Principles of Program Analysis
Data Flow Analysis

Rodrigo Bonifácio

February 15, 2022

(Section 1.1) Program Analysis

Goal

Predict safe and computable approximations of the possible behaviours that arise at runtime when executing a program.

(Section 1.1) Program Analysis

Goal

Predict safe and computable approximations of the possible behaviours that arise at runtime when executing a program. Usage Scenarios:

- program optimization
- program transformation
- program design metrics
- program security

(Section 1.2) The While Language

Features

- tiny (imperative) programming language
- integer and boolean expressions
- conditionals and loops

(Section 1.2) The While Language

Features

- tiny (imperative) programming language
- integer and boolean expressions
- conditionals and loops
- every statement has a unique label

- running example: reaching definitions.

(Section 1.4) An introduction to DataFlow Analysis

1. The program is modeled as a (control-flow) graph: the nodes are the the elementary blocks (statements) and the edges describe how the control might pass from one statement to another.
2. We define functions to every node, so that data flow information can be computed using pairs of *Gen* and *Kill* abstractions—considering the information produced in the previou(s) node(s). What should we do at merge nodes?

(Section 1.4) An introduction to DataFlow Analysis

1. The program is modeled as a (control-flow) graph: the nodes are the the elementary blocks (statements) and the edges describe how the control might pass from one statement to another.
2. We define functions to every node, so that data flow information can be computed using pairs of *Gen* and *Kill* abstractions—considering the information produced in the previou(s) node(s). What should we do at merge nodes?

Computing a solution for a dataflow problem often requires multiple iteractions, where every new iteration provides a better approximation (monotone framework). The iteration continues until achieving a fixpoint.

(Section 1.4) An introduction to DataFlow Analysis

1. The program is modeled as a (control-flow) graph: the nodes are the the elementary blocks (statements) and the edges describe how the control might pass from one statement to another.
2. We define functions to every node, so that data flow information can be computed using pairs of *Gen* and *Kill* abstractions—considering the information produced in the previou(s) node(s). What should we do at merge nodes?

Computing a solution for a dataflow problem often requires multiple iteractions, where every new iteration provides a better approximation (monotone framework). The iteration continues until achieving a fixpoint. $f(s) = s$

Running Example

Example 01: Factorial

While language

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

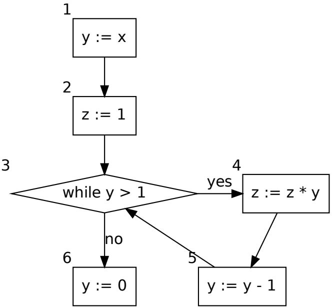
Running Example

Example 01: Factorial

While language

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

Control Flow Graph



Reaching Definitions

Reaching Definitions

- Application in different areas, including tainted analysis.
- Allows the construction of *def-use* and *use-def* chains. These chains facilitate several optmizing transformations.

Reaching Definitions Algorithm

Informal definition

For every vertice `(from, to)` in the control flow, we check if `assignment(v, exp) := to` holds (`to` is an assignment).

Reaching Definitions Algorithm

Informal definition

For every vertice `(from, to)` in the control flow, we check if `assignment(v, exp) := to` holds (`to` is an assignment).If this is the case, we remove all sets of definitions that assign a value to `v` (Kill) from the abstraction, and expose a new definition of `v` at statement `to` (Gen). If `to` is not a definition (an assignment), than we just propagate the sets of definitions that arrive at `to`.

Reaching Definitions Algorithm

Equations

- $Out(s) = Gen(s) \cup (In(s) - Kill(s))$
- $In(s) = \bigcup_p Out(p), p \in pred(s), s \in stmts$

Reaching Definitions Algorithm

Equations

- $Out(s) = Gen(s) \cup (In(s) - Kill(s))$
- $In(s) = \bigcup_p Out(p), p \in pred(s), s \in stmts$

Iterative algorithm

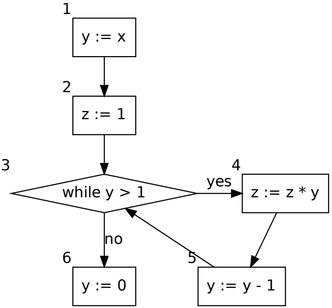
```
// initialization
Out(Start) = {};
for(s in stmts) do Out(s) = {}

while(changes to any out occurs) do
//do until achieving a fixed point
  for(s in stmts)
    // compute In(s)
    // compute Out(s)
```

Iteration 1

Equations

- $Out(s) = Gen(s) \cup (In(s) - Kill(s))$
- $In(s) = \bigcup_p Out(p), p \in pred(s), s \in stmts$

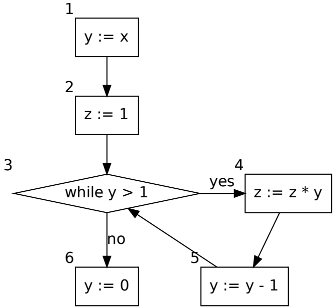


n	IN[n]	OUT[n]
1	{ }	{(y,1)}

Iteration 1

Equations

- $Out(s) = Gen(s) \cup (In(s) - Kill(s))$
- $In(s) = \bigcup_p Out(p), p \in pred(s), s \in stmts$

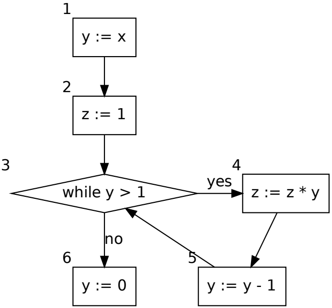


n	IN[n]	OUT[n]
1	{ }	{(y,1)}
2	{(y,1)}	{(y,1), (z,2)}

Iteration 1

Equations

- $Out(s) = Gen(s) \cup (In(s) - Kill(s))$
- $In(s) = \bigcup_p Out(p), p \in pred(s), s \in stmts$

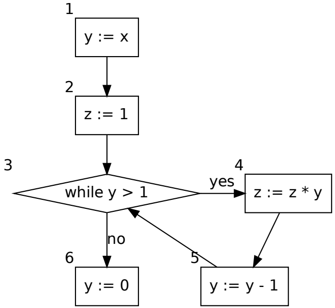


n	IN[n]	OUT[n]
1	{ }	{(y,1)}
2	{(y,1)}	{(y,1), (z,2)}
3	{(y,1), (z,2)}	{(y,1), (z,2)}

Iteration 1

Equations

- $Out(s) = Gen(s) \cup (In(s) - Kill(s))$
- $In(s) = \bigcup_p Out(p), p \in pred(s), s \in stmts$

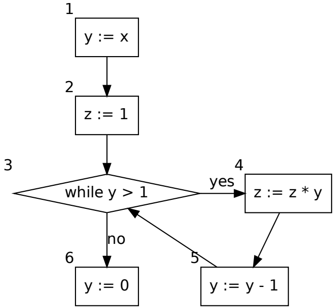


n	IN[n]	OUT[n]
1	{ }	{(y,1)}
2	{(y,1)}	{(y,1), (z,2)}
3	{(y,1), (z,2)}	{(y,1), (z,2)}
4	{(y,1), (z,2)}	{(y,1), (z,4)}

Iteration 1

Equations

- $Out(s) = Gen(s) \cup (In(s) - Kill(s))$
- $In(s) = \bigcup_p Out(p), p \in pred(s), s \in stmts$

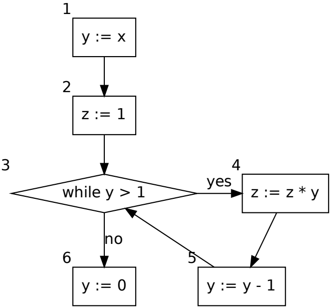


n	IN[n]	OUT[n]
1	{ }	{(y,1)}
2	{(y,1)}	{(y,1), (z,2)}
3	{(y,1), (z,2)}	{(y,1), (z,2)}
4	{(y,1), (z,2)}	{(y,1), (z,4)}
5	{(y,1), (z,4)}	{(y,5), (z,4)}

Iteration 1

Equations

- $Out(s) = Gen(s) \cup (In(s) - Kill(s))$
- $In(s) = \bigcup_p Out(p), p \in pred(s), s \in stmts$

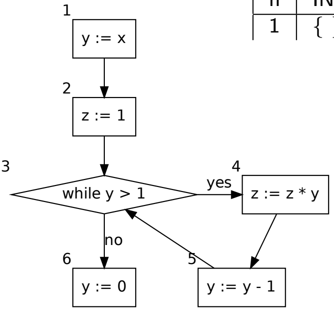


n	IN[n]	OUT[n]
1	{ }	{(y,1)}
2	{(y,1)}	{(y,1), (z,2)}
3	{(y,1), (z,2)}	{(y,1), (z,2)}
4	{(y,1), (z,2)}	{(y,1), (z,4)}
5	{(y,1), (z,4)}	{(y,5), (z,4)}
6	{(y,1), (z,2)}	{(y,6), (z,2)}

Iteration 2

Equations

- $Out(s) = Gen(s) \cup (In(s) - Kill(s))$
- $In(s) = \bigcup_p Out(p), p \in pred(s), s \in stmts$

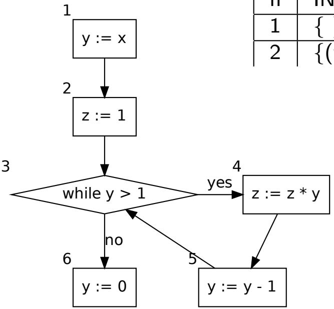


n	IN[n]	OUT[n]
1	{ }	{(y,1)}

Iteration 2

Equations

- $Out(s) = Gen(s) \cup (In(s) - Kill(s))$
- $In(s) = \bigcup_p Out(p), p \in pred(s), s \in stmts$

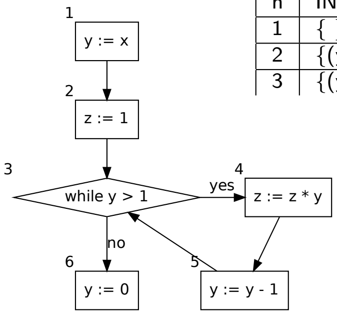


n	IN[n]	OUT[n]
1	{ }	{(y,1)}
2	{(y,1)}	{(y,1), (z,2)}

Iteration 2

Equations

- $Out(s) = Gen(s) \cup (In(s) - Kill(s))$
- $In(s) = \bigcup_p Out(p), p \in pred(s), s \in stmts$

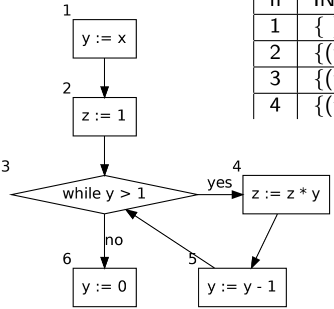


n	IN[n]	OUT[n]
1	{ }	{(y,1)}
2	{(y,1)}	{(y,1), (z,2)}
3	{(y,1), (z,2), (y,5), (z,4)}	{(y,1), (z,2), (y,5), (z,4)}

Iteration 2

Equations

- $Out(s) = Gen(s) \cup (In(s) - Kill(s))$
- $In(s) = \bigcup_p Out(p), p \in pred(s), s \in stmts$

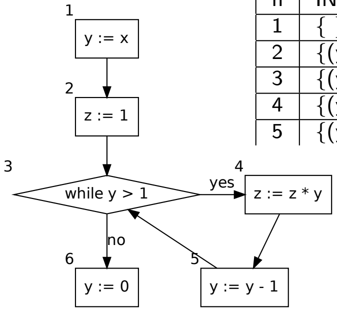


n	IN[n]	OUT[n]
1	{ }	{(y,1)}
2	{(y,1)}	{(y,1), (z,2)}
3	{(y,1), (z,2), (y,5), (z,4)}	{(y,1), (z,2), (y,5), (z,4)}
4	{(y,1), (z,2), (y,5), (z,4)}	{(y,1), (y,5), (z,4)}

Iteration 2

Equations

- $Out(s) = Gen(s) \cup (In(s) - Kill(s))$
- $In(s) = \bigcup_p Out(p), p \in pred(s), s \in stmts$

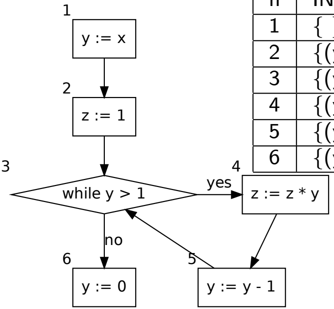


n	IN[n]	OUT[n]
1	{ }	{(y,1)}
2	{(y,1)}	{(y,1), (z,2)}
3	{(y,1), (z,2), (y,5), (z,4)}	{(y,1), (z,2), (y,5), (z,4)}
4	{(y,1), (z,2), (y,5), (z,4)}	{(y,1), (y,5), (z,4)}
5	{(y,1), (y,5), (z,4)}	{(y,5), (z,4)}

Iteration 2

Equations

- $Out(s) = Gen(s) \cup (In(s) - Kill(s))$
- $In(s) = \bigcup_p Out(p), p \in pred(s), s \in stmts$

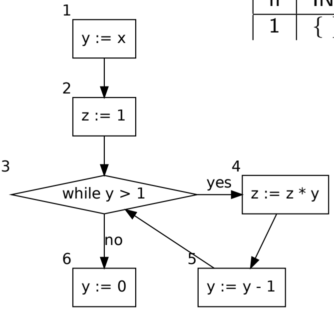


n	IN[n]	OUT[n]
1	{ }	{(y,1)}
2	{(y,1)}	{(y,1), (z,2)}
3	{(y,1), (z,2), (y,5), (z,4)}	{(y,1), (z,2), (y,5), (z,4)}
4	{(y,1), (z,2), (y,5), (z,4)}	{(y,1), (y,5), (z,4)}
5	{(y,1), (y,5), (z,4)}	{(y,5), (z,4)}
6	{(y,1), (z,2), (y,5), (z,4)}	{(y,6), (z,2), (z,4)}

Iteration 3 (fixed point)

Equations

- $Out(s) = Gen(s) \cup (In(s) - Kill(s))$
- $In(s) = \bigcup_p Out(p), p \in pred(s), s \in stmts$

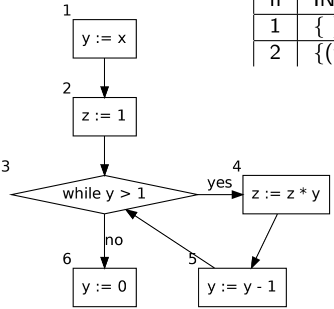


n	IN[n]	OUT[n]
1	{ }	{(y,1)}

Iteration 3 (fixed point)

Equations

- $Out(s) = Gen(s) \cup (In(s) - Kill(s))$
- $In(s) = \bigcup_p Out(p), p \in pred(s), s \in stmts$

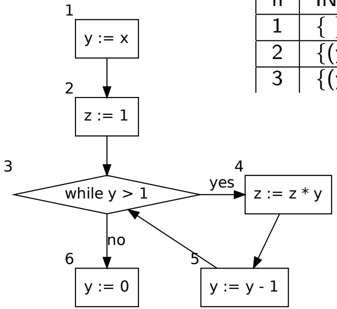


n	IN[n]	OUT[n]
1	{ }	{(y,1)}
2	{(y,1)}	{(y,1), (z,2)}

Iteration 3 (fixed point)

Equations

- $Out(s) = Gen(s) \cup (In(s) - Kill(s))$
- $In(s) = \bigcup_p Out(p), p \in pred(s), s \in stmts$

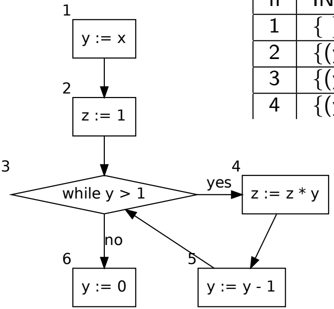


n	IN[n]	OUT[n]
1	{ }	{(y,1)}
2	{(y,1)}	{(y,1), (z,2)}
3	{(y,1), (z,2), (y,5), (z,4)}	{(y,1), (z,2), (y,5), (z,4)}

Iteration 3 (fixed point)

Equations

- $Out(s) = Gen(s) \cup (In(s) - Kill(s))$
- $In(s) = \bigcup_p Out(p), p \in pred(s), s \in stmts$

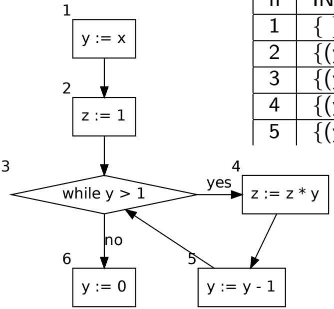


n	IN[n]	OUT[n]
1	{ }	{(y,1)}
2	{(y,1)}	{(y,1), (z,2)}
3	{(y,1), (z,2), (y,5), (z,4)}	{(y,1), (z,2), (y,5), (z,4)}
4	{(y,1), (z,2), (y,5), (z,4)}	{(y,1), (y,5), (z,4)}

Iteration 3 (fixed point)

Equations

- $Out(s) = Gen(s) \cup (In(s) - Kill(s))$
- $In(s) = \bigcup_p Out(p), p \in pred(s), s \in stmts$

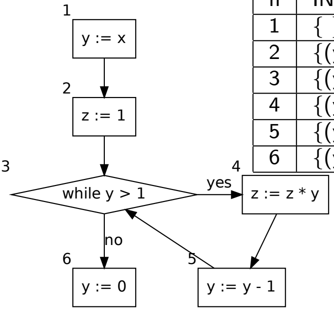


n	IN[n]	OUT[n]
1	{ }	{(y,1)}
2	{(y,1)}	{(y,1), (z,2)}
3	{(y,1), (z,2), (y,5), (z,4)}	{(y,1), (z,2), (y,5), (z,4)}
4	{(y,1), (z,2), (y,5), (z,4)}	{(y,1), (y,5), (z,4)}
5	{(y,1), (y,5), (z,4)}	{(y,5), (z,4)}

Iteration 3 (fixed point)

Equations

- $Out(s) = Gen(s) \cup (In(s) - Kill(s))$
- $In(s) = \bigcup_p Out(p), p \in pred(s), s \in stmts$



n	IN[n]	OUT[n]
1	{ }	{(y,1)}
2	{(y,1)}	{(y,1), (z,2)}
3	{(y,1), (z,2), (y,5), (z,4)}	{(y,1), (z,2), (y,5), (z,4)}
4	{(y,1), (z,2), (y,5), (z,4)}	{(y,1), (y,5), (z,4)}
5	{(y,1), (y,5), (z,4)}	{(y,5), (z,4)}
6	{(y,1), (z,2), (y,5), (z,4)}	{(y,6), (z,2), (z,4)}

Available expressions

Available expressions

- Given a program point u, this algorithm identifies the expressions whose results at u are the same as their previously computed values (regardless of the execution paths that reach u) [**?**].
- An expression `x + y` is available at a program point u if every path from the start node to u evaluates `x + y`, and after the last evaluation there is no subsequent assignment either to `x` or `y` [**?**].

Questions:

- what is the “shape” of the abstraction data type ?

Questions:

- what is the “shape” of the abstraction data type ?
- what meet operator is the best fit for our problem?
- what CFG should we use (forward or backward)?
- how would you define the Gen and Kill functions?

Available Expressions

Informal definition

For every vertice `(from, to)` in the control flow, we check if `assignment(v, exp) := to` holds (`to` is an assignment).

Available Expressions

Informal definition

For every vertice `(from, to)` in the control flow, we check if `assignment(v, exp) := to` holds (`to` is an assignment).If this is the case, we remove all expressions that use the variable `v` (Kill) from the abstraction. For every statement, we include (colorblueGen) into the abstraction every (complex) expression that uses a variable.

Available Expressions

Equations

- $Out(s) = Gen(s) \cup (In(s) - Kill(s))$
- $In(s) = \bigcap_p Out(p), p \in pred(s), s \in stmts$

Available Expressions

Equations

- $Out(s) = Gen(s) \cup (In(s) - Kill(s))$
- $In(s) = \bigcap_p Out(p), p \in pred(s), s \in stmts$

Iterative algorithm

```
// initialization
Out(Start) = {};
for(s in stmts) do Out(s) = U   // all complex expressions

while(changes to any out occur) do
  for(s in stmts)
    // compute In(s)
    // compute Out(s)
```

Consider the example from [?]

Example 02 (WHILE language)

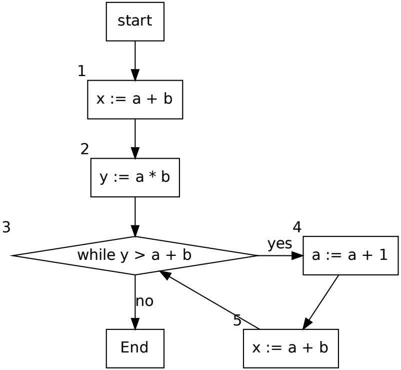
```
x := a + b;              (1)
y := a * b;              (2)
while(y > a + b) do      (3)
  a := a + 1;            (4)
  x := a + b;            (5)
```

Consider the example from [?]

Example 02 (WHILE language)

```
x := a + b;             (1)
y := a * b;             (2)
while(y > a + b) do     (3)
  a := a + 1;           (4)
  x := a + b;           (5)
```

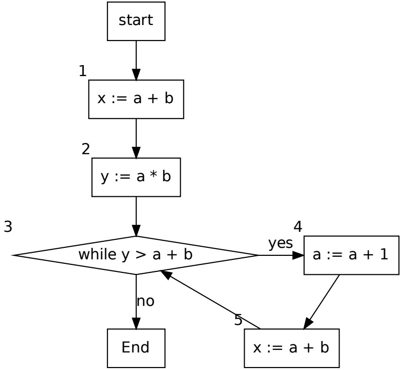
Control Flow Graph



Iteration 1

Equations

- $Out(s) = Gen(s) \cup (In(s) - Kill(s))$
- $In(s) = \bigcap_p Out(p), p \in pred(s), s \in stmts$

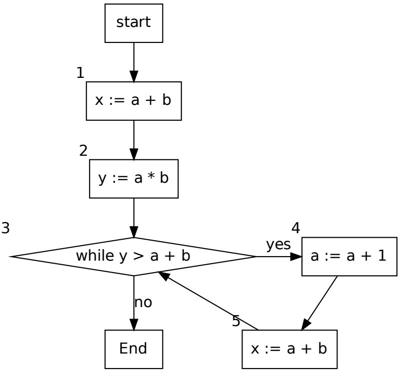


n	IN[n]	OUT[n]
1	{ }	{ a + b }

Iteration 1

Equations

- $Out(s) = Gen(s) \cup (In(s) - Kill(s))$
- $In(s) = \bigcap_p Out(p), p \in pred(s), s \in stmts$

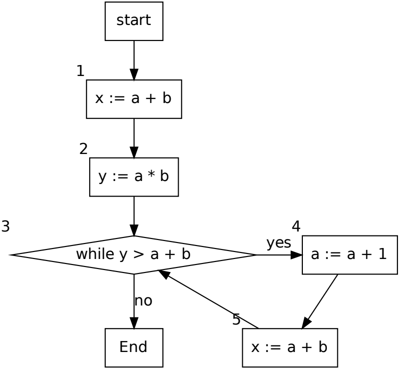


n	IN[n]	OUT[n]
1	{ }	{ a + b }
2	{ a + b }	{ a + b, a * b }

Iteration 1

Equations

- $Out(s) = Gen(s) \cup (In(s) - Kill(s))$
- $In(s) = \bigcap_p Out(p), p \in pred(s), s \in stmts$

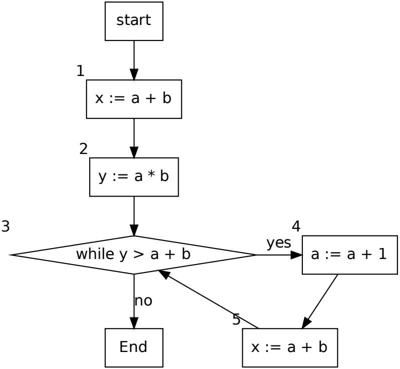


n	IN[n]	OUT[n]
1	{ }	{ a + b }
2	{ a + b }	{ a + b, a * b }
3	{ a + b, a * b }	{ a + b, a * b }

Iteration 1

Equations

- $Out(s) = Gen(s) \cup (In(s) - Kill(s))$
- $In(s) = \bigcap_p Out(p), p \in pred(s), s \in stmts$

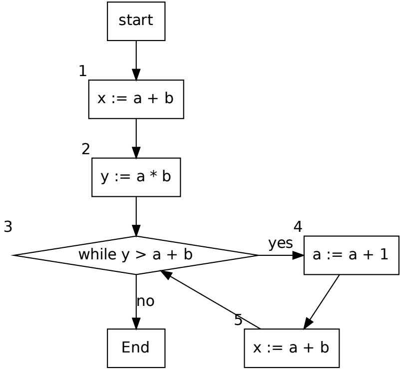


n	IN[n]	OUT[n]
1	{ }	{ a + b }
2	{ a + b }	{ a + b, a * b }
3	{ a + b, a * b }	{ a + b, a * b }
4	{ a + b, a * b }	{ }

Iteration 1

Equations

- $Out(s) = Gen(s) \cup (In(s) - Kill(s))$
- $In(s) = \bigcap_p Out(p), p \in pred(s), s \in stmts$

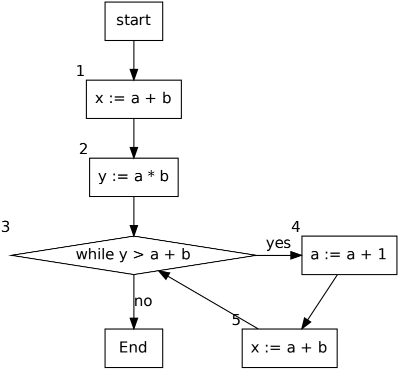


n	IN[n]	OUT[n]
1	{ }	{ a + b }
2	{ a + b }	{ a + b, a * b }
3	{ a + b, a * b }	{ a + b, a * b }
4	{ a + b, a * b }	{ }
5	{ }	{ a + b }

Iteration 2 (fixed point)

Equations

- $Out(s) = Gen(s) \cup (In(s) - Kill(s))$
- $In(s) = \bigcap_p Out(p), p \in pred(s), s \in stmts$

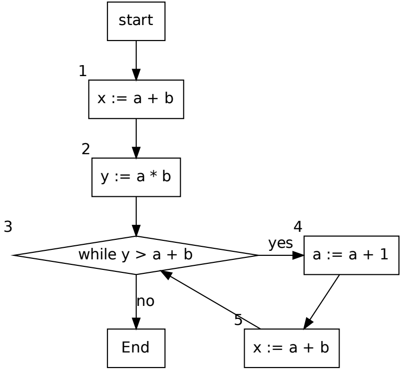


n	IN[n]	OUT[n]
1	{ }	{ a + b }

Iteration 2 (fixed point)

Equations

- $Out(s) = Gen(s) \cup (In(s) - Kill(s))$
- $In(s) = \bigcap_p Out(p), p \in pred(s), s \in stmts$

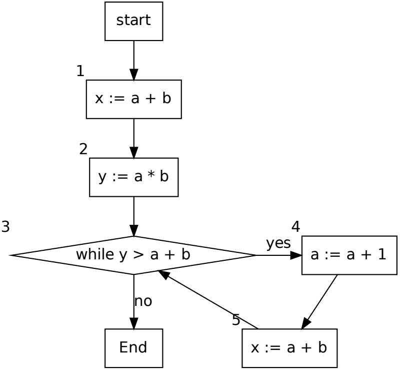


n	IN[n]	OUT[n]
1	{ }	{ a + b }
2	{ a + b }	{ a + b, a * b }

Iteration 2 (fixed point)

Equations

- $Out(s) = Gen(s) \cup (In(s) - Kill(s))$
- $In(s) = \bigcap_p Out(p), p \in pred(s), s \in stmts$

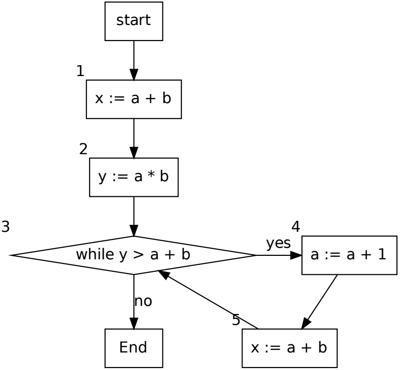


n	IN[n]	OUT[n]
1	{ }	{ a + b }
2	{ a + b }	{ a + b, a * b }
3	{ a + b }	{ a + b }

Iteration 2 (fixed point)

Equations

- $Out(s) = Gen(s) \cup (In(s) - Kill(s))$
- $In(s) = \bigcap_p Out(p), p \in pred(s), s \in stmts$

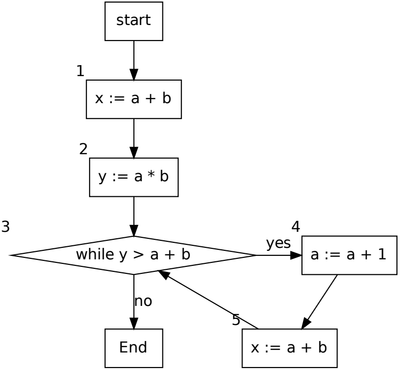


n	IN[n]	OUT[n]
1	{ }	{ a + b }
2	{ a + b }	{ a + b, a * b }
3	{ a + b }	{ a + b }
4	{ a + b }	{ }

Iteration 2 (fixed point)

Equations

- $Out(s) = Gen(s) \cup (In(s) - Kill(s))$
- $In(s) = \bigcap_p Out(p), p \in pred(s), s \in stmts$



n	IN[n]	OUT[n]
1	{ }	{ a + b }
2	{ a + b }	{ a + b, a * b }
3	{ a + b }	{ a + b }
4	{ a + b }	{ }
5	{ }	{ a + b }

Live variable analysis

Live variable analysis

- A variable v is live at the exit of a statement s if there is a path from s (that defines v) to another statement u that uses the variable v; and along this path, there is no other assignment to variable v.

Questions:

- what is the “shape” of the abstraction data type?

Questions:

- what is the “shape” of the abstraction data type?
- what meet operator is the best fit for our problem?

Questions:

- what is the “shape” of the abstraction data type?
- what meet operator is the best fit for our problem?
- what CFG should we use (forward or backward)?

Questions:

- what is the "shape" of the abstraction data type?
- what meet operator is the best fit for our problem?
- what CFG should we use (forward or backward)?
- how would you define the Gen and Kill functions?

Live Variable

Informal definition

For every vertice `(from, to)` in the control flow, we check if `assignment(v, exp) := from` holds (`from` is an assignment).

Live Variable

Informal definition

For every vertice `(from, to)` in the control flow, we check if `assignment(v, exp) := from` holds (`from` is an assignment).If this is the case, we do not expose upwards the need of a previous definition of `v` (unless `v` also appears in `exp`). Otherwise, we expose upwards every variable that `from` uses.

Live Variable Algorithm

Equations

- $In(s) = Gen(s) \cup (Out(s) - Kill(s))$
- $Out(s) = \bigcup_p In(p), p \in successors(s), s \in stmts$

Live Variable Algorithm

Equations

- $In(s) = Gen(s) \cup (Out(s) - Kill(s))$
- $Out(s) = \bigcup_p In(p), p \in successors(s), s \in stmts$

Iterative algorithm

```
// initialization
In(End) = {};
for(s in stmts) do Out(s) = {}   // all complex expressions

while(changes to any in occur) do
  for(s in stmts)
    // compute Out(s)
    // compute In(s)
```

Consider the example from [?]

Example 03 (While language)

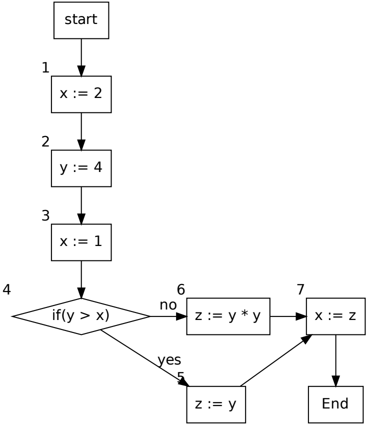
```
x := 2;              (1)
y := 4;              (2)
x := 1;              (3)
if(y > x) then       (4)
  z := y;            (5)
else
  z := y * y;        (6)
x := z;              (7)
```

Consider the example from [?]

Example 03 (While language)

```
x := 2;                 (1)
y := 4;                 (2)
x := 1;                 (3)
if(y > x) then          (4)
  z := y;               (5)
else
  z := y * y;           (6)
x := z;                 (7)
```

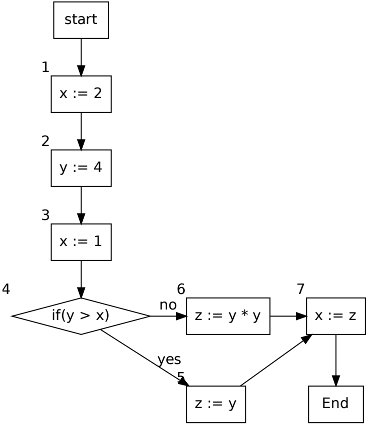
Control Flow Graph



Iteration 1 (fixed point)

Equations

- $In(s) = Gen(s) \cup (Out(s) - Kill(s))$
- $Out(s) = \bigcup_p In(p), p \in successors(s), s \in stmts$

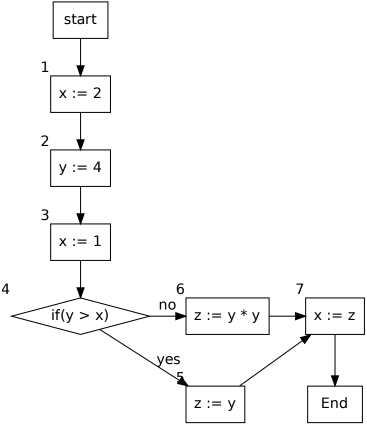


n	IN[n]	OUT[n]
7	{ z }	{ }

Iteration 1 (fixed point)

Equations

- $In(s) = Gen(s) \cup (Out(s) - Kill(s))$
- $Out(s) = \bigcup_p In(p), p \in successors(s), s \in stmts$

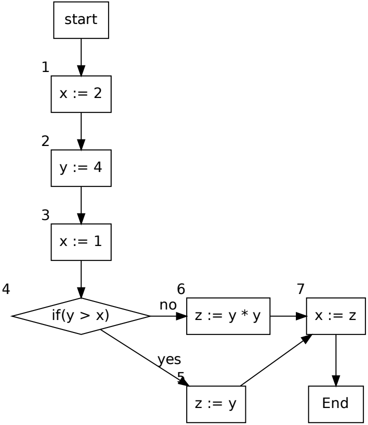


n	IN[n]	OUT[n]
7	{ z }	{ }
6	{ y }	{ z }

Iteration 1 (fixed point)

Equations

- $In(s) = Gen(s) \cup (Out(s) - Kill(s))$
- $Out(s) = \bigcup_p In(p), p \in successors(s), s \in stmts$

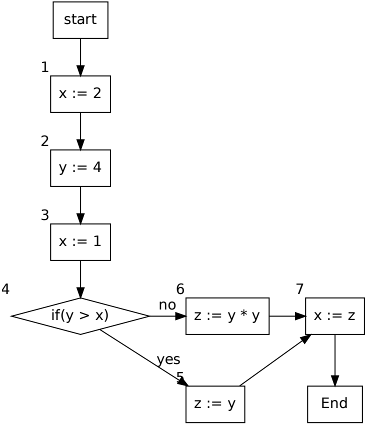


n	IN[n]	OUT[n]
7	{ z }	{ }
6	{ y }	{ z }
5	{ y }	{ z }

Iteration 1 (fixed point)

Equations

- $In(s) = Gen(s) \cup (Out(s) - Kill(s))$
- $Out(s) = \bigcup_p In(p), p \in successors(s), s \in stmts$

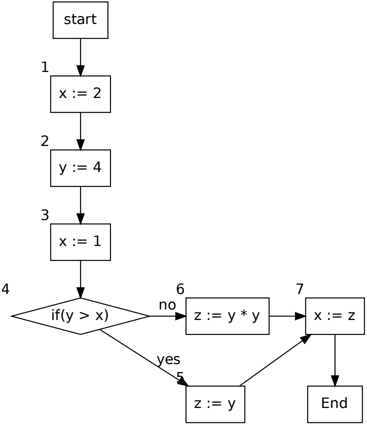


n	IN[n]	OUT[n]
7	{ z }	{ }
6	{ y }	{ z }
5	{ y }	{ z }
4	{ x, y }	{ y }

Iteration 1 (fixed point)

Equations

- $In(s) = Gen(s) \cup (Out(s) - Kill(s))$
- $Out(s) = \bigcup_p In(p), p \in successors(s), s \in stmts$

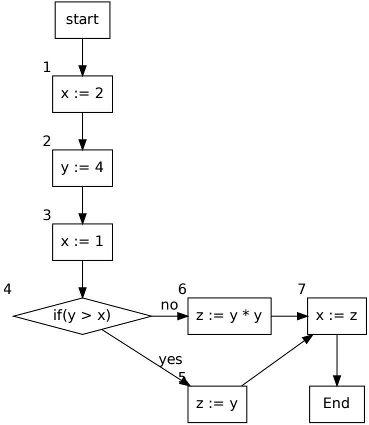


n	IN[n]	OUT[n]
7	{ z }	{ }
6	{ y }	{ z }
5	{ y }	{ z }
4	{ x, y }	{ y }
3	{ y }	{ x, y}

Iteration 1 (fixed point)

Equations

- $In(s) = Gen(s) \cup (Out(s) - Kill(s))$
- $Out(s) = \bigcup_p In(p), p \in successors(s), s \in stmts$

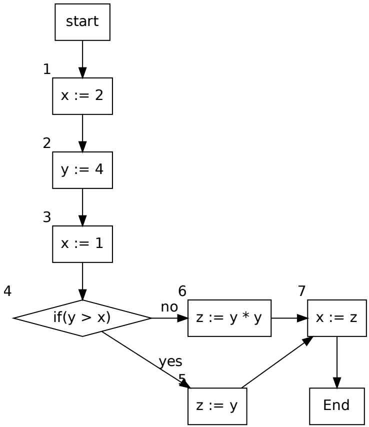


n	IN[n]	OUT[n]
7	{ z }	{ }
6	{ y }	{ z }
5	{ y }	{ z }
4	{ x, y }	{ y }
3	{ y }	{ x, y}
2	{ }	{ y }

Iteration 1 (fixed point)

Equations

- $In(s) = Gen(s) \cup (Out(s) - Kill(s))$
- $Out(s) = \bigcup_p In(p), p \in successors(s), s \in stmts$

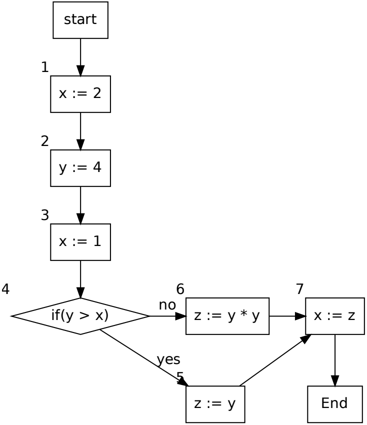


n	IN[n]	OUT[n]
7	{ z }	{ }
6	{ y }	{ z }
5	{ y }	{ z }
4	{ x, y }	{ y }
3	{ y }	{ x, y}
2	{ }	{ y }
1	{ }	{ }

Iteration 1 (fixed point)

Equations

- $In(s) = Gen(s) \cup (Out(s) - Kill(s))$
- $Out(s) = \bigcup_p In(p), p \in successors(s), s \in stmts$



n	IN[n]	OUT[n]
7	{ z }	{ }
6	{ y }	{ z }
5	{ y }	{ z }
4	{ x, y }	{ y }
3	{ y }	{ x, y}
2	{ }	{ y }
1	{ }	{ }

DataFlow analysis in the JimpleFramework

Project

- G1 Dead Variable Analysis
- G2 Very Busy Expressions Analysis
- G3 Optmizations (e.g., constant propagation and dead code elimination)

Very busy expressions

Very busy expressions

- Anticipable Expressions Analysis: An expression $e \in \mathit{Expr}$ (binary arithmetic expression) is anticipable at a program point u if every path from u to *End* contains a computation of e which is not preceded by an assignment to any operand of e [**?**].
- An expression is very busy at the exit from a label if, no matter what path is taken from the label, the expression must always be used before any of the variables occurring in it are redefined. [**?**].

Questions:

- what is the “shape” of the abstraction data type ?

Questions:

- what is the “shape” of the abstraction data type ?
- what meet operator is the best fit for our problem?
- what CFG should we use (forward or backward)?
- how would you define the Gen and Kill functions?

Very Busy Expressions Algorithm

Equations

- $Out(s) = \bigcap_p In(p), p \in successors(s), s \in stmts$
- $In(s) = Gen(s) \cup (Out(s) - Kill(s))$

Very Busy Expressions Algorithm

Equations

- $Out(s) = \bigcap_p In(p), p \in successors(s), s \in stmts$
- $In(s) = Gen(s) \cup (Out(s) - Kill(s))$

Iterative algorithm

```
// initialization
Un = set of all binary arithmetic expression
Out(End) = {};
for(s in stmts) do Out(s) = Un

while(changes to any out occurs) do
// until achieving a fixed point
   for(s in stmts)
   // compute In(s)
   // compute Out(s)
```

Control Flow Graph

The file VbWhile.pdf hasn't been created from VbWhile.dot yet.
Run 'dot -Tpdf -o VbWhile.pdf VbWhile.dot' to create it.
Or invoke LaTeX with the -shell-escape option to have this done autor

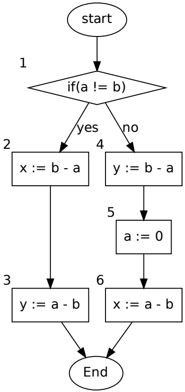
Example 04 (WHILE language)

```
if(a != b) then    (1)
  x := b - a;      (2)
  y := a - b;      (3)
else
  y := b - a;      (4)
  a = 0;           (5)
  x = a- b;        (6)
```

Iteration 0

Equations

- $Out(s) = \bigcap_p In(p), p \in successors(s), s \in stmts$
- $In(s) = Gen(s) \cup (Out(s) - Kill(s))$

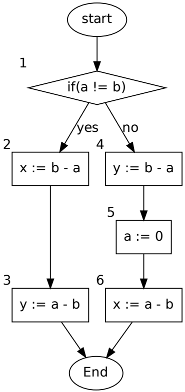


n	IN[n]	OUT[n]
6	{ }	{ a - b, b - a }
5	{ }	{ a - b, b - a }
4	{ }	{ a - b, b - a }
3	{ }	{ a - b, b - a }
2	{ }	{ a - b, b - a }
1	{ }	{ }

Iteration 1

Equations

- $Out(s) = \bigcap_p In(p), p \in successors(s), s \in stmts$
- $In(s) = Gen(s) \cup (Out(s) - Kill(s))$

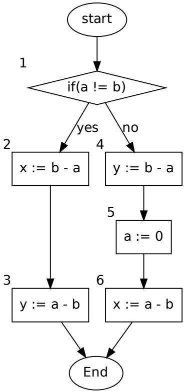


n	IN[n]	OUT[n]
6	{ a - b }	{ }

Iteration 1

Equations

- $Out(s) = \bigcap_p In(p), p \in successors(s), s \in stmts$
- $In(s) = Gen(s) \cup (Out(s) - Kill(s))$

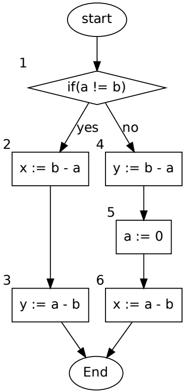


n	IN[n]	OUT[n]
6	{ a - b }	{ }
5	{ }	{a - b}

Iteration 1

Equations

- $Out(s) = \bigcap_p In(p), p \in successors(s), s \in stmts$
- $In(s) = Gen(s) \cup (Out(s) - Kill(s))$

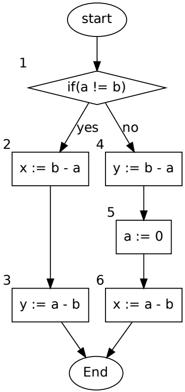


n	IN[n]	OUT[n]
6	{ a - b }	{ }
5	{ }	{a - b}
4	{ b - a }	{ }

Iteration 1

Equations

- $Out(s) = \bigcap_p In(p), p \in successors(s), s \in stmts$
- $In(s) = Gen(s) \cup (Out(s) - Kill(s))$

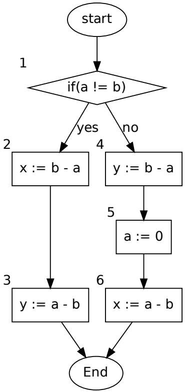


n	IN[n]	OUT[n]
6	{ a - b }	{ }
5	{ }	{a - b}
4	{ b - a }	{ }
3	{ a - b }	{ }

Iteration 1

Equations

- $Out(s) = \bigcap_p In(p), p \in successors(s), s \in stmts$
- $In(s) = Gen(s) \cup (Out(s) - Kill(s))$

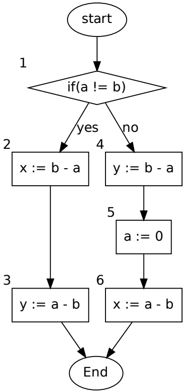


n	IN[n]	OUT[n]
6	{ a - b }	{ }
5	{ }	{a - b}
4	{ b - a }	{ }
3	{ a - b }	{ }
2	{ a - b, b - a }	{ a - b }

Iteration 1

Equations

- $Out(s) = \bigcap_p In(p), p \in successors(s), s \in stmts$
- $In(s) = Gen(s) \cup (Out(s) - Kill(s))$

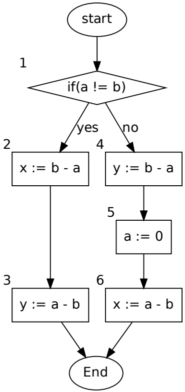


n	IN[n]	OUT[n]
6	{ a - b }	{ }
5	{ }	{a - b}
4	{ b - a }	{ }
3	{ a - b }	{ }
2	{ a - b, b - a }	{ a - b }
1	{ b - a }	{ }

Iteration 1

Equations

- $Out(s) = \bigcap_p In(p), p \in successors(s), s \in stmts$
- $In(s) = Gen(s) \cup (Out(s) - Kill(s))$



n	IN[n]	OUT[n]
6	{ a - b }	{ }
5	{ }	{a - b}
4	{ b - a }	{ }
3	{ a - b }	{ }
2	{ a - b, b - a }	{ a - b }
1	{ b - a }	{ }

Example 05 (Jimple (simplified))

```
  if(a = b) goto label1 (1)
  x := b - a;           (2)
  y := a - b;           (3)
  goto label2           (7)
label 1:
  y := b - a;           (4)
  a = 0;                (5)
  x = a- b;             (6)
label 2:
  return;               (8)
```

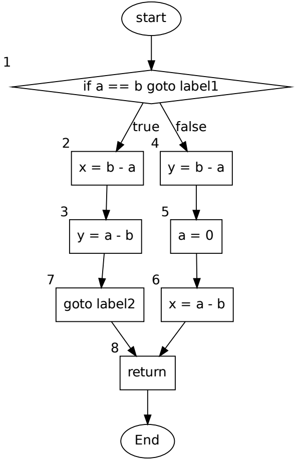
Control Flow Graph

The file VbJimple.pdf hasn't been
Run 'dot -Tpdf -o VbJimple.pd
Or invoke LATEX with the -shell-e

Iteration 1

Equations

- $Out(s) = \bigcap_p In(p), p \in successors(s), s \in stmts$
- $In(s) = Gen(s) \cup (Out(s) - Kill(s))$

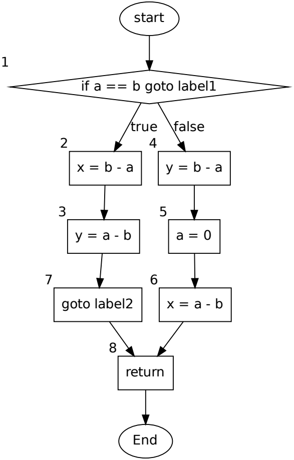


n	IN[n]	OUT[n]
8	{ }	{ }

Iteration 1

Equations

- $Out(s) = \bigcap_p In(p), p \in successors(s), s \in stmts$
- $In(s) = Gen(s) \cup (Out(s) - Kill(s))$

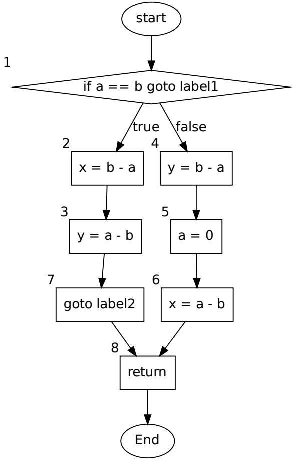


n	IN[n]	OUT[n]
8	{ }	{ }
7	{ }	{ }

Iteration 1

Equations

- $Out(s) = \bigcap_p In(p), p \in successors(s), s \in stmts$
- $In(s) = Gen(s) \cup (Out(s) - Kill(s))$

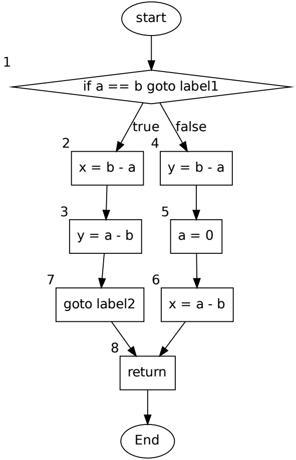


n	IN[n]	OUT[n]
8	{ }	{ }
7	{ }	{ }
6	{ }	{ }

Iteration 1

Equations

- $Out(s) = \bigcap_p In(p), p \in successors(s), s \in stmts$
- $In(s) = Gen(s) \cup (Out(s) - Kill(s))$

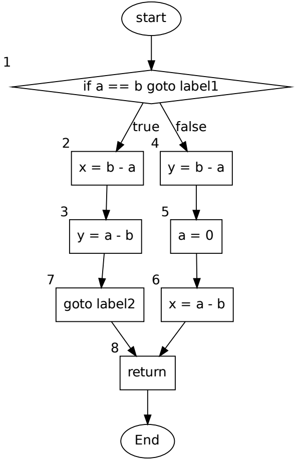


n	IN[n]	OUT[n]
8	{ }	{ }
7	{ }	{ }
6	{ }	{ }
5	{ }	{ }

Iteration 1

Equations

- $Out(s) = \bigcap_p In(p), p \in successors(s), s \in stmts$
- $In(s) = Gen(s) \cup (Out(s) - Kill(s))$

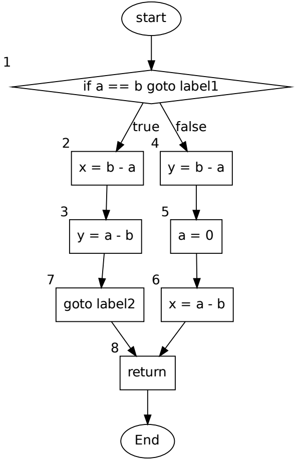


n	IN[n]	OUT[n]
8	{ }	{ }
7	{ }	{ }
6	{ }	{ }
5	{ }	{ }
4	{ }	{ }

Iteration 1

Equations

- $Out(s) = \bigcap_p In(p), p \in successors(s), s \in stmts$
- $In(s) = Gen(s) \cup (Out(s) - Kill(s))$

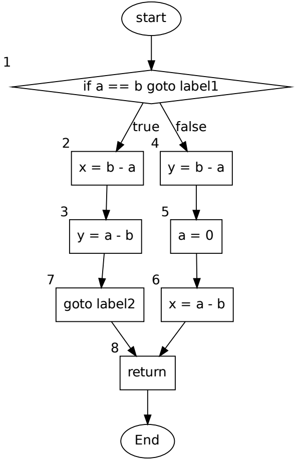


n	IN[n]	OUT[n]
8	{ }	{ }
7	{ }	{ }
6	{ }	{ }
5	{ }	{ }
4	{ }	{ }
3	{ }	{ }

Iteration 1

Equations

- $Out(s) = \bigcap_p In(p), p \in successors(s), s \in stmts$
- $In(s) = Gen(s) \cup (Out(s) - Kill(s))$

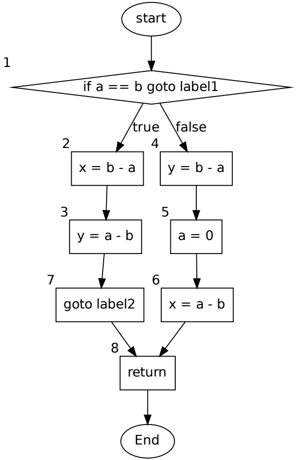


n	IN[n]	OUT[n]
8	{ }	{ }
7	{ }	{ }
6	{ }	{ }
5	{ }	{ }
4	{ }	{ }
3	{ }	{ }
2	{ }	{ }

Iteration 1

Equations

- $Out(s) = \bigcap_p In(p), p \in successors(s), s \in stmts$
- $In(s) = Gen(s) \cup (Out(s) - Kill(s))$

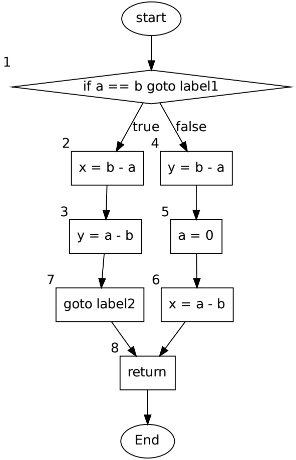


n	IN[n]	OUT[n]
8	{ }	{ }
7	{ }	{ }
6	{ }	{ }
5	{ }	{ }
4	{ }	{ }
3	{ }	{ }
2	{ }	{ }
1	{ }	{ }

Iteration 1

Equations

- $Out(s) = \bigcap_p In(p), p \in successors(s), s \in stmts$
- $In(s) = Gen(s) \cup (Out(s) - Kill(s))$



n	IN[n]	OUT[n]
8	{ }	{ }
7	{ }	{ }
6	{ }	{ }
5	{ }	{ }
4	{ }	{ }
3	{ }	{ }
2	{ }	{ }
1	{ }	{ }

References