

Basic Blocks and CFG

KC Sivaramakrishnan

IIT Madras

Monsoon 2026

Basic blocks revisited

A graph representation of intermediate code.

Basic block properties

- The flow of control can only enter the basic block through the first instruction in the block.
- No jumps into the middle of the block.
- Control leaves the block without halting / branching (except may be the last instruction of the block).

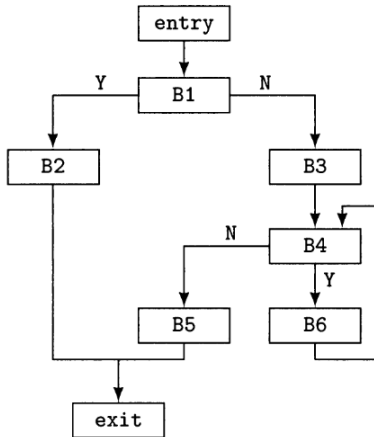
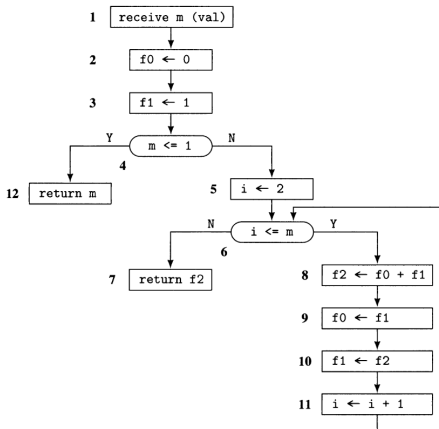
The basic blocks become the nodes of a flow graph, whose edges indicate which blocks can follow which other blocks.

Example

```
unsigned int fib(m)
{
    unsigned int m;
    { unsigned int f0 = 0, f1 = 1, f2, i;
      if (m <= 1) {
        return m;
      }
      else {
        for (i = 2; i <= m; i++) {
          f2 = f0 + f1;
          f0 = f1;
          f1 = f2;
        }
        return f2;
      }
    }
}
```

```
1      receive m (val)
2      f0 ← 0
3      f1 ← 1
4      if m <= 1 goto L3
5      i ← 2
6  L1:  if i <= m goto L2
7      return f2
8  L2:  f2 ← f0 + f1
9      f0 ← f1
10     f1 ← f2
11     i ← i + 1
12     goto L1
13  L3:  return m
```

Example - flow chart and control-flow



Deep dive - Basic block

Basic block definition

- A basic block is a maximal sequence of instructions that can be entered only at the first instruction.
- The basic block can be exited only from the last instruction of the basic block.
- Implication: First instruction can be a) first instruction of a procedure, b) target of a branch, c) instruction following a branch
- First instruction is called the leader of the BB.

How to construct the basic block?

- Identify all the leaders in the program.
- For each leader: include in its basic block all the instructions from the leader to the next leader (next leader not included) or the end of the routine, in sequence.

What about function calls?

- Considered as the last statement in a basic block. Hence, the statement following the call would be a leader.

Example 2

```
for i=1 ... 10 do
  for j=1 ... 10 do
    a[i,j] = 0.0;
  for i=1 ... 10 do
    a[i,i] = 1.0;
```

Three-address code

```
1) i = 1
2) j = 1
3) t1 = 10 * i
4) t2 = t1 + j
5) t3 = 8 * t2
6) t4 = t3 - 88
7) a[t4] = 0.0
8) j = j + 1
9) if j <= 10 goto (3)
10) i = i + 1
11) if i <= 10 goto (2)
12) i = 1
13) t5 = i - 1
14) t6 = 88 * t5
15) a[t6] = 1.0
16) i = i + 1
17) if i <= 10 goto (13)
```

Leader / kind

B1: first instruction

B2: branch target

B3: branch target

B4: follows a branch

B5: follows a branch

B6: branch target

Leaders: 1, 2, 3, 10, 12, 13. Each starts a new basic block.

Example 2: basic blocks and CFG

Three-address code

```
1) i = 1
2) j = 1
3) t1 = 10 * i
4) t2 = t1 + j
5) t3 = 8 * t2
6) t4 = t3 - 88
7) a[t4] = 0.0
8) j = j + 1
9) if j <= 10 goto (3)
10) i = i + 1
11) if i <= 10 goto (2)
12) i = 1
13) t5 = i - 1
14) t6 = 88 * t5
15) a[t6] = 1.0
16) i = i + 1
17) if i <= 10 goto (13)
```

Leader / kind

B1: first instruction

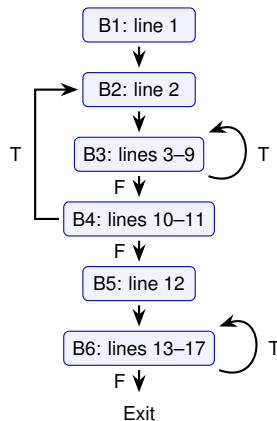
B2: branch target

B3: branch target

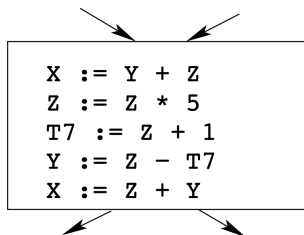
B4: follows a branch

B5: follows a branch

B6: branch target

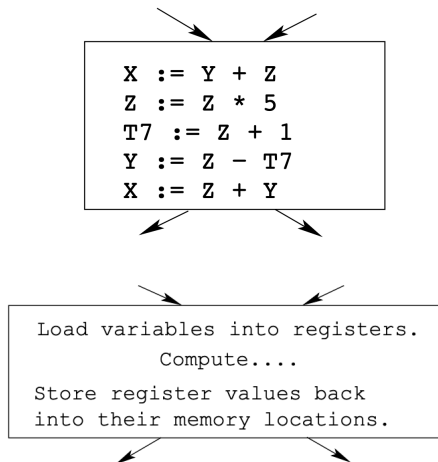


Basic Block Code Generation

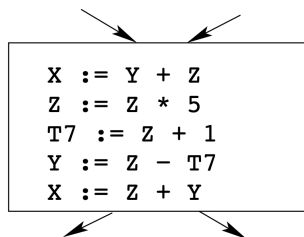


- We want to keep variables in registers for as long as possible, to avoid having to reload them whenever they are needed.
- We don't know which path through the flow-graph has taken us to this basic block $\implies$ We can't assume that any variables are in registers.
- We don't know where we will go from this block $\implies$ Values kept in registers must be stored back into their memory locations before the block is exited.

Basic Block Code Generation

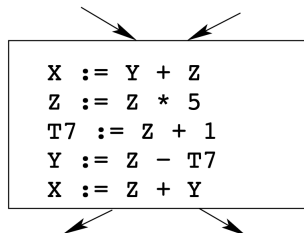


Next use information



- We want to keep variables in registers for as long as possible, to avoid having to reload them whenever they are needed.
- When a variable isn't needed any more we free the register to reuse it for other variables. $\implies$ We must know if a particular value will be used later in the basic block.

Next use information



- If, after computing a value X , we will soon be using the value again, we should keep it in a register. If the value has no further use in the block we can reuse the register

Next use information...

```
(5) x = ...  
    ... (no ref to X) ...  
(14) ... = ... x ...
```

- X is live at (5) because the value computed at (5) is used later in the basic block.
- X 's “next use” at (5) is (14).
- It is a good idea to keep X in a register between (5) and (14).

Next use information...

(12) ... = ... x ...
 ... (no ref to X) ...

(25) x = ...

- X is dead at (12) because its value has no further use in the block.
- Don't keep X in a register after (12).

What does next-use information mean?

At each program point, track the **current value** of each variable v :

- **Live?** Can this value be read later before it is overwritten?
- **Next use:** The next instruction inside this block that reads it; write — if there is none.

Initial state: immediately after the last instruction

Use the block's live-out set if known. Otherwise, conservatively assume all non-temporary variables are live on exit. Variables outside this set start dead.

Set every next-use entry to —: there are no later instructions in this block.

Live with next use — is possible: the value may be needed in a later block, even though this block has no further use of it.

Computing next use: scan backwards

For an assignment $L : dest = src_1 \text{ op } src_2$, visit instructions from last to first.

Before processing L

The working table describes the state **immediately after** L .

- ➊ **Save the after-state.** Attach a copy of the current liveness and next-use entries to instruction L .
- ➋ **Kill the old destination value.** Set $dest$ to (dead, $-$).
The assignment overwrites that value.
- ➌ **Record the reads.** For each variable operand, set its entry to (live, L).
Ignore constants.

The table now describes the state **immediately before** L . Use it as the after-state for the preceding instruction.

Order matters: for $x = x + 1$, kill x first, then mark it live because the right-hand side reads it.

Next use: cross one instruction backwards

Execution order

```
(1) x = y + z
(2) z = x * 5
(3) y = z - 7
(4) x = z + y
```

Assume x , y , and z are all live on exit.

Start after instruction 4

	x	y	z
Live?	yes	yes	yes
Next use	—	—	—

Save this as instruction 4's **after-state**.

- 1 Kill old x : instruction 4 overwrites it.
- 2 Mark z and y live: instruction 4 reads them.

Result: before instruction 4

	x	y	z
Live?	no	yes	yes
Next use	—	4	4

Before instruction 4 = after instruction 3. Continue scanning upwards.

Next use: the complete backward scan

Each entry is **(live?, next use)**; – means no next use in this block.

Program point	x	y	z	Code (execution order)
After 4: block exit	(yes, –)	(yes, –)	(yes, –)	(1) $x = y + z$
Before 4 = after 3	(no, –)	(yes, 4)	(yes, 4)	(2) $z = x * 5$
Before 3 = after 2	(no, –)	(no, –)	(yes, 3)	(3) $y = z - 7$
Before 2 = after 1	(yes, 2)	(no, –)	(no, –)	(4) $x = z + y$
Before 1: block entry	(no, –)	(yes, 1)	(yes, 1)	Analyse from 4 up to 1; reveal one state per step.

Cross (1) $x = y + z$: kill old x ; record reads of y and z at 1.

The value of x produced at 1 is next used at 2. Its old value is dead before 1.

CFG - Control flow graph

Definition:

- A rooted directed graph $G = (N, E)$, where N is given by the set of basic blocks + two special BBs: `entry` and `exit`.
 - `entry` node has no predecessor.
 - `exit` node has no successor.
- An edge connects two basic blocks b_1 and b_2 if control can pass from b_1 to b_2 .
- An edge from `entry` node to the initial basic block.
- From each final basic block (with no successors) to `exit` BB.

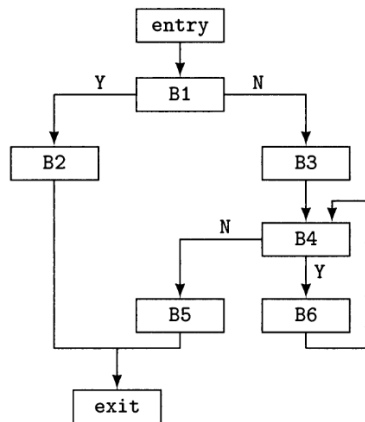
- `successor` and `predecessor` – defined in a natural way.
- A basic block is called branch node - if it has more than one successor.
- `join node` – has more than one predecessor.
- For each basic block b :

$$Succ(b) = \{n \in N \mid \exists e \in E \text{ such that } e = b \rightarrow n\}$$

$$Pred(b) = \{n \in N \mid \exists e \in E \text{ such that } e = n \rightarrow b\}$$

- A region is a strongly connected subgraph of a flow-graph.

CFG Analysis: Finding Loops



- Identifying loops in a CFG is important for optimizations.
- We can identify loops by using dominators
 - a node *A* in the flowgraph dominates a node *B* if every path from `entry` node to *B* includes *A*.
- back edge: An edge in the flow graph, whose destination dominates its source (example - edge from **B6** to **B4**).
- Each back edge identifies a natural loop. Its header dominates every node in that loop.

Dominators

Dominance relation:

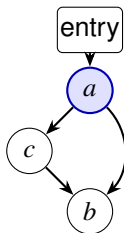
Consider the nodes reachable from `entry` in a CFG.

- Node d dominates node i (written $d \text{ dom } i$), if every path in the CFG from `entry` to i includes d .
- Reflexive: $a \text{ dom } a$
- Antisymmetric: $a \text{ dom } b, b \text{ dom } a \Rightarrow a = b$
- Transitive: if $a \text{ dom } b$ and $b \text{ dom } c$, then $a \text{ dom } c$
- We write $\text{dom}(a)$ to denote the dominators of a .

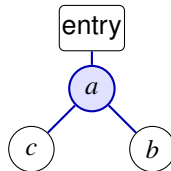
Dominators: what about a predecessor?

If $a \text{ dom } b$, $a \neq b$, and $c \in \text{Pred}(b)$, must $a \text{ dom } c$?

Control-flow graph



Dominator tree



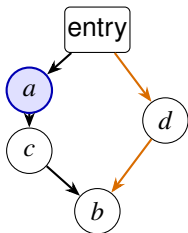
A parent is an immediate dominator.
Ancestors (and the node itself) are its dominators.

Yes: $a \text{ dom } c$. If a path to c avoided a , appending $c \rightarrow b$ would give a path to b avoiding a , contradicting $a \text{ dom } b$.

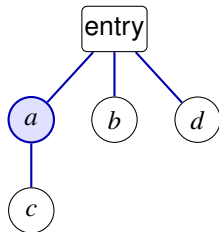
Dominators: is one predecessor enough?

If $c \in \text{Pred}(b)$ and $a \text{ dom } c$, must $a \text{ dom } b$?

Control-flow graph



Dominator tree



No. The orange path $\text{entry} \rightarrow d \rightarrow b$ avoids a .

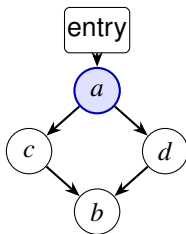
$\text{dom}(c) = \{\text{entry}, a, c\}$, but $\text{dom}(b) = \{\text{entry}, b\}$.

Dominating one predecessor does not cover the other incoming paths.

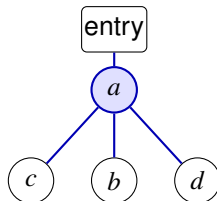
Dominators: what if all predecessors agree?

For $b \neq \text{entry}$, if a dominates every predecessor of b , must $a \text{ dom } b$?

Control-flow graph



Dominator tree



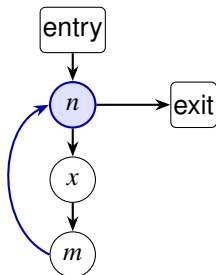
Yes. Every path to b enters through a predecessor.

For this example:

$$\begin{aligned} \text{dom}(c) &= \{\text{entry}, a, c\}, & \text{dom}(d) &= \{\text{entry}, a, d\}, \\ \text{dom}(c) \cap \text{dom}(d) &= \{\text{entry}, a\}, & \text{dom}(b) &= \{\text{entry}, a, b\}. \end{aligned}$$

A back edge returns to a dominating header

An edge $m \rightarrow n$ is a **back edge** when $n \text{ dom } m$.



Is the blue edge $m \rightarrow n$ a back edge?

- Every path from entry to m passes through n .
- Therefore $n \text{ dom } m$: **yes**.
- n is the **header**; m is the source of the back edge.

The repeated execution is

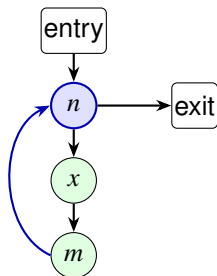
$$n \rightarrow x \rightarrow m \rightarrow n \rightarrow \dots$$

Control can leave through $n \rightarrow \text{exit}$.

Check dominance, not the direction of the arrow on the page.

Finding the natural loop: walk backwards

For the back edge $m \rightarrow n$, include n and all nodes that can reach m **without passing through n** .



Follow predecessor edges backwards:

- 1 Start with $\{n, m\}$.
- 2 The predecessor of m is x : add x .
- 3 The predecessor of x is n : stop at the header.

$$L(m \rightarrow n) = \{n, x, m\}$$

Do not explore predecessors of n .

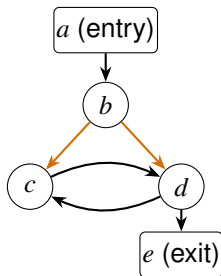
Why exclude entry and exit?

Entry can reach m only through n ; exit cannot reach m at all.

Even though n dominates exit, exit is outside the natural loop.

A cycle with two entrances has no single header

Does the cycle $c \rightarrow d \rightarrow c$ form a natural loop?



Test $d \rightarrow c$: does c dominate d ?

No: $a \rightarrow b \rightarrow d$ avoids c .

Test $c \rightarrow d$: does d dominate c ?

No: $a \rightarrow b \rightarrow c$ avoids d .

Neither edge is a back edge.

The cycle can be entered at either c or d .
Neither node is a dominating header.

A cycle need not be a natural loop. This is an example of irreducible control flow.

How goto creates two entrances

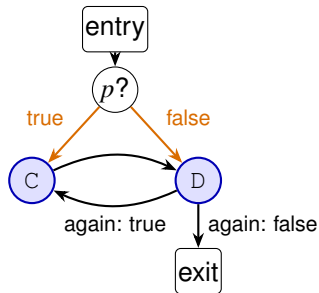
Which blocks form a cycle? Can either be its dominating header?

Code

```
if (p) goto C;  
else goto D;
```

```
C:  
    work_c();  
D:  
    work_d();  
    if (again()) goto C;
```

Control-flow graph



Execution falls through from C to D.

The cycle is $C \rightarrow D \rightarrow C$.

Entering at D bypasses C; entering at C bypasses D.

Neither dominates the other: **this control flow is irreducible.**