Lecture 3: WHILE3ADDR, Control-Flow Graphs and Intro to Data-Flow Analaysis

17-355/17-665/17-819: Program Analysis

Rohan Padhye

September 4, 2025

* Course materials developed with Jonathan Aldrich and Claire Le Goues

Review: WHILE abstract syntax

S	statements
a	arithmetic expressions (AExp)
x, y	program variables (Vars)
n	number literals
b	boolean expressions (BExp)

$$
\begin{array}{rcl}
S & ::= & x := a \\
 & \mid & \text{skip} \\
 & \mid & S_1;\ S_2 \\
 & \mid & \text{if } b \text{ then } S_1 \text{ else } S_2 \\
 & \mid & \text{while } b \text{ do } S
\end{array}
\qquad
\begin{array}{rcl}
b & ::= & \text{true} \\
 & \mid & \text{false} \\
 & \mid & \text{not } b \\
 & \mid & b_1\ op_b\ b_2 \\
 & \mid & a_1\ op_r\ a_2
\end{array}
\qquad
\begin{array}{rcl}
a & ::= & x \\
 & \mid & n \\
 & \mid & a_1\ op_a\ a_2
\end{array}
\qquad
\begin{array}{rcl}
op_b & ::= & \text{and} \mid \text{or} \\
op_r & ::= & < \ \mid\ \leqslant \ \mid\ = \\
 & \mid & > \ \mid\ \geqslant \\
op_a & ::= & + \mid - \mid * \mid /
\end{array}
$$

WHILE syntax

- Abstract representation that corresponds well to concrete syntax
- Useful for recursive or inductive reasoning
- Sometimes challenging to track how data and control flows in program execution order
- 3-address-code is commonly used by compilers to represent imperative language code.
  - AST -> 3-address transformation is straightforward.

WHILE3ADDR

- w = x * y + z
- 1: t = x * y
  2: w = t + z

- if b then S1 else S2
- 1: if b then goto 4
  2: S2
  3: goto 5
  4: S1
  5: …

WHILE3ADDR: An Intermediate Representation

Simpler, more uniform than WHILE syntax

Categories:

$I \in$ **Instruction** instructions
$x, y \in$ **Var** variables
$n \in$ **Num** number literals

Syntax:

I ::= x := n | x := y | x := y op z
| goto n | if x op_r 0 goto n
op_a ::= + | - | * | / | …
op_r ::= < | ≤ | = | > | ≥ | ...
$P \in$ **Num** → I

Exercise: Translate ***while b do S*** to WHILE3ADDR

Categories:

I ∈ **Instruction** instructions
x, y ∈ **Var** variables
n ∈ **Num** number literals

Syntax:

```
I    ::= x := n | x := y | x := y op z
       |   goto n | if x opr 0 goto n
opa  ::= + | - | * | / | …
opr  ::= < | ≤ | = | > | ≥ | ...
P  ∈ Num → I
```

While3Addr Extensions (more later)

```
I  ::=   x := n | x := y | x := y op z |  goto n | if x opᵣ 0 goto n
       | x := f(y)
       | return x
       | x := y.m(z)
       | read x
       | print x
       | x := &p
       | x := *p
       | *p := x
       | x := y.f
       | x.f := y
```

WHILE3ADDR Semantics

- Configuration (state) includes environment + program counter:

$$c \in E \times \mathbb{N}$$

- Evaluation occurs with respect to a global program that maps labels to instructions: $P \in \mathbb{N} \rightarrow I$

$$P \vdash < E, n > \rightsquigarrow < E', n' >$$

$$\frac{P(n) = x := m}{P \vdash \langle E, n \rangle \rightsquigarrow \langle E[x \mapsto m], n+1 \rangle}\ \textit{step-const}$$

$$\frac{P[n] = x := y}{P \vdash \langle E, n \rangle \rightsquigarrow \langle E[x \mapsto E(y)], n+1 \rangle}\ \textit{step-copy}$$

$$\frac{P(n) = x := y\ op\ z \quad E(y)\ \mathbf{op}\ E(z) = m}{P \vdash \langle E, n \rangle \rightsquigarrow \langle E[x \mapsto m], n+1 \rangle}\ \textit{step-arith}$$

$$\frac{P(n) = \text{goto}\ m}{P \vdash \langle E, n \rangle \rightsquigarrow \langle E, m \rangle}\ \textit{step-goto}$$

$$\frac{P(n) = \text{if}\ x\ op_r\ 0\ \text{goto}\ m \quad E(x)\ \mathbf{op_r}\ 0 = true}{P \vdash \langle E, n \rangle \rightsquigarrow \langle E, m \rangle}\ \textit{step-iftrue}$$

$$\frac{P(n) = \text{if}\ x\ op_r\ 0\ \text{goto}\ m \quad E(x)\ \mathbf{op_r}\ 0 = false}{P \vdash \langle E, n \rangle \rightsquigarrow \langle E, n+1 \rangle}\ \textit{step-iffalse}$$

Data-Flow Analysis

Computes universal properties about program state at specific program points. (e.g. will *x* be zero at line 7?)

- About program state
  - About data store (e.g. variables, heap memory)
  - Not about control (e.g. termination, performance)
- At program points
  - Statically identifiable (e.g. line 7, or when foo() calls bar())
  - Not dynamically computed (E.g. when x is 12 or when foo() is invoked 12 times)
- Universal
  - Reasons about all possible executions (always/never/maybe)
  - Not about specific program paths (see: symbolic execution, testing)

Abstraction

$$\sigma \in Var \rightarrow L$$

$$\alpha : \mathbb{Z} \rightarrow L$$

Abstraction

$\sigma \in Var \to L$

$\alpha : \mathbb{Z} \to L$

Zero Analysis

$L = \{Z, N, \top\}$

$$\alpha_Z(0) = Z$$
$$\alpha_Z(n) = N \text{where } n \neq 0$$

Flow Functions for Zero Analysis

A flow function maps values from σ to σ

$f[\![I]\!]$ -- flow across instruction I (think: "abstract semantics")

$$f_Z[\![x := 0]\!](\sigma) =$$

$$f_Z[\![x := n]\!](\sigma) =$$

$$f_Z[\![x := y]\!](\sigma) =$$

$$f_Z[\![x := y \; op \; z]\!](\sigma) =$$

$$f_Z[\![\text{goto } n]\!](\sigma) =$$

$$f_Z[\![\text{if } x = 0 \text{ goto } n]\!](\sigma) =$$

Flow Functions for Zero Analysis

A flow function maps values from σ to σ

$f[\![I]\!]$ -- flow across instruction I (think: “abstract semantics”)

$$f_Z[\![x := 0]\!](\sigma) = \sigma[x \mapsto Z]$$

$$f_Z[\![x := n]\!](\sigma) = \sigma[x \mapsto N] \text{ where } n \neq 0$$

$$f_Z[\![x := y]\!](\sigma) = \sigma[x \mapsto \sigma(y)]$$

$$f_Z[\![x := y \; op \; z]\!](\sigma) = \sigma[x \mapsto \top]$$

$$f_Z[\![\text{goto } n]\!](\sigma) = \sigma$$

$$f_Z[\![\text{if } x = 0 \text{ goto } n]\!](\sigma) = \sigma$$

Flow Functions for Zero Analysis

Specializing for Precision

$$f_Z[\![x := y - y]\!](\sigma) \quad =$$

$$f_Z[\![x := y + z]\!](\sigma) \quad =$$

Flow Functions for Zero Analysis

Specializing for Precision

$$
\begin{aligned}
f_Z[\![x := y - y]\!](\sigma) &= \sigma[x \mapsto Z] \\
f_Z[\![x := y + z]\!](\sigma) &= \sigma[x \mapsto \sigma(y)] \quad \text{where } \sigma(z) = Z
\end{aligned}
$$

Exercise: Define another flow function for some arithmetic instruction and certain conditions where you can also provide a more precise result than ⊤

Flow Functions for Zero Analysis

Specializing for Precision

$$f_Z[\![\text{if } x = 0 \text{ goto } n]\!]_T(\sigma) \quad =$$
$$f_Z[\![\text{if } x = 0 \text{ goto } n]\!]_F(\sigma) \quad =$$

Flow Functions for Zero Analysis

Specializing for Precision

$$
\begin{aligned}
f_Z[\![\text{if } x = 0 \text{ goto } n]\!]_T(\sigma) &= \sigma[x \mapsto Z] \\
f_Z[\![\text{if } x = 0 \text{ goto } n]\!]_F(\sigma) &= \sigma[x \mapsto N]
\end{aligned}
$$

Exercise: Define a flow function for a conditional branch testing whether a variable x < 0

Control-flow Graphs

$$
\begin{aligned}
1:&\quad \text{if } x = 0 \text{ goto } 4 \\
2:&\quad y := 0 \\
3:&\quad \text{goto } 6 \\
4:&\quad y := 1 \\
5:&\quad x := 1 \\
6:&\quad z := y
\end{aligned}
$$

$\boxed{1\text{: if } x = 0 \text{ goto } 4}$

$\boxed{2\text{: } y \coloneqq 0}$

$\boxed{3\text{: goto } 6}$

$\boxed{4\text{: } y \coloneqq 1}$

$\boxed{5\text{: } x \coloneqq 1}$

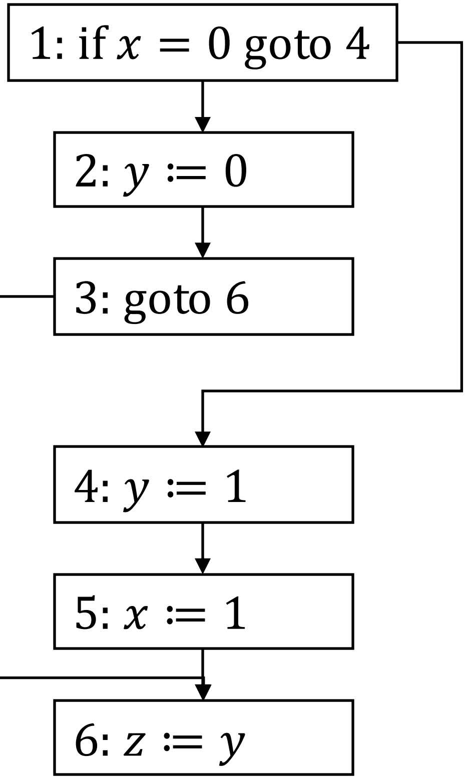
$\boxed{6\text{: } z \coloneqq y}$

Nodes = Statements

Edges = (s1, s2) is an edge iff s1 and s2 can be executed consecutively aka ”control flow”

Control-flow Graphs

$$
\begin{array}{ll}
1: & \text{if } x = 0 \text{ goto } 4 \\
2: & y := 0 \\
3: & \text{goto } 6 \\
4: & y := 1 \\
5: & x := 1 \\
6: & z := y
\end{array}
$$



Nodes = Statements

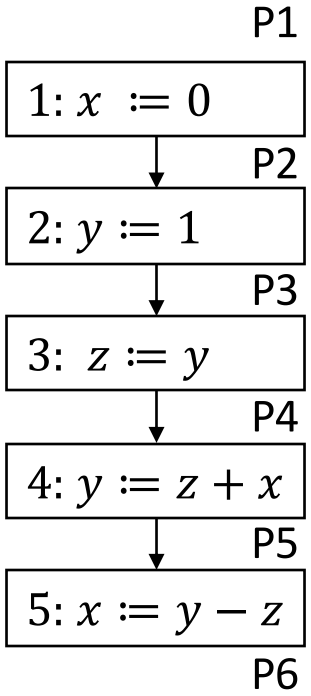
Edges = (s1, s2) is an edge iff s1 and s2 can be executed consecutively aka "control flow"

Common properties of CFGs:

- Weakly connected
- Only one entry node
- Only one exit (terminal) node

Example of Zero Analysis: Straightline Code

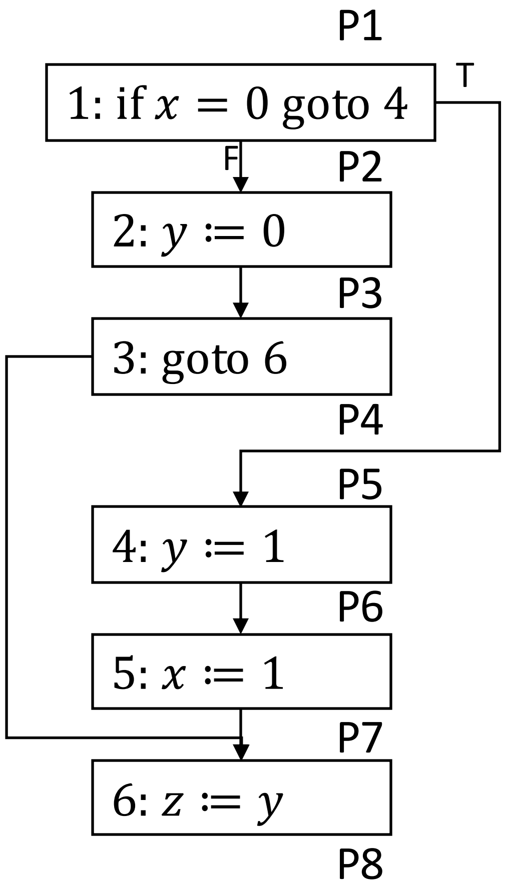
$$
\begin{aligned}
1:&\quad x := 0\\
2:&\quad y := 1\\
3:&\quad z := y\\
4:&\quad y := z + x\\
5:&\quad x := y - z
\end{aligned}
$$



	x	y	z
P1			
P2			
P3			
P4			
P5			
P6			

Example of Zero Analysis: Branching Code

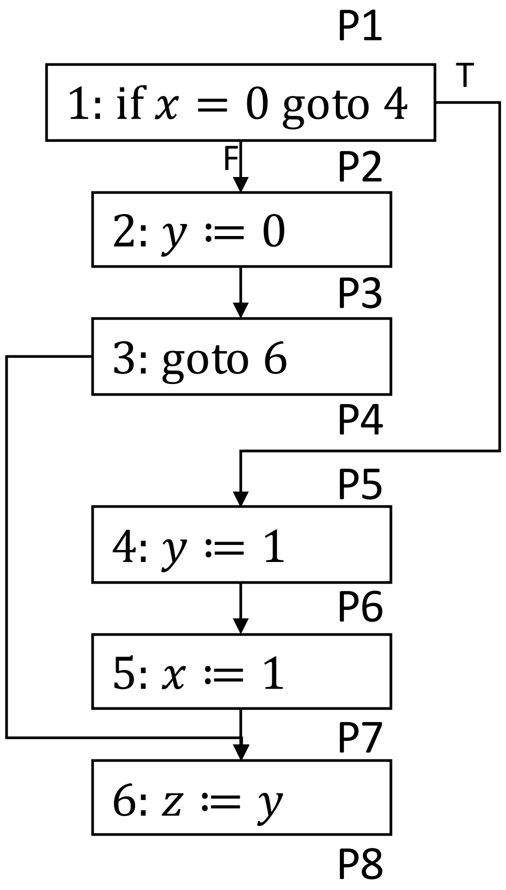
$$\begin{array}{ll} 1: & \text{if } x = 0 \text{ goto } 4 \\ 2: & y := 0 \\ 3: & \text{goto } 6 \\ 4: & y := 1 \\ 5: & x := 1 \\ 6: & z := y \end{array}$$



	x	y	z
P1			
P2			
P3			
P4			
P5			
P6			
P7			
P8			

Example of Zero Analysis: Branching Code

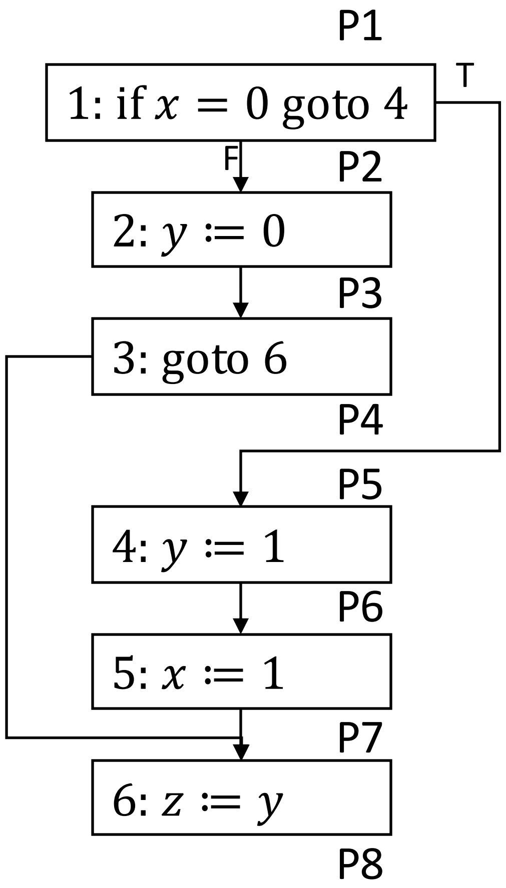
$$
\begin{array}{ll}
1: & \text{if } x = 0 \text{ goto } 4 \\
2: & y := 0 \\
3: & \text{goto } 6 \\
4: & y := 1 \\
5: & x := 1 \\
6: & z := y
\end{array}
$$



	x	y	z
P1	?	?	?
P2	Z_T, N_F	?	?
P3	N	Z	?
P4	N	Z	?
P5	Z	?	?
P6	Z	N	?
P7	N	N?	?
P8	N??	N??	N??

Example of Zero Analysis: Branching Code

$$
\begin{array}{ll}
1: & \text{if } x = 0 \text{ goto } 4 \\
2: & y := 0 \\
3: & \text{goto } 6 \\
4: & y := 1 \\
5: & x := 1 \\
6: & z := y
\end{array}
$$



	x	y	z
P1	?	?	?
P2	Z_T, N_F	?	?
P3	N	Z	?
P4	N	Z	?
P5	Z	?	?
P6	Z	N	?
P7	N	$\top$	?
P8	N	$\top$	$\top$

Next Time

- Lattices
- Definition of a Data-Flow Analysis
- Solution of a Data-Flow Analysis
- Kildall's Algorithm