

Lecture 2 – Control Flow Graphs

SWS128: Software Analysis

Jihyeok Park



2026 Fall

- MiniPy: one def, no calls, lists as the only heap objects.
- Big-step semantics, and an interpreter for it.
- Today: **all executions at once**, without running any.

1. Control Flow Graphs

- Definition

- Examples

- Semantics

2. Construction

- Translation

- Basic Blocks

- Seeing the Graph

- Running It

3. Dominators and SSA

- Dominators

- Fixpoint Iteration

- SSA Form

1. Control Flow Graphs

Definition

Examples

Semantics

2. Construction

Translation

Basic Blocks

Seeing the Graph

Running It

3. Dominators and SSA

Dominators

Fixpoint Iteration

SSA Form

- A **control flow graph** (CFG): **program points** joined by edges, one command per edge.
- The AST says how a program is *nested*. The CFG says where control can *go*.
- Every analysis in this course runs on it.

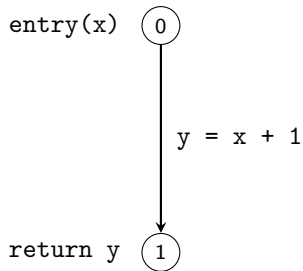
A **control flow graph** $G = (N, E, \text{entry}, \text{exit})$:

- N — **program points**.
- E — edges $p \xrightarrow{c} q$, each with one **command**:

$$c ::= x := e \mid x[e] := e \mid \text{pass} \mid \text{assume}(e)$$

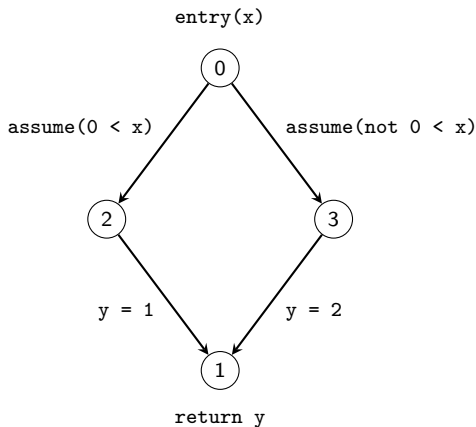
- $\text{entry}, \text{exit} \in N$ — parameters in, return out.
- $\text{assume}(e)$: can be taken only when e is true.
- A **path**: $\text{entry} \rightarrow p_1 \rightarrow \dots \rightarrow p_k$. Every run follows some path.

```
def f(x):  
    y = x + 1  
    return y
```



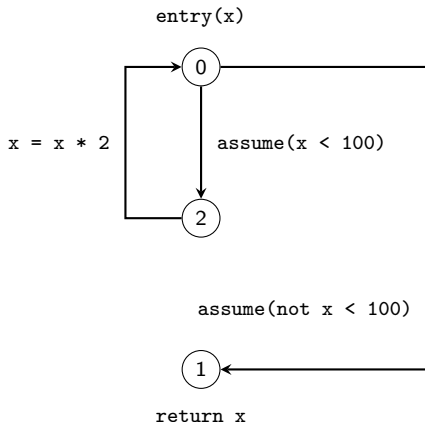
```
def f(x):  
    if 0 < x:  
        y = 1  
    else:  
        y = 2  
    return y
```

- A condition: two assume edges.



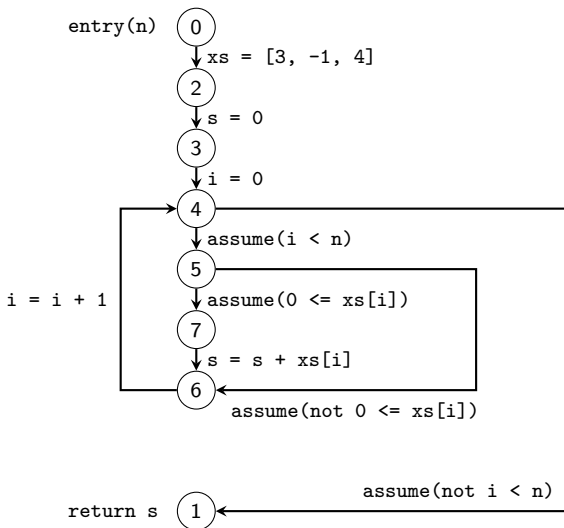
```
def f(x):  
    while x < 100:  
        x = x * 2  
    return x
```

- The body comes **back**: a **loop head** and a **back edge**.



```
def total(n):  
    xs = [3, -1, 4]  
    s = 0  
    i = 0  
    while i < n:  
        if 0 <= xs[i]:  
            s = s + xs[i]  
            i = i + 1  
    return s
```

- Nesting is gone. Only flow remains.



A step takes an edge and runs its command:

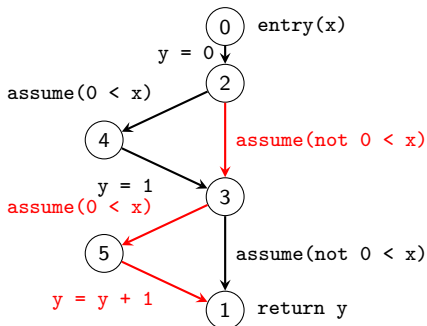
$$\langle p, \sigma \rangle \rightarrow \langle q, \sigma' \rangle \quad \text{iff} \quad p \xrightarrow{c} q \quad \text{and} \quad \sigma \vdash c \Rightarrow \sigma'$$

The new command:

$$\frac{\sigma \vdash e \Rightarrow \text{true}}{\sigma \vdash \text{assume}(e) \Rightarrow \sigma}$$

- A false assume has no rule: that edge cannot be taken.
- A run is a path on which **every assume held**.

```
def f(x):  
    y = 0  
    if 0 < x:  
        y = 1  
    if 0 < x:  
        y = y + 1  
    return y
```



- The red path returns 1. No input does.
- **Over-approximation starts here.**

1. Control Flow Graphs

Definition

Examples

Semantics

2. Construction

Translation

Basic Blocks

Seeing the Graph

Running It

3. Dominators and SSA

Dominators

Fixpoint Iteration

SSA Form

$\text{translate}(s, \text{src}, \text{dst})$: edges that run s from src to dst .

- $x = e$: one edge $\text{src} \xrightarrow{s} \text{dst}$.
- $s_1; s_2$: a fresh point m ; $\text{src} \rightarrow s_1 \rightarrow m \rightarrow s_2 \rightarrow \text{dst}$.
- **if** c : A **else**: B : $\text{src} \xrightarrow{\text{assume}(c)} A \rightarrow \text{dst}$ and $\text{src} \xrightarrow{\text{assume}(\text{not } c)} B \rightarrow \text{dst}$.
- **while** c : A : $\text{src} \xrightarrow{\text{assume}(c)} A \rightarrow \text{src}$ and $\text{src} \xrightarrow{\text{assume}(\text{not } c)} \text{dst}$.
- The program: $\text{translate}(s^+, \text{entry}, \text{exit})$.

```
@dataclass(frozen=True)
class Assume:
    cond: Expr

Cmd = Stmt | Assume

@dataclass(frozen=True)
class Edge:
    id: int
    src: int
    cmd: Cmd
    dst: int
```

- A point is just an int.

```
@dataclass
class CFG:
    params: tuple[str, ...]
    ret: Expr
    edges: list[Edge]
    succ: dict[int, list[Edge]]
    pred: dict[int, list[Edge]]
    loop_heads: set[int]
    entry: int
    exit: int

    # (defaults omitted)
    # bound at the entry: the input
    # evaluated at the exit: the output
```

```
def build(prog: Prog) -> CFG:
    g = CFG(prog.params, prog.ret)
    g.entry = g.add()
    g.exit = g.add()
    _block(g, prog.body, g.entry, g.exit)
    return g

def _block(g: CFG, body: Block, src: int, dst: int) -> None:
    """Edges for `body`, from point src to point dst."""
    if not body:
        g.edge(src, Pass(), dst)
        return
    for st in body[:-1]:
        mid = g.add()
        _stmt(g, st, src, mid)
        src = mid
    _stmt(g, body[-1], src, dst)
```

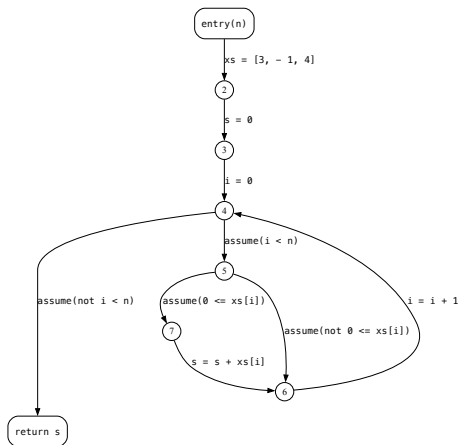
```
def _stmt(g: CFG, st: Stmt, src: int, dst: int) -> None:
    match st:
        case If(cond, then, els):
            _branch(g, cond, then, src, dst)
            _branch(g, UnOp("not", cond), els, src, dst)
        case While(cond, body):
            g.loop_heads.add(src)
            _branch(g, cond, body, src, src)
            g.edge(src, Assume(UnOp("not", cond)), dst)
        case _:
            # x = e | x[e] = e | pass
            g.edge(src, st, dst)

def _branch(g: CFG, cond: Expr, body: Block, src: int, dst: int) -> None:
    """src --assume(cond)--> body --> dst."""
    if not body:
        g.edge(src, Assume(cond), dst)
    else:
        mid = g.add()
        g.edge(src, Assume(cond), mid)
        _block(g, body, mid, dst)
```

- Compilers put a straight-line sequence on one edge: a **basic block**. In total: $xs = [3, -1, 4]$, $s = 0$, $i = 0$ form one block.
- We keep one command per edge. Simpler, and our programs are small.

```
$ python3 cfg.py example.py | dot -Tpdf -o cfg.pdf
```

```
def total(n):  
    xs = [3, -1, 4]  
    s = 0  
    i = 0  
    while i < n:  
        if 0 <= xs[i]:  
            s = s + xs[i]  
            i = i + 1  
    return s
```



dot is Graphviz, optional: graphviz.org/download

```
>>> from syntax import parse
>>> from cfg import build, find
>>> g = build(parse(open("example.py").read()))

>>> len(g.nodes), len(g.edges)
(8, 9)

>>> g.params, g.entry, g.exit
(('n',), 0, 1)

>>> g.loop_heads
{4}

>>> [str(e) for e in g.succ[4]]
['assume(i < n)', 'assume(not i < n)']

>>> e = find(g, "s = s + xs[i]")
>>> e.src, e.dst
(7, 6)
```

1. Control Flow Graphs

Definition

Examples

Semantics

2. Construction

Translation

Basic Blocks

Seeing the Graph

Running It

3. Dominators and SSA

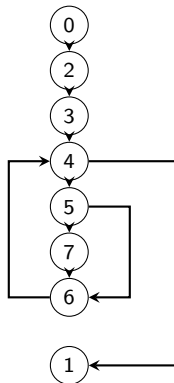
Dominators

Fixpoint Iteration

SSA Form

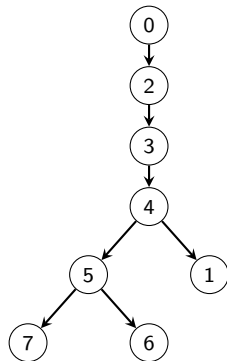
d **dominates** p if **every** path from entry to p passes through d .

- Every point dominates itself. entry dominates everything.
- The **immediate dominator**: the closest strict dominator. They form a tree.
- Meaning: “ d **must** have happened before p .”



the CFG

p	$\text{dom}(p)$
0	$\{0\}$
2	$\{0, 2\}$
3	$\{0, 2, 3\}$
4	$\{0, 2, 3, 4\}$
5	$\{0, 2, 3, 4, 5\}$
7	$\{0, 2, 3, 4, 5, 7\}$
6	$\{0, 2, 3, 4, 5, 6\}$
1	$\{0, 2, 3, 4, 1\}$



the dominator tree

$$\text{dom}(\text{entry}) = \{\text{entry}\} \quad \text{dom}(p) = \{p\} \cup \bigcap_{q \rightarrow p} \text{dom}(q) \quad (p \neq \text{entry})$$

- $\text{dom}(p)$ is defined by the $\text{dom}(q)$ of its predecessors — and around a loop, by itself. **Recursive.**
- So: start from a guess, and apply the definition again and again until it stops changing.

One round applies the definition at every point:

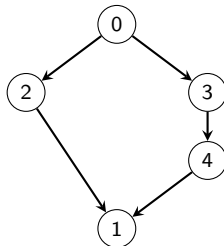
$$F(D)(p) = \{p\} \cup \bigcap_{q \rightarrow p} D(q)$$

$$D_0(\text{entry}) = \{\text{entry}\}, \quad D_0(p) = N \quad (p \neq \text{entry})$$

$$D_{i+1} = F(D_i) \quad \text{stop when } D_{i+1} = D_i$$

- Start from the largest guess: a round only **removes**. Sets are finite, so it **stops**.
- Where it stops, $F(D) = D$: the definition holds at every point. A **fixpoint**.

```
def f(x):
    if 0 < x:
        y = 1
    else:
        y = 2
        y = y * 2
    return y
```

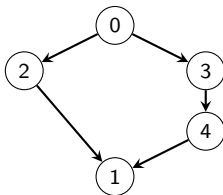


p	D_0	D_1	D_2	$D_3 = D_4$
0	$\{0\}$	$\{0\}$	$\{0\}$	$\{0\}$
2	N	$\{0, 2\}$	$\{0, 2\}$	$\{0, 2\}$
3	N	$\{0, 3\}$	$\{0, 3\}$	$\{0, 3\}$
4	N	N	$\{0, 3, 4\}$	$\{0, 3, 4\}$
1	N	N	$\{0, 1, 2\}$	$\{0, 1\}$

- dom(1) shrinks **twice**: first the short branch arrives, then the long one.

Round by round recomputes *every* point. Only points whose predecessors changed can change:

- 1: $D(p) \leftarrow$ initial for all p
 - 2: $W \leftarrow$ all points but entry
 - 3: **while** $W \neq \emptyset$ **do**
 - 4: pop p from W
 - 5: $\text{new} \leftarrow F(D)(p)$
 - 6: **if** $\text{new} \neq D(p)$ **then**
 - 7: $D(p) \leftarrow \text{new}$
 - 8: push every successor of p onto W , unless already there
- Same fixpoint, less work. Empty worklist $\Rightarrow$ nothing left that can change.
 - W can be a queue, a stack, or anything else: the order changes the work, not the answer.



step	pop	dom(0)	dom(2)	dom(3)	dom(4)	dom(1)	push	W
0		{0}	N	N	N	N		1, 2, 3, 4
1	1	{0}	N	N	N	N		2, 3, 4
2	2	{0}	{0, 2}	N	N	N	1	3, 4, 1
3	3	{0}	{0, 2}	{0, 3}	N	N	4	4, 1
4	4	{0}	{0, 2}	{0, 3}	{0, 3, 4}	N	1	1
5	1	{0}	{0, 2}	{0, 3}	{0, 3, 4}	{0, 1}		$\emptyset$

- Five pops instead of 4×4 . Steps 3 and 4 push a point already in W .

```
class Worklist:
    """A queue of points that holds each point at most once."""

    def __init__(self, points):
        self.queue, self.members = deque(), set()
        for p in points:
            self.push(p)

    def push(self, p: int) -> None:
        if p not in self.members:
            self.queue.append(p)
            self.members.add(p)

    def pop(self) -> int:
        p = self.queue.popleft()
        self.members.remove(p)
        return p

    def __bool__(self) -> bool:
        return bool(self.queue)
```

- A queue plus a set: pushing a point already inside does nothing.

```
def dominators(g: CFG) -> dict[int, set[int]]:
    """dom(p) = {p} union (intersection of dom(q) for every edge q -> p)."""
    # initial: the entry is its own dominator; every other point starts full
    dom = {p: (set(g.nodes) if p != g.entry else {g.entry}) for p in g}
    # worklist: every point but the entry, whose equation is already solved
    work = Worklist(p for p in g if p != g.entry)
    while work:
        p = work.pop()
        # transfer: the equation for p
        new = {p} | set.intersection(*(dom[e.src] for e in g.pred[p]))
        if new != dom[p]:
            dom[p] = new
            for e in g.succ[p]:
                if e.dst != g.entry:
                    work.push(e.dst)
    return dom
```

- Three things: an **initial** value, a **worklist**, a **transfer** function.

An analysis is three things:

- **Initial**: the starting value at every point.
- **Transfer** F : how a point's value is recomputed from its neighbours.
- **Worklist**: the points still to recompute. Pop one, apply F , push its neighbours if it changed. Empty $\Rightarrow$ fixpoint.
- **Every analysis in this course fixes these three and runs the same loop.**

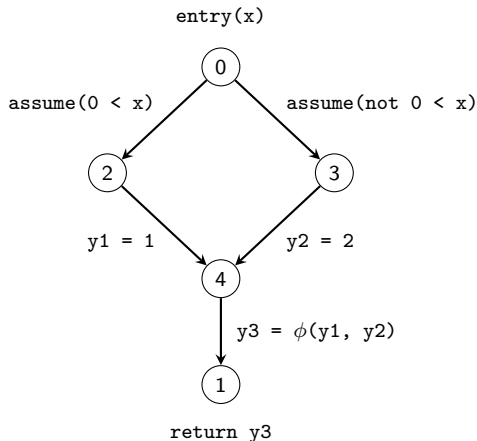
Static Single Assignment: one operator per line, every variable assigned once.

```
x = a + b * c  
x = x + 1
```

```
t1 = b * c  
x1 = a + t1  
x2 = x1 + 1
```

- Every use names **exactly one** assignment.

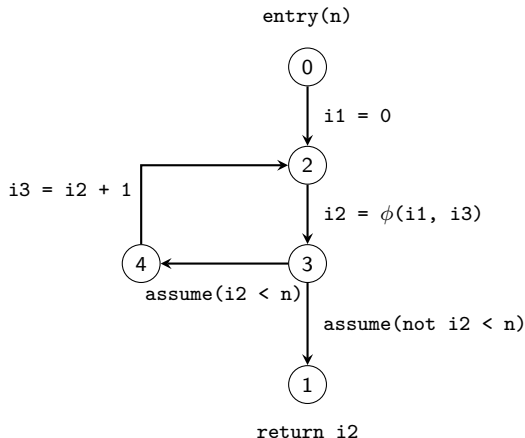
```
def f(x):  
    if 0 < x:  
        y = 1  
    else:  
        y = 2  
    return y
```



- At 4, y could be y_1 or y_2 . Every use must name **one** assignment.
- So the join is followed by a new one: ϕ picks the side control came from.

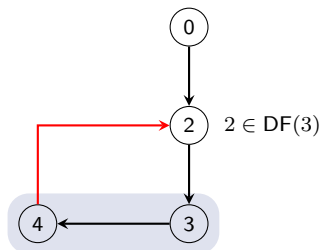
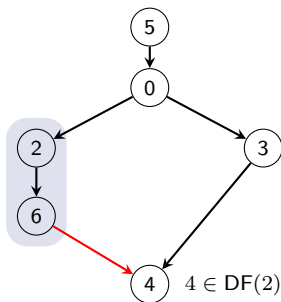
```
def f(n):  
    i = 0  
    while i < n:  
        i = i + 1  
    return i
```

- The loop head 2 is a join too: from 0, and from the back edge.
- Its ϕ edge is taken every time control reaches 2.



Inside the region d dominates, a definition leaving d is the only one. The **dominance frontier** is where that region ends:

$$DF(d) = \{ p \mid \exists q \rightarrow p. \ d \in \text{dom}(q) \text{ and } d \notin \text{dom}(p) \setminus \{p\} \}$$



- The red edge leaves the region: its target is reached *through* d and *around* d . A ϕ goes there.
- LLVM works in SSA. We do not.

1. Control Flow Graphs

- Definition

- Examples

- Semantics

2. Construction

- Translation

- Basic Blocks

- Seeing the Graph

- Running It

3. Dominators and SSA

- Dominators

- Fixpoint Iteration

- SSA Form

- Dataflow Analysis

Jihyeok Park

jihyeok_park@korea.ac.kr

<https://plrg.korea.ac.kr>