# Program Analysis

# Program Analysis Goals

- Automatically computing information about a program
  - Checking for possible program optimizations
    = replacing the program with a "better" one
    that does "the same"
  - Generating "intelligent" error messages
  - Validating other aspects (e.g. timing)

# Elimination of Useless Assignments

```
x := 3;
a := z;
x := a;
```

⇒

```
a := z;
x := a;
```

# Loop Invariant Code Motion

```
for (i=0;i<10;i++) {
    z = r*r;
    k = k*i;
}
```

```
z = r*r;
for (i=0;i<10;i++) {
    k = k*i;
}
```

# Uninitialized Variables

```
int f(int x) {
   int a;

   if (x==0) {
      a = 2;
   }
   return (a+2);
}
```

possibly uninitialized variable

# Dereferencing NULL Pointers

```
int search(list_t *list, int x) {
while(list!=NULL) {
    list = list -> next;
    if (list -> elem == x) return 1;
}
return 0;
```

possible dereferencing of a NULL pointer

# Other Examples

- Cache analysis
- Pipeline analysis
- Stack analysis
- Safety-critical analyses
- ...

# Dynamic PA vs Static PA

- Dynamic PA:
  collecting information
  by means of testing
  - → Disadvantage:
    generated code still must cover all possible inputs
- Static PA:
  generating information
  without executing the program
  - → Optimizations hold for any given input

# Overview

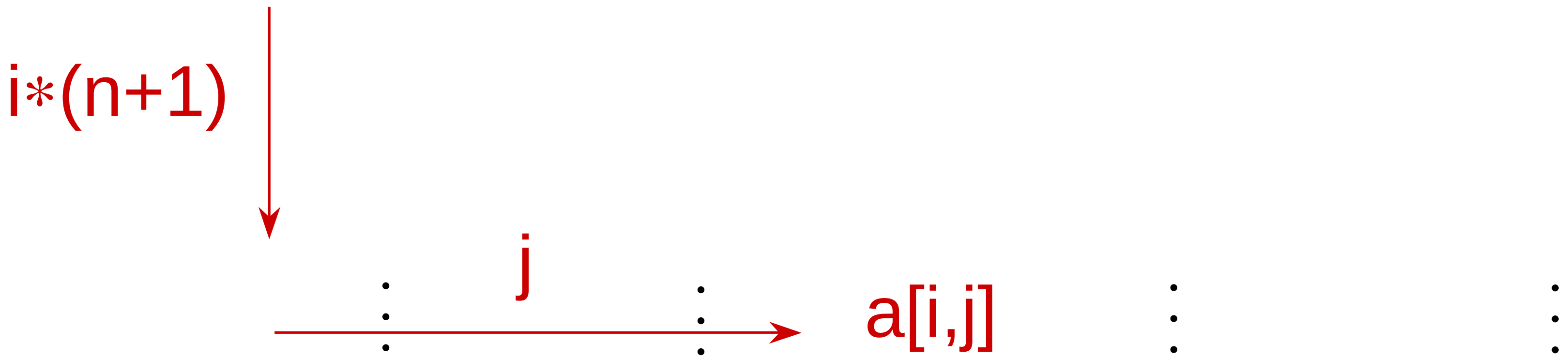
# Example

- Input:
  Pascal-like program with arrays
- Output:
  C-like program with address computation

- Elimination of redundant computations
- Simplifying address computations
- Sequence of analyses and transformations

# Input

a, b, c : array [0 : m, 0 : n] of integer




Line-wise array placement

# Output

```
i := 0;
while i <= m do
    ( j := 0;
      while j <= n do
        ( a[i,j] := b[i,j] + c[i,j];
          j := j + 1
        );
      i := i + 1
    );
```

# C-Like Arrays

```
i := 0;
while i <= m do
    ( j := 0;
      while j <= n do
        ( ta   :=   Base(a) + i*(n+1)+j;
          C(ta):= C(Base(b) + i*(n+1)+j) +
                  C(Base(c) + i*(n+1)+j) ;
          j := j + 1
        );
      i := i + 1 );
```

a[i,j]:=b[i,j]+c[i,j];

# Analysis: Available Expressions

First computation

Recomputations

```
i := 0;
while i <= m do
    ( j := 0;
      while j <= n do
        ( ta    :=   Base(a) + i*(n+1)+j;
          C(ta):= C(Base(b) + i*(n+1)+j) +
                  C(Base(c) + i*(n+1)+j) ;
          j := j + 1
        );
      i := i + 1 );
```

# Transformation: Common Subexpression Elimination

```
i := 0;
while i <= m do
    ( j := 0;
      while j <= n do
        ( t1   := i*(n+1)+j;
          ta   :=   Base(a) + t1;
          C(ta):= C(Base(b) + t1) +
                  C(Base(c) + t1) ;
          j := j + 1
        );
      i := i + 1 );
```

# Analysis: Loop Invariant Computation

```
i := 0;
while i <= m do
    ( j := 0;
      while j <= n do
        ( t1    := i*(n+1)+j;
          ta    := Base(a) + t1;
          C(ta):= C(Base(b) + t1) +
                  C(Base(c) + t1) ;
          j := j + 1
        );
      i := i + 1 );
```

# Transformation: Loop Invariant Code Motion

```
i := 0;
while i <= m do
    ( j := 0;  t2 := i*(n+1);
      while j <= n do
        ( t1   := t2 + j;
          ta   := Base(a) + t1;
          C(ta):= C(Base(b) + t1) +
                  C(Base(c) + t1) ;
             j := j + 1
        );
      i := i + 1 );
```

# Analysis: Induction Variable / Reaching Definitions

```
i := 0;
while i <= m do
    ( j := 0;  t2 := i*(n+1);
      while j <= n do
        ( t1   := t2 + j;
          ta   :=   Base(a) + t1;
          C(ta):= C(Base(b) + t1) +
                  C(Base(c) + t1) ;
          j := j + 1
        );
      i := i + 1 );
```

Loop invariant expression

# Transformation: Strength Reduction

```
i  := 0;
t3 := 0;
while i <= m do
    ( j := 0;  t2 := t3;
      while j <= n do
        ( t1   := t2 + j;
          ta   :=   Base(a) + t1;
          C(ta):= C(Base(b) + t1) +
                  C(Base(c) + t1) ;
          j := j + 1
        );
      t3 := t3 + (n+1);
      i := i + 1 );
```

# Copy Analysis

```
i  := 0;
t3 := 0;
while i <= m do
     ( j := 0;  t2 := t3;
       while j <= n do
         ( t1   := t2 + j;
           ta   :=   Base(a) + t1;
           C(ta):= C(Base(b) + t1) +
                   C(Base(c) + t1) ;
           j := j + 1
         );
       t3 := t3 + (n+1);
       i := i + 1 );
```

# Copy Propagation

```
i  := 0;
t3 := 0;
while i <= m do
    ( j := 0;  t2 := t3;
      while j <= n do
        ( t1   := t3 + j;
          ta   :=   Base(a) + t1;
          C(ta):= C(Base(b) + t1) +
                  C(Base(c) + t1) ;
          j := j + 1
        );
      t3 := t3 + (n+1);
      i := i + 1 );
```

# Analysis: Dead Variable

```
i  := 0;
t3 := 0;
while i <= m do
    ( j := 0;  t2 := t3;
      while j <= n do
        ( t1    := t3 + j;
          ta    :=   Base(a) + t1;
          C(ta):= C(Base(b) + t1) +
                  C(Base(c) + t1) ;
          j := j + 1
        );
      t3 := t3 + (n+1);
      i := i + 1 );
```

Useless assignment (points to `t2 := t3;`)

# Transformation: Elimination of Useless Assignments

```
i  := 0;
t3 := 0;
while i <= m do
    ( j := 0;
      while j <= n do
        ( t1   := t3 + j;
          ta   :=   Base(a) + t1;
          C(ta):= C(Base(b) + t1) +
                  C(Base(c) + t1) ;
             j := j + 1
        );
      t3 := t3 + (n+1);
      i := i + 1 );
```

# Elimination of Redundant Computations

1. Introduce a temporary variable
2. Initialize the variable before the first computation of an expression
3. Replace all uses of the expression with uses of the variable

```
  ta  :=   B(a)+i*(n+1)+j;
C(ta) := C(B(b)+i*(n+1)+j) +
         C(B(c)+i*(n+1)+j) ;
```

⇒

```
  t1 := i*(n+1)+j;
  ta :=   B(a)+t1;
C(ta):= C(B(b)+t1) +
        C(B(c)+t1) ;
```

# Elimination of Redundant Computations

- Requirements:
  - None of expression variables change their values between the first and the last computation
  - 2 uses at least

```
  ta  :=   B(a)+i*(n+1)+j;
C(ta) := C(B(b)+i*(n+1)+j) +
         C(B(c)+i*(n+1)+j) ;
```

⇒

```
 t1 := i*(n+1)+j;
 ta :=   B(a)+t1;
C(ta):= C(B(b)+t1) +
        C(B(c)+t1) ;
```

# Loop Invariant Code Motion

- Loop invariant expressions pulled out of the loop
- Requirement: None of expression variables change their values within the loop

```
for (i=0;i<10;i++) {
  z = r*r;
  k = k*i;
}
```

```
z = r*r;
for (i=0;i<10;i++) {

   k = k*i;
}
```

# Strength Reduction

- Simplifying arithmetic operations within a loop: Replacing monolithic operations with iterative operations

```
i := 0;

while i <= m do
      t2 := i*(n+1);
      ...
      i  := i+1;

end
```

```
i := 0;
t3 := 0;
while i <= m do
      t2 := t3;
      ...
      i  := i+1;
      t3 := t3+(n+1);
end
```

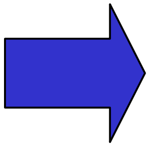
# Strength Reduction

- Requirements:
  - The expression is a multiplication or a power
  - The expression is a function of the form
    **e = n*x + k** or
    **e = n^x** where **x** is the loop count
  - **n** and **k** are invariant within the loop

```
i := 0;

while i <= m do
      t2 := i*(n+1);
      ...
      i  := i+1;

end
```



```
i := 0;
t3 := 0;
while i <= m do
      t2 := t3;
      ...
      i  := i+1;
      t3 := t3+(n+1);
end
```

# Elimination of Redundant Variables

- Replace uses of variable **x** with uses of variable **y** if **x** is a copy of **y**
- Requirement: Wherever **y** is used, **x** has the same value as **y**

```
t2 := t3;
...
t1 := t2 + j ;
```

```
t2 := t3;
...
t1 := t3 + j;
```

# Useless Assignment Elimination

- Remove assignments to variables whose value is not used

```
x := 3;
a := z;
x := a;
```

```
a := z;
x := a;
```

# Remarks

- No Pointers involved
- No function calls inside the loop
- No aliases between variables

- Ordering between optimizations has to be found

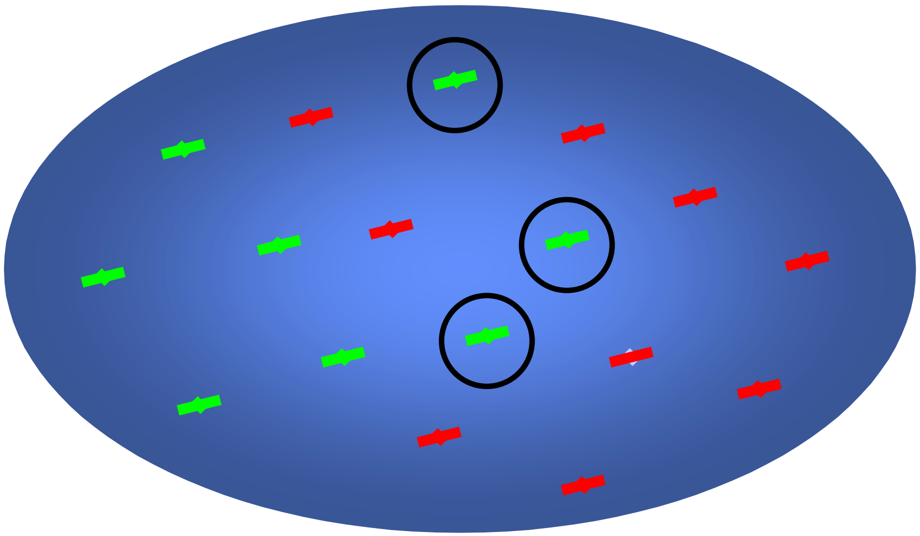
- Typical optimizations for „optimizing“ compiler

# Overview

# The Halting Problem

- Interesting problems are not computable
  - ➔ Need of approximative solutions
- Only erring on the safe side allowed
- The safe side depends on interpretation of collected information
- Less erring means more precision

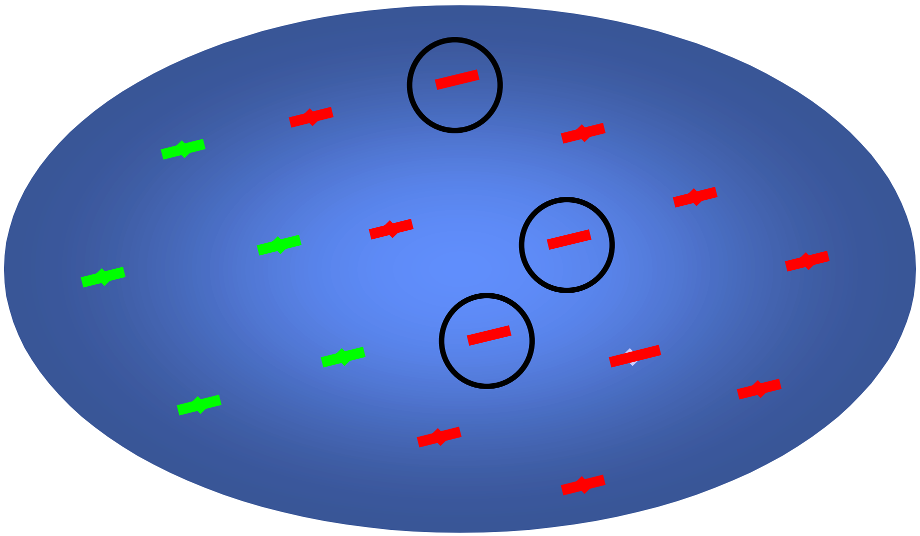
# Exact Answers



Property
holds

Property
does not hold

# Approximations



Property
*definitely* holds

→ Erring on the safe side

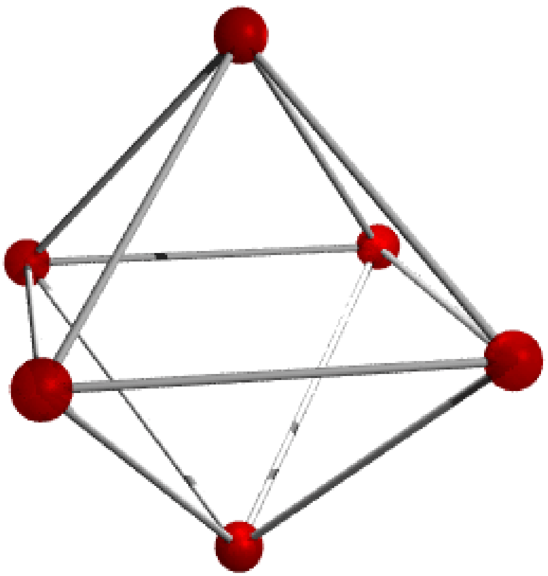
Property
*might not* hold

# Overview

# Abstract Interpretation

- Introduced 1977 by
  Patric and Radhia Cousot

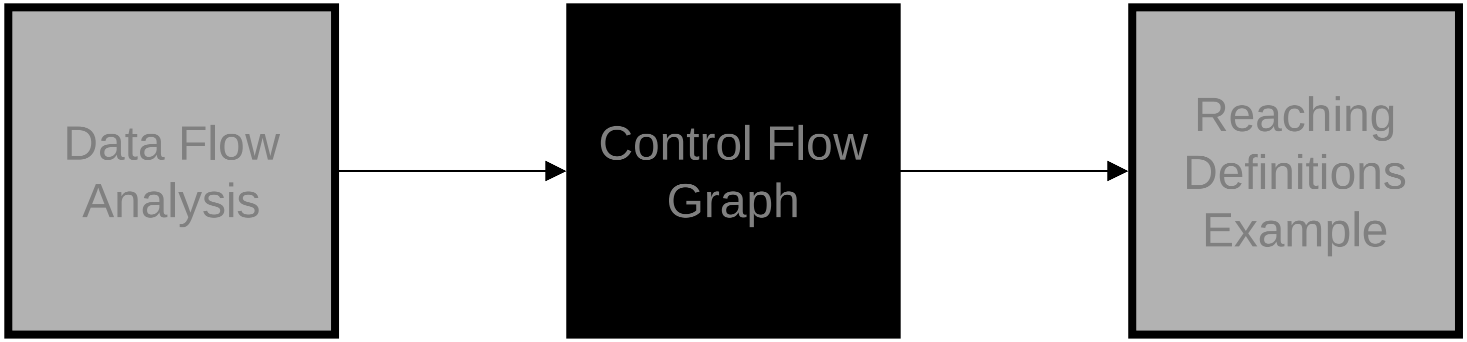
- Provides for
  - Analysis design
  - Correctness proves



# Data Flow Analysis

- Introduced ~1973
- Goal:
  Proving properties of a program
  for each program point
- Works on control flow graphs
- Correctness can be proved
  by abstract interpretation

# Overview


Data Flow Analysis
Control Flow Graph
Reaching Definitions Example

# Control Flow Graph

- Representation of program structure
- Nodes:
  Statements or statement parts
- Edges:
  Possible flow of control
- Edges can be labeled

# Example: Control Flow Graph

[y := x]$^1$;

[z := 1]$^2$;

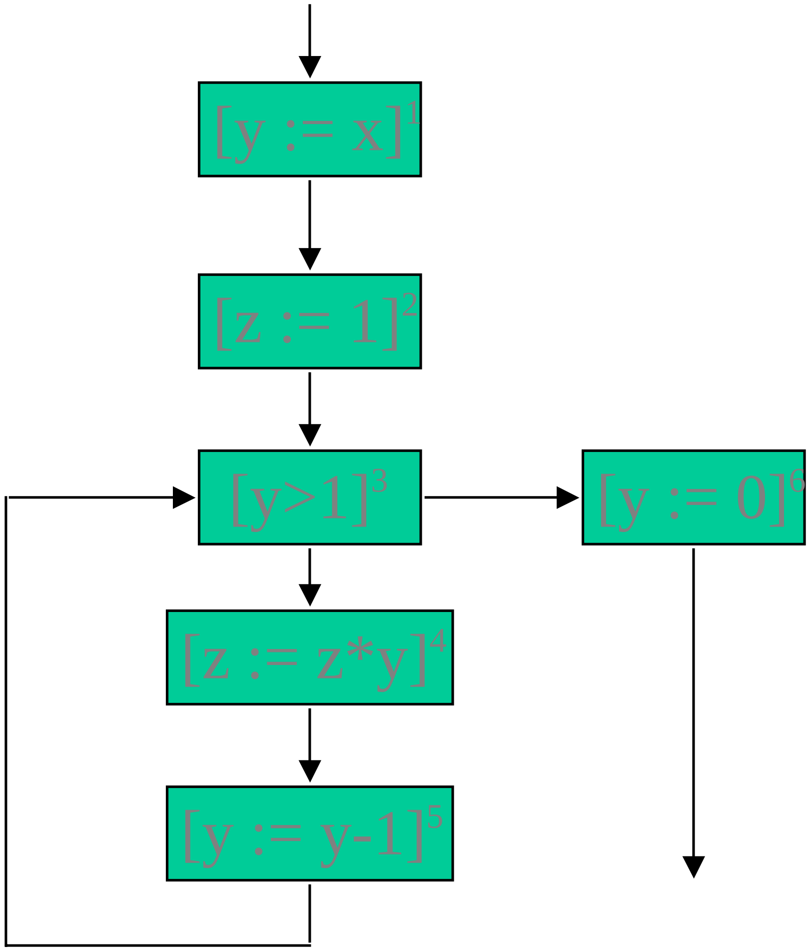
<u>while</u> [y>1]$^3$

<u>do</u> ( [z := z*y]$^4$;

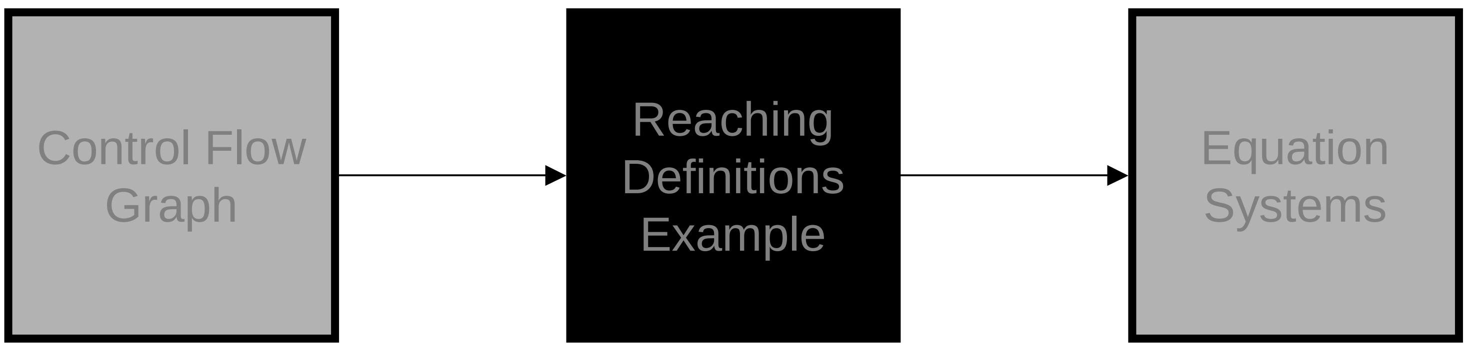
[y := y-1]$^5$

);

[y := 0]$^6$



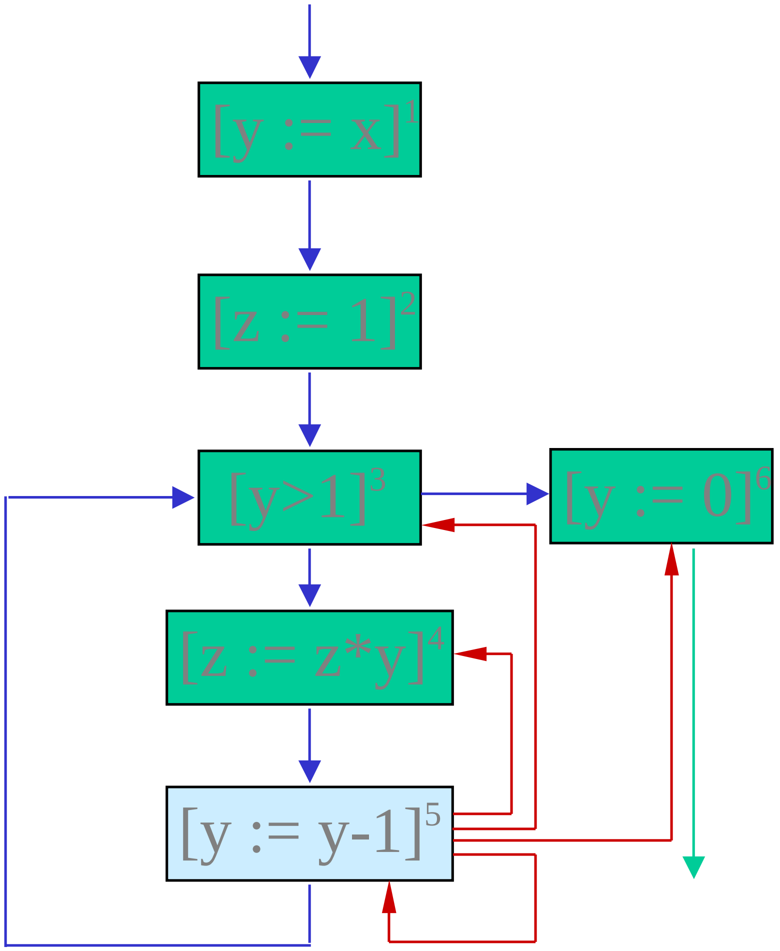
# Overview



Control Flow Graph
Reaching Definitions Example
Equation Systems

# Reaching Definitions

- An assignment $[x{:=}a]^k$ reaches $k'$ if there is a path to $k'$ on which the last assignment of $x$ was in $k$



# Reaching Definitions

(x,?), (y,?), (z,?)

```
[y := x]1;
[z := 1]2;
while [y>1]3
do  ( [z := z*y]4;
      [y := y-1]5
    );
[y := 0]6
```

# Reaching Definitions

(x,?), ~~(y,?)~~, (z,?)

(x,?), (y,1), (z,?)

[y := x]$^1$;

[z := 1]$^2$;

while [y>1]$^3$

do ( [z := z*y]$^4$;

[y := y-1]$^5$

);

[y := 0]$^6$

# Reaching Definitions

```
[y := x]1;
[z := 1]2;
while [y>1]3
do  (  [z := z*y]4;
       [y := y-1]5
    );
[y := 0]6
```

(x,?), (y,?), (z,?)

(x,?), (y,1), ~~(z,?)~~

(x,?), (y,1), (z,2)

# Reaching Definitions

$[y := x]^1$; ← (x,?), (y,?), (z,?)

← (x,?), (y,1), (z,?)

$[z := 1]^2$;

← (x,?), (y,1), (z,2)

while $[y>1]^3$ ← (x,?), (y,1), (z,2)

do ( $[z := z*y]^4$;

$[y := y-1]^5$

);

← (x,?), (y,1), (z,2)

$[y := 0]^6$

# Reaching Definitions

(x,?), (y,?), (z,?)

[y := x]$^1$;

(x,?), (y,1), (z,?)

[z := 1]$^2$;

(x,?), (y,1), (z,2)

<u>while</u> [y>1]$^3$

(x,?), (y,1), ~~(z,2)~~

<u>do</u> ( [z := z*y]$^4$;

(x,?), (y,1), (z,4)

[y := y-1]$^5$

);

(x,?), (y,1), (z,2)

[y := 0]$^6$

# Reaching Definitions

| Program | Reaching definitions |
| --- | --- |
| | (x,?), (y,?), (z,?) |
| $[y := x]^1$; | |
| | (x,?), (y,1), (z,?) |
| $[z := 1]^2$; | |
| | (x,?), (y,1), (z,2) |
| **while** $[y>1]^3$ | |
| | (x,?), (y,1), (z,2) |
| **do** ( $[z := z*y]^4$; | |
| | (x,?), ~~(y,1)~~, (z,4) |
| $[y := y-1]^5$ | |
| | (x,?), (y,5), (z,4) |
| ); | |
| | (x,?), (y,1), (z,2) |
| $[y := 0]^6$ | |

# Reaching Definitions

(x,?), (y,?), (z,?)

$[y := x]^1$;

(x,?), (y,1), (z,?)

$[z := 1]^2$;

(x,?), (y,1), (z,2), (y,5), (z,4)

**while** $[y>1]^3$

(x,?), (y,1), (z,2)

**do** ( $[z := z*y]^4$;

(x,?), (y,1), (z,4)

$[y := y-1]^5$

(x,?), (y,5), (z,4)

);

(x,?), (y,1), (z,2)

$[y := 0]^6$

# Reaching Definitions

```
                              <- (x,?), (y,?), (z,?)
[y := x]^1;
                              <- (x,?), (y,1), (z,?)
[z := 1]^2;
                              <- (x,?), (y,1), (z,2), (y,5), (z,4)
while [y>1]^3
                              <- (x,?), (y,1), (z,2), (y,5), (z,4)
do  (  [z := z*y]^4;
                              <- (x,?), (y,1), (z,4)
       [y := y-1]^5
                              <- (x,?), (y,5), (z,4)
    );
                              <- (x,?), (y,1), (z,2), (y,5), (z,4)
[y := 0]^6
```

# Reaching Definitions

(x,?), (y,?), (z,?)

$[y := x]^1$;

(x,?), (y,1), (z,?)

$[z := 1]^2$;

(x,?), (y,1), (z,2), (y,5), (z,4)

<u>while</u> $[y>1]^3$

(x,?), (y,1), ~~(z,2)~~, (y,5), ~~(z,4)~~

<u>do</u> ( $[z := z*y]^4$;

(x,?), (y,1), (z,4), (y,5)

$[y := y-1]^5$

(x,?), (y,5), (z,4)

);

(x,?), (y,1), (z,2), (y,5), (z,4)

$[y := 0]^6$

# Reaching Definitions

[y := x][1];

[z := 1][2];

<u>while</u> [y>1][3]

<u>do</u> ( [z := z*y][4];

[y := y-1][5]

);

[y := 0][6]

(x,?), (y,?), (z,?)

(x,?), (y,1), (z,?)

(x,?), (y,1), (z,2), (y,5), (z,4)

(x,?), (y,1), (z,2), (y,5), (z,4)

(x,?), ~~(y,1)~~, (z,4), ~~(y,5)~~

(x,?), (y,5), (z,4)

(x,?), (y,1), (z,2), (y,5), (z,4)

# Reaching Definitions

(x,?), (y,?), (z,?)

[y := x]$^1$;

(x,?), (y,1), (z,?)

[z := 1]$^2$;

(x,?), (y,1), (z,2), (y,5), (z,4)

<u>while</u> [y>1]$^3$

(x,?), (y,1), (z,2), (y,5), (z,4)

<u>do</u> ( [z := z*y]$^4$;

(x,?), (y,1), (z,4), (y,5)

[y := y-1]$^5$

(x,?), (y,5), (z,4)

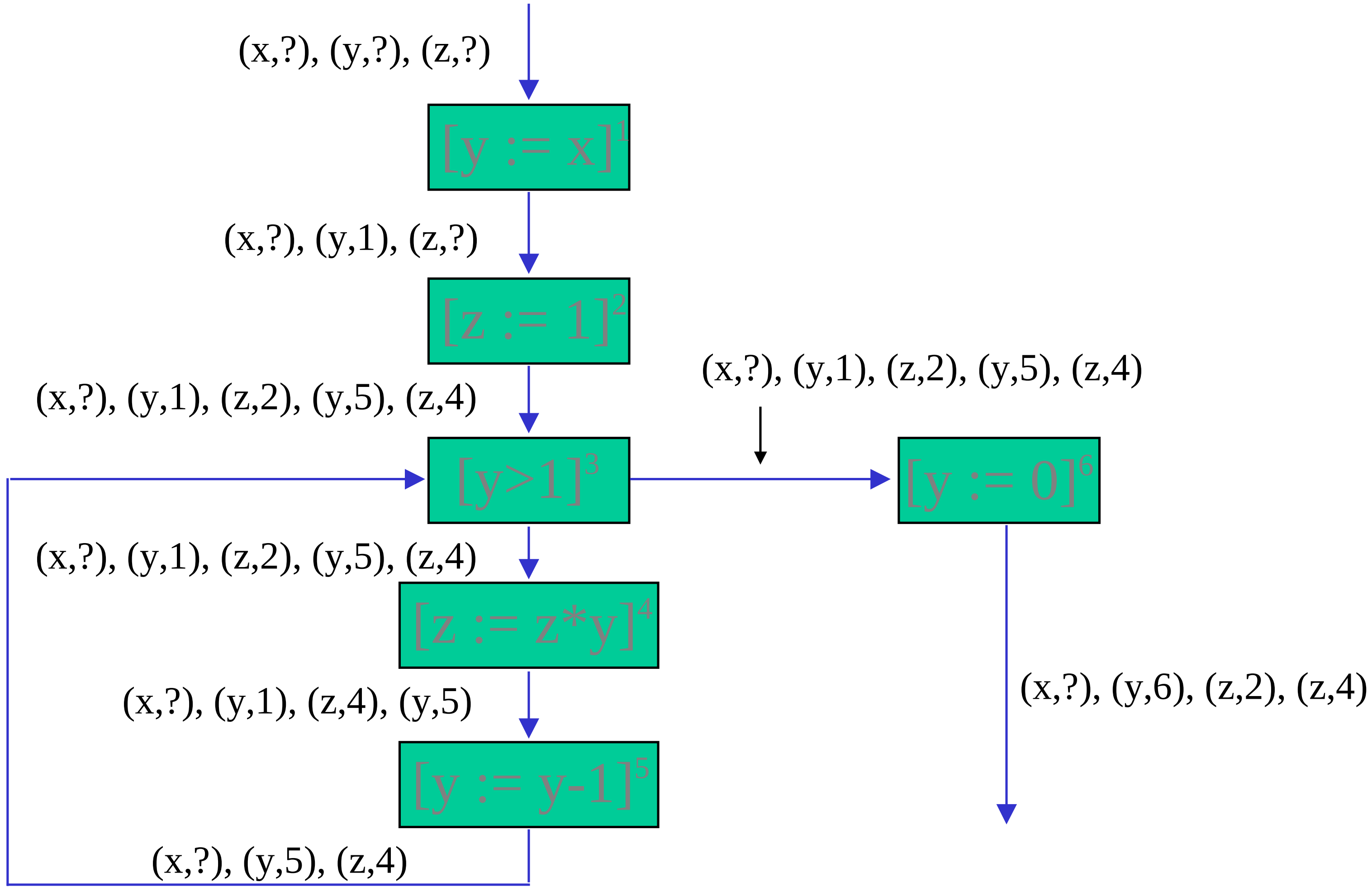
);

(x,?), ~~(y,1)~~, (z,2), ~~(y,5)~~, (z,4)

[y := 0]$^6$

(x,?), (y,6), (z,2), (z,4)

# Reaching Definitions



# Reaching Definitions

Another safe solution - but not the best:

| Program | Reaching definitions |
| --- | --- |
| | (x,?), (y,?), (z,?) |
| [y := x]¹; | |
| | (x,?), (y,1), (z,?) |
| [z := 1]²; | |
| | (x,?), (y,1), (z,2), (y,5), (z,4) |
| while [y>1]³ | (x,?), (y,1), (z,2), (y,5), (z,4) |
| do ( [z := z*y]⁴; | |
| | (x,?), (y,1), (z,2), (y,5), (z,4) |
| [y := y-1]⁵ | |
| | (x,?), (y,1), (z,2), (y,5), (z,4) |
| ); | |
| | (x,?), (y,1), (z,2), (y,5), (z,4) |
| [y := 0]⁶ | |
| | (x,?), (y,6), (z,2), (z,4) |

# Reaching Definitions

Unsafe solution:

(x,?), (y,?), (z,?)

[y := x][1];

(x,?), (y,1), (z,?)

[z := 1][2];

(x,?), (y,1), (z,2), (y,5), (z,4)

<u>while</u> [y>1][3]

(x,?), ~~(y,1)~~, ~~(z,2)~~, (y,5), (z,4)

<u>do</u> ( [z := z*y][4];

(x,?), ~~(y,1)~~, (z,4), (y,5)

[y := y-1][5]

(x,?), (y,5), (z,4)

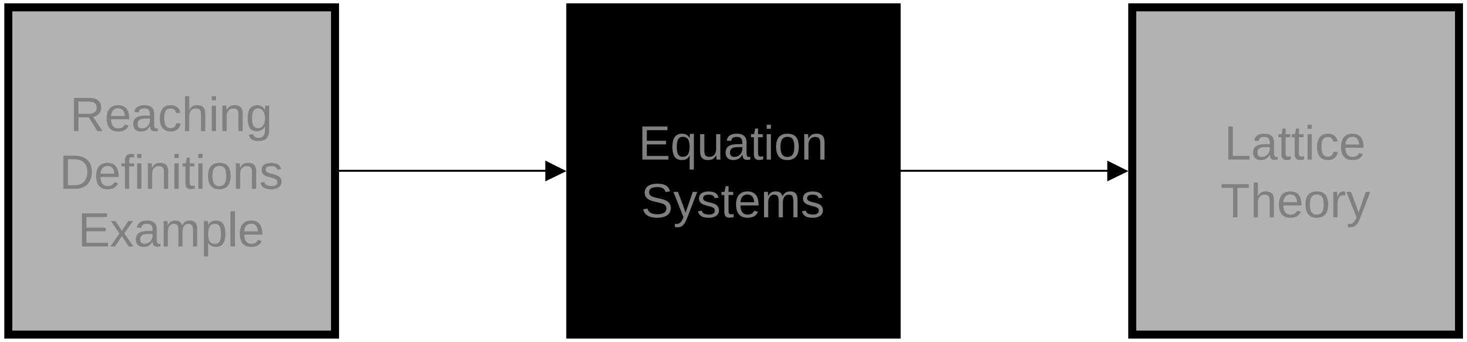
);

(x,?), ~~(y,1)~~, ~~(z,2)~~, (y,5), (z,4)

[y := 0][6]

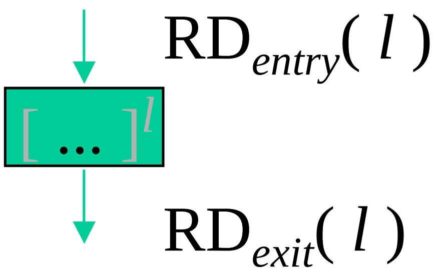
(x,?), (y,6), (z,2), (z,4)

# Overview


Reaching Definitions Example
Equation Systems
Lattice Theory

# Computing
# Data Flow Information

- Extracting an equation system
- Computing the least fixpoint




- remove **killed** variables
- add **generated** variables

$[y := x]^1$;

$[z := 1]^2$;

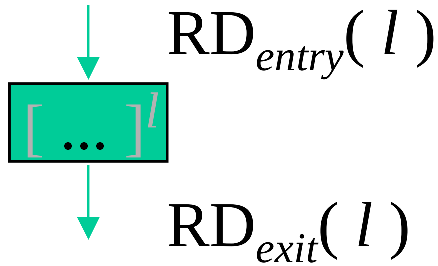
<u>while</u> $[y>1]^3$

<u>do</u> ( $[z := z*y]^4$;

$[y := y-1]^5$

);

$[y := 0]^6$




- remove **killed** variables
- add **generated** variables

[y := x]^1;      ←      $RD_{exit}(1) = (RD_{entry}(1) \setminus \{ (y,l) \mid l \in \mathbf{Lab} \}) \cup \{ (y,1) \}$

[z := 1]^2;

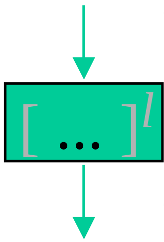
<u>while</u> [y>1]^3

<u>do</u> ( [z := z*y]^4;

[y := y-1]^5

);

[y := 0]^6




$RD_{entry}(\,l\,)$

$RD_{exit}(\,l\,)$

- remove killed variables
- add generated variables

---

$[y := x]^1$; ← $RD_{exit}(1) = (RD_{entry}(1) \setminus \{\ (y,l) \mid l \in \mathbf{Lab}\}\ ) \cup \{\ (y,1)\}$

$[z := 1]^2$; ← $RD_{exit}(2) = (RD_{entry}(2) \setminus \{\ (z,l) \mid l \in \mathbf{Lab}\}\ ) \cup \{\ (z,2)\}$

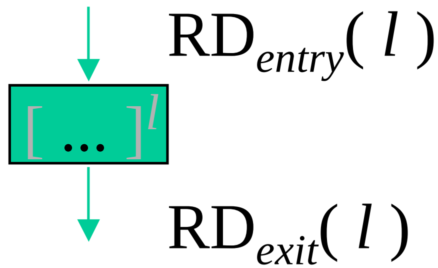
<u>while</u> $[y>1]^3$

<u>do</u> ( $[z := z*y]^4$;

$[y := y-1]^5$

);

$[y := 0]^6$




- remove **killed** variables
- add **generated** variables

---

$[y := x]^1$; ← $RD_{exit}(1) = (RD_{entry}(1) \setminus \{\, (y,l) \mid l \in \mathbf{Lab} \}\,) \cup \{\, (y,1)\}$

$[z := 1]^2$; ← $RD_{exit}(2) = (RD_{entry}(2) \setminus \{\, (z,l) \mid l \in \mathbf{Lab} \}\,) \cup \{\, (z,2)\}$

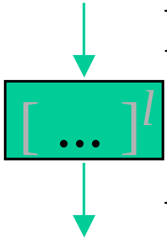
<u>while</u> $[y>1]^3$ ← $RD_{exit}(3) = RD_{entry}(3)$

<u>do</u> ( $[z := z*y]^4$;

$[y := y-1]^5$

);

$[y := 0]^6$




$RD_{entry}(\,l\,)$

$RD_{exit}(\,l\,)$

- remove **killed** variables
- add **generated** variables

---

$[y := x]^1$; $RD_{exit}(1) = (RD_{entry}(1) \setminus \{ (y,l) \mid l \in \mathbf{Lab} \}) \cup \{ (y,1) \}$

$[z := 1]^2$; $RD_{exit}(2) = (RD_{entry}(2) \setminus \{ (z,l) \mid l \in \mathbf{Lab} \}) \cup \{ (z,2) \}$

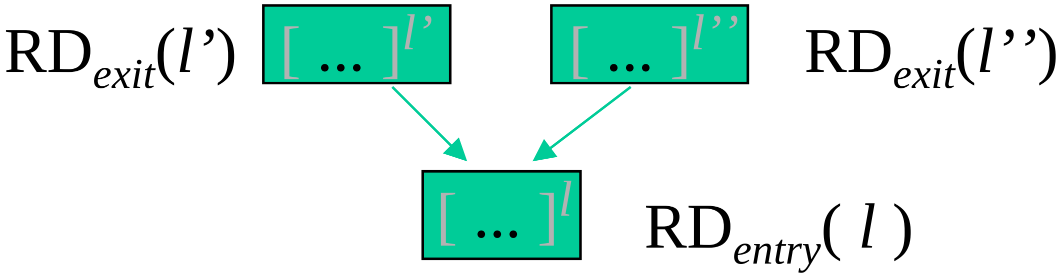
<u>while</u> $[y>1]^3$ $RD_{exit}(3) = RD_{entry}(3)$

<u>do</u> ( $[z := z*y]^4$; $RD_{exit}(4) = (RD_{entry}(4) \setminus \{ (z,l) \mid l \in \mathbf{Lab} \}) \cup \{ (z,4) \}$

$[y := y-1]^5$ $RD_{exit}(5) = (RD_{entry}(5) \setminus \{ (y,l) \mid l \in \mathbf{Lab} \}) \cup \{ (y,5) \}$

);

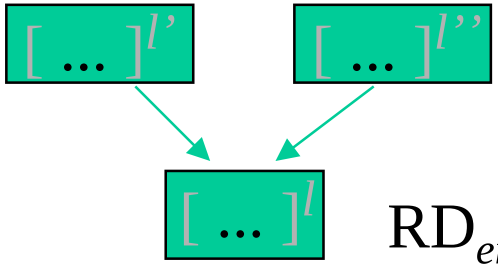
$[y := 0]^6$ $RD_{exit}(6) = (RD_{entry}(6) \setminus \{ (y,l) \mid l \in \mathbf{Lab} \}) \cup \{ (y,6) \}$




- **join** information from where control could come from

```
[y := x]^1;
[z := 1]^2;
while [y>1]^3
do  ( [z := z*y]^4;
      [y := y-1]^5
    );
[y := 0]^6
```

$RD_{exit}(l')$ $RD_{exit}(l'')$




$RD_{entry}(\, l \,)$

- join information from where control could come from

---

$[y := x]^1$;

$[z := 1]^2$; ← $RD_{entry}(2) = RD_{exit}(1)$

<u>while</u> $[y>1]^3$

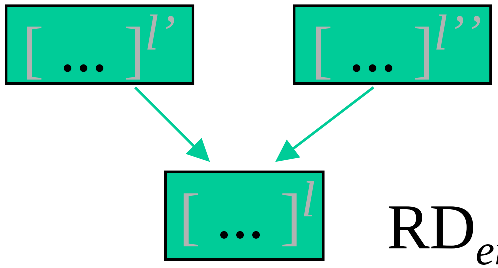
<u>do</u> ( $[z := z*y]^4$;

$[y := y-1]^5$

);

$[y := 0]^6$

$RD_{exit}(l')$




$RD_{exit}(l'')$

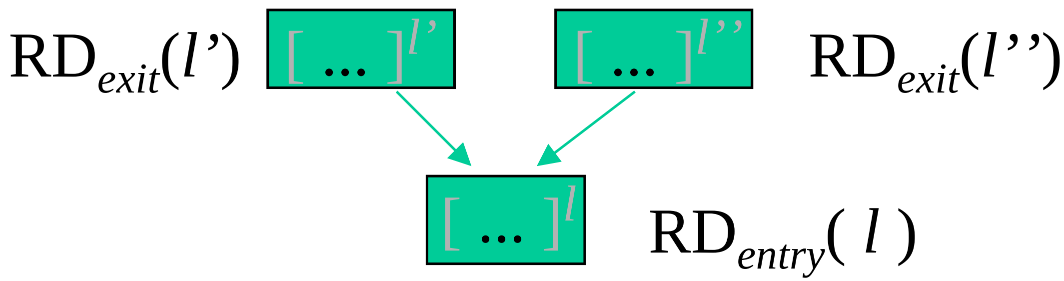
$RD_{entry}(l)$

- join information from where control could come from

```
[y := x]^1;
[z := 1]^2;
while [y>1]^3
do  ( [z := z*y]^4;
        [y := y-1]^5
    );
[y := 0]^6
```

$RD_{entry}(2) = RD_{exit}(1)$

$RD_{entry}(3) = RD_{exit}(2) \cup RD_{exit}(5)$




- join information from where control could come from

[y := x]$^{1}$;

[z := 1]$^{2}$; ← $RD_{entry}(2) = RD_{exit}(1)$

<u>while</u> [y>1]$^{3}$ ← $RD_{entry}(3) = RD_{exit}(2) \cup RD_{exit}(5)$

<u>do</u> ( [z := z*y]$^{4}$; ← $RD_{entry}(4) = RD_{exit}(3)$

[y := y-1]$^{5}$ ← $RD_{entry}(5) = RD_{exit}(4)$

);

[y := 0]$^{6}$ ← $RD_{entry}(6) = RD_{exit}(3)$

$[\ \ldots\ ]^{l}$ $\qquad RD_{entry}(\,l\,)$

- initially all variables are uninitialized

---

| | |
|---|---|
| [y := x]$^{1}$; | $RD_{entry}(1) = \{\ (x,?),\ (y,?),\ (z,?)\ \}$ |
| [z := 1]$^{2}$; | $RD_{entry}(2) = RD_{exit}(1)$ |
| <u>while</u> [y>1]$^{3}$ | $RD_{entry}(3) = RD_{exit}(2) \cup RD_{exit}(5)$ |
| <u>do</u> ( [z := z*y]$^{4}$; | $RD_{entry}(4) = RD_{exit}(3)$ |
| [y := y-1]$^{5}$ | $RD_{entry}(5) = RD_{exit}(4)$ |
| ); | |
| [y := 0]$^{6}$ | $RD_{entry}(6) = RD_{exit}(3)$ |

# Reaching Definitions: Summary

- 12 sets:
  - $RD_{entry}(1), \ldots, RD_{exit}(6)$
- 12 equations:
  - $RD_j = F_j\ (RD_{entry}(1), \ldots, RD_{exit}(6))$
- One function:
  - $F: (\wp(Var_* \times Lab_*))^{12} \Rightarrow (\wp(Var_* \times Lab_*))^{12}$
- We want the least fixed point of $F$: but does it exist?