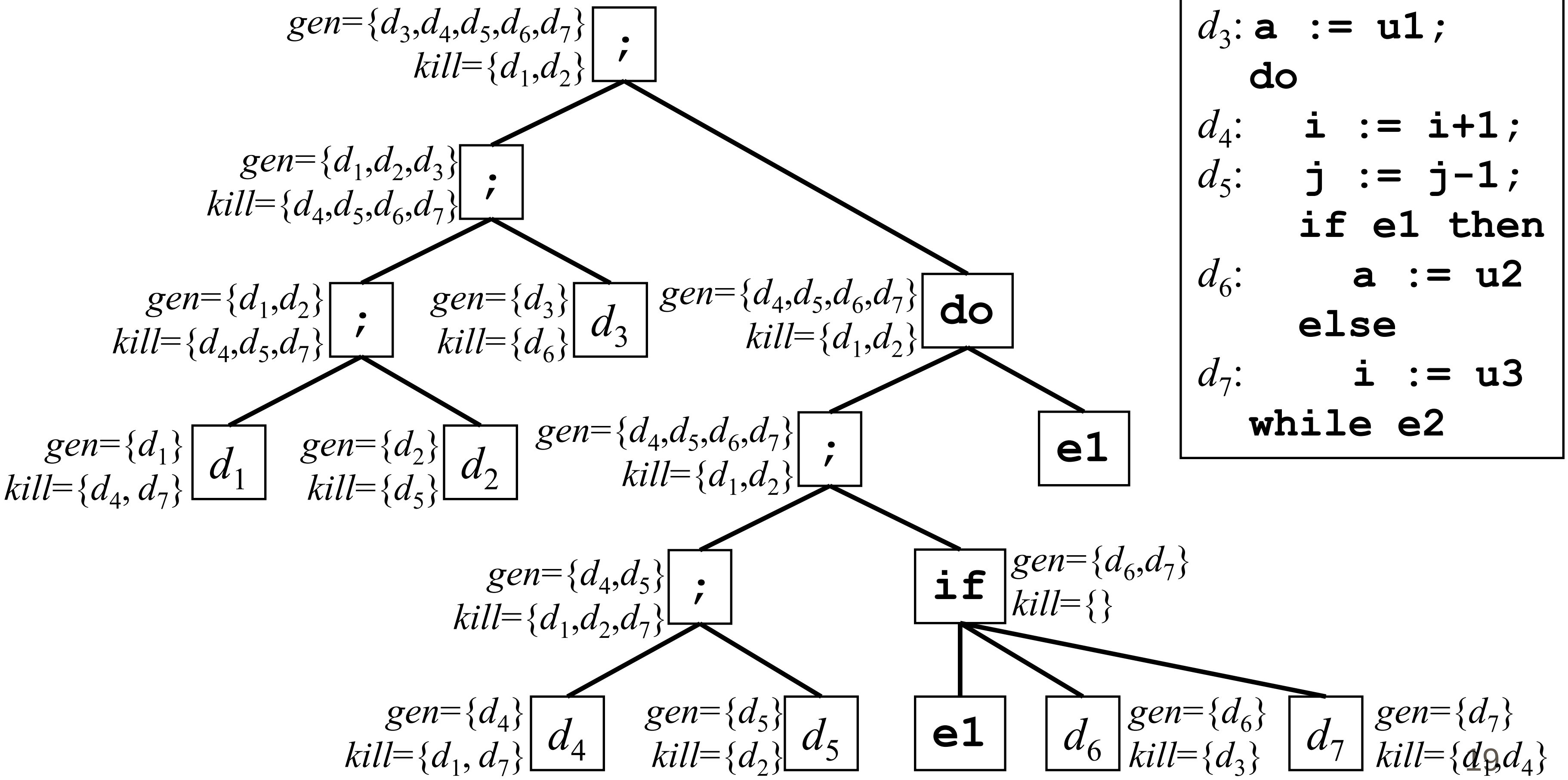
$$
\begin{aligned}
gen[S] &= gen[S_1] \cup gen[S_2] \\
kill[S] &= kill[S_1] \cap kill[S_2] \\
in[S_1] &= in[S] \\
in[S_2] &= in[S] \\
out[S] &= out[S_1] \cup out[S_2]
\end{aligned}
$$

Reaching Definitions



$$
\begin{aligned}
gen[S] &= gen[S_1] \\
kill[S] &= kill[S_1] \\
in[S_1] &= in[S] \cup gen[S_1] \\
out[S] &= out[S_1]
\end{aligned}
$$

Example Reaching Definitions



; gen=$\{d_3,d_4,d_5,d_6,d_7\}$ kill=$\{d_1,d_2\}$

; gen=$\{d_1,d_2,d_3\}$ kill=$\{d_4,d_5,d_6,d_7\}$

; gen=$\{d_1,d_2\}$ kill=$\{d_4,d_5,d_7\}$

d_3 gen=$\{d_3\}$ kill=$\{d_6\}$

do gen=$\{d_4,d_5,d_6,d_7\}$ kill=$\{d_1,d_2\}$

d_1 gen=$\{d_1\}$ kill=$\{d_4, d_7\}$

d_2 gen=$\{d_2\}$ kill=$\{d_5\}$

; gen=$\{d_4,d_5,d_6,d_7\}$ kill=$\{d_1,d_2\}$

e1

; gen=$\{d_4,d_5\}$ kill=$\{d_1,d_2,d_7\}$

if gen=$\{d_6,d_7\}$ kill=$\{\}$

d_4 gen=$\{d_4\}$ kill=$\{d_1, d_7\}$

d_5 gen=$\{d_5\}$ kill=$\{d_2\}$

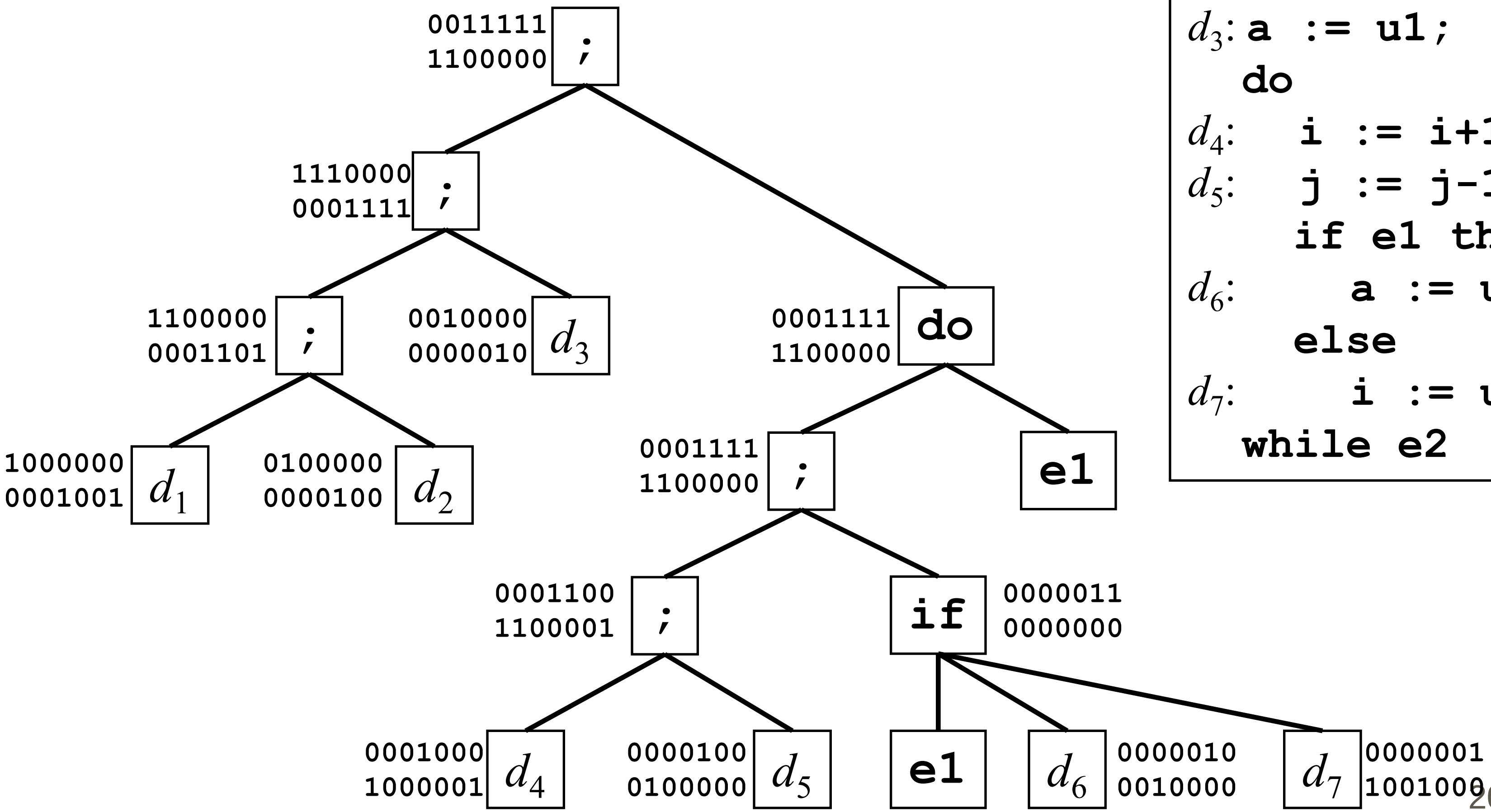
e1

d_6 gen=$\{d_6\}$ kill=$\{d_3\}$

d_7 gen=$\{d_7\}$ kill=$\{d_1,d_4\}$

```
$d_1$: i := m-1;
$d_2$: j := n;
$d_3$: a := u1;
       do
$d_4$:    i := i+1;
$d_5$:    j := j-1;
          if e1 then
$d_6$:       a := u2
          else
$d_7$:       i := u3
       while e2
```

Using Bit-Vectors to Compute Reaching Definitions



```
$d_1$: i  := m-1;
$d_2$: j  := n;
$d_3$: a  := u1;
   do
$d_4$:    i := i+1;
$d_5$:    j := j-1;
          if e1 then
$d_6$:       a := u2
          else
$d_7$:       i := u3
   while e2
```

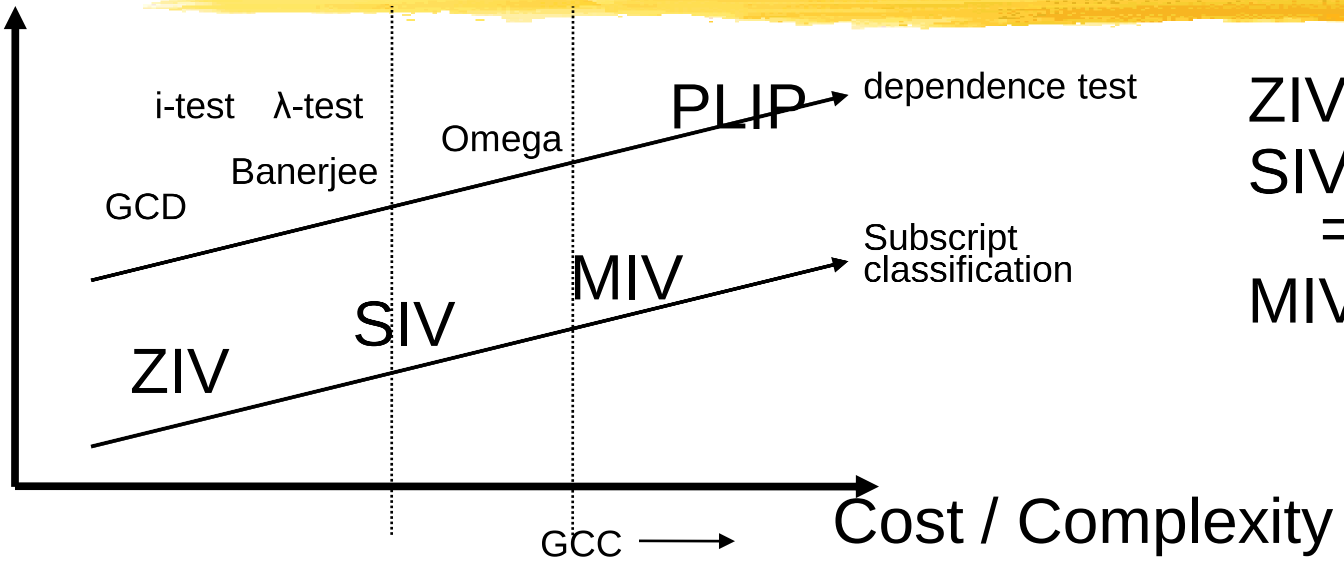
QR Algorithm – Smart Antenna

Matlab Code (QR Algorithm)

```
%parameter N 8 16;
%parameter K 100 1000;

for k = 1:1:K,
    for j = 1:1:N,
        [ r(j,j), x(k,j), t ]=Vectorize( r(j,j), x(k,j) );
        for i = j+1:1:N,
             [ r(j,i), x(k,i), t]=Rotate( r(j,i), x(k,i), t );
       end
    end
end
```

Data Dependence Tests



ZIV: a[3] = a[5]

SIV: a[i] = a[2], a[i] = a[i]

MIV: a[i][j] = a[j][i+j]

GCD, Banerjee, i-test, λ-test, cannot handle:

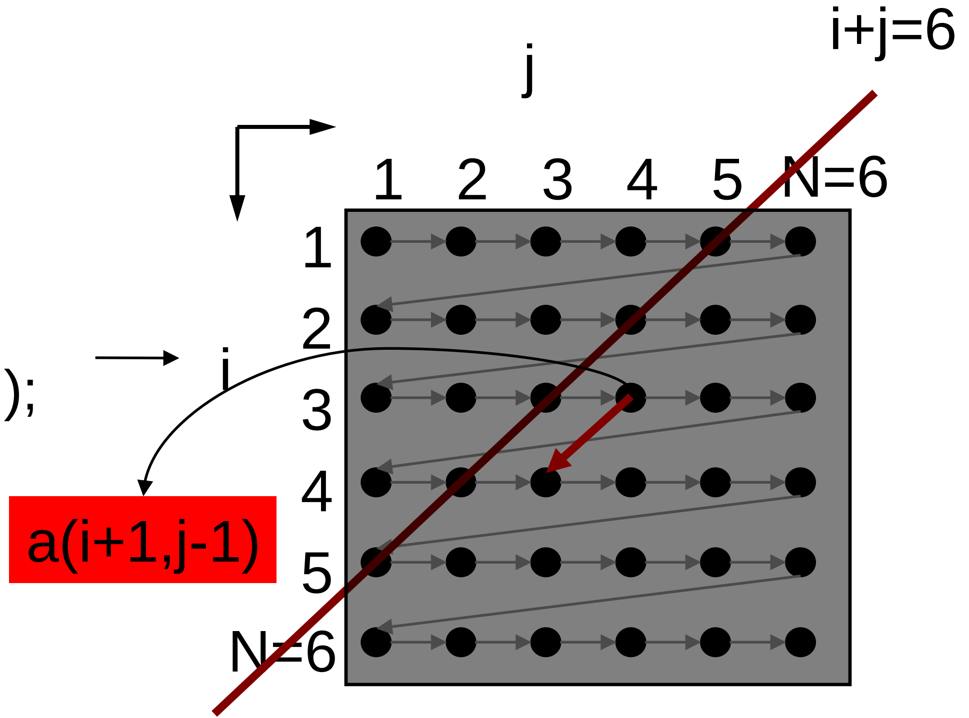
- if conditionals
- parametric loop bounds
- coupled subscripts
- parametric subscripts

Omega, PLIP:

- Exact data dependencies
- Omega: Fourier-Motzkin
- PLIP: dual-simplex method, more precise with parametric codes

Exact Dependency Analysis

```
for i= 1 : 1 : N,
    for j= 1 : 1 : N,
        [ a(i+j) ] = funcA( a(i+j) );
    end
end
```



Ax >= b (polytope)

The for-next loops define an Iteration Domain

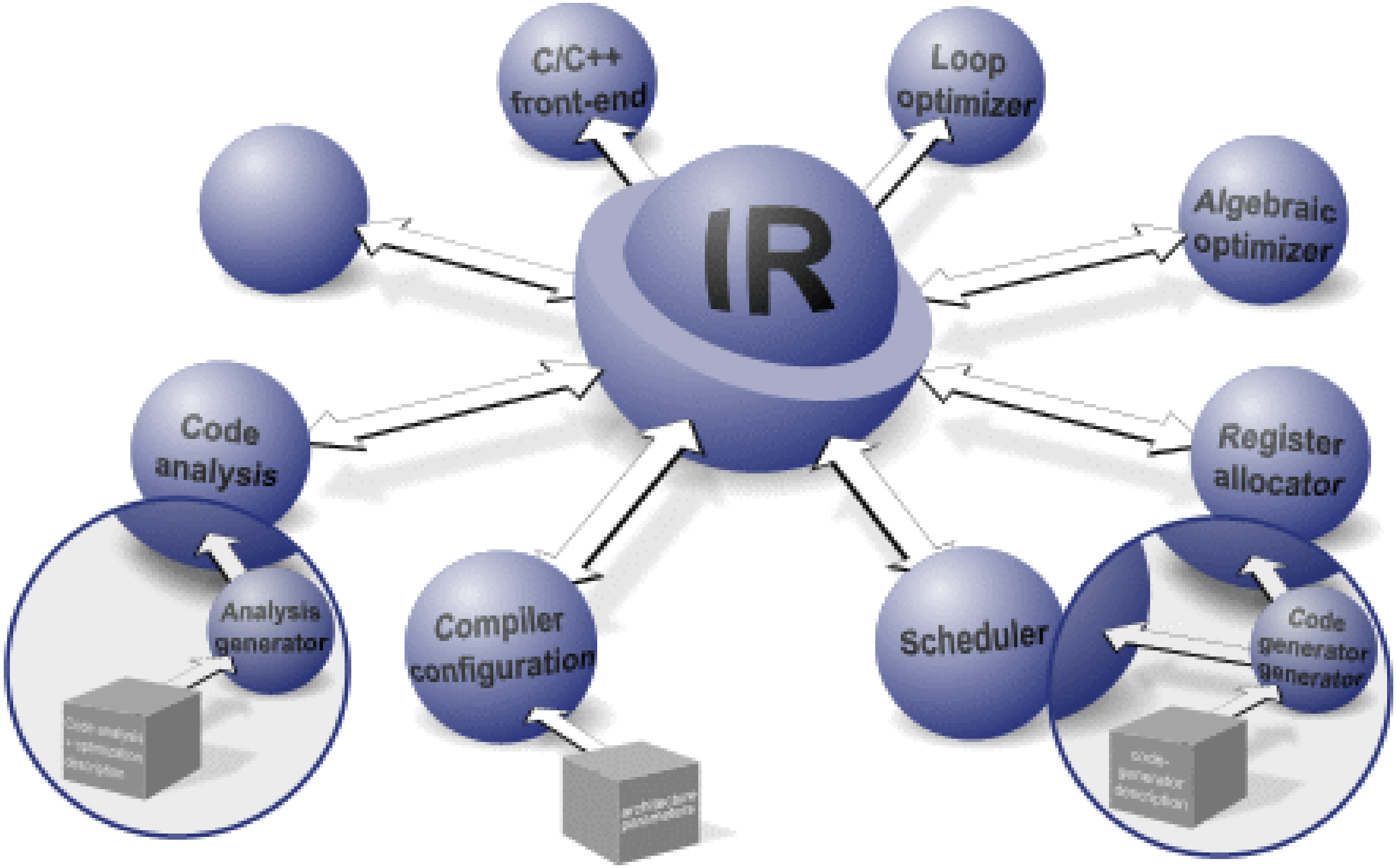
Many more optimizations

- Aliases analysis (pointers)
    - if two or more expressions denote the same memory address, the expressions are aliases of one another.

Compiler Frameworks

- Open Source
  - GCC
  - LLVM
  - Open64
  - SUIF
- Commercial
  - Target
  - Altrium
  - ACE
- In-house
  - Many

Compilers


C/C++
front-end
Loop
optimizer
IR
Algebraic
optimizer
Code
analysis
Register
allocator
Analysis
generator
Code analysis
+ optimization
description
Compiler
configuration
architecture
parameters
Scheduler
Code
generator
generator
code-
generator
description