Basic Blocks and CFG

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).

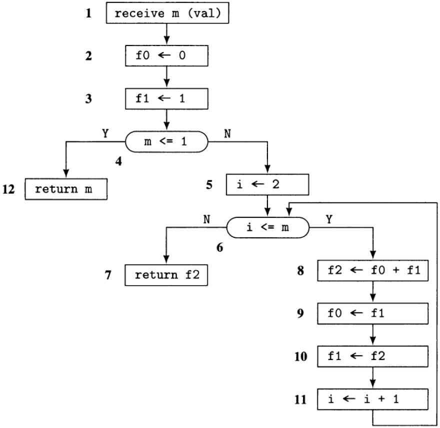
The basic blocks become the nodes of a flow graph, whose edges indicate which blocks can follow which other blocks.

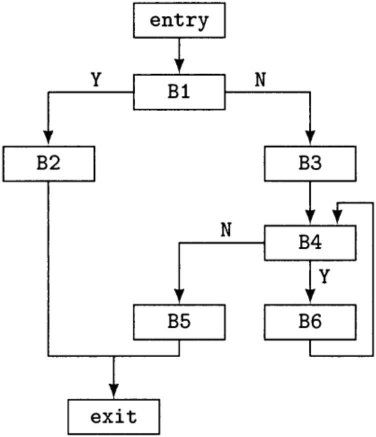
Example

```
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;
   }
}
```

```
     receive m (val)
     f0 ← 0
     f1 ← 1
     if m <= 1 goto L3
     i ← 2
L1:  if i <= m goto L2
     return f2
L2:  f2 ← f0 + f1
     f0 ← f1
     f1 ← f2
     i ← i + 1
     goto L1
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.

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.

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;
```

```
 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)
```

Next use information

- Goal: when the value of a variable will be <u>used</u> next.

```
L1: x = ...
...
L2: y = x
```

  Statement `L2` uses the value of `x` computed (<u>defined</u>) at `L1`. We also say `x` is live at `L2`.
- For each three-address statement $x = y + z$, what is the next use of x, y, and z?
- We want to compute next use information within a basic block.
- Many uses : Register Allocation, Dead code elimination, etc.

Algorithm to compute next use information

Input: A basic block B of three-address statements. We assume that the symbol table initially shows all non-temporary variables in B as being live on exit.

Output: At each statement `L : x = y op z` in B, we attach to L the liveness and next-use information of `x, y,` and `z`.

begin

List *lst* = Starting at last statement in B and list of instructions obtained by scanning backwards to the beginning of B;

foreach statement `L: x = y op z` $\in lst$ **do**

Attach to statement `L` the information currently found in the symbol table regarding the next use and liveness of `x, y`, and `z`;

In the symbol table, set `x` to "not live" and "no next use.";

In the symbol table, set `y` and `z` to "live" and the next uses of `y` and `z` to `L` ;

end

end

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.

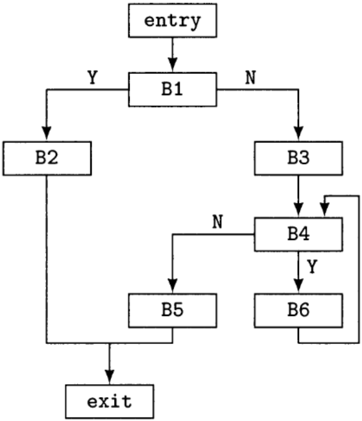
CFG continued

- `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:

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

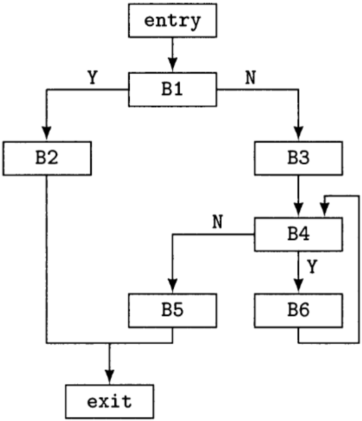
- A region is a strongly connected subgraph of a flow-graph.

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.

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`.
- A loop consists of all nodes dominated by its entry node (head of the back edge) and having exactly one back edge in it.

Dominators

Dominance relation:

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

Dominators

Dominance relation:

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

Questions:

- If $a\ dom\ b$, $a \neq_{\beta}$ and $c \in Pred(b)$, what can be say about a and c?

Dominators

Dominance relation:

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

Questions:

- If $a\ dom\ b$, $a \neq_\beta$ and $c \in Pred(b)$, what can be say about a and c? $a\ dom\ c$.

Dominators

Dominance relation:

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

Questions:

- If $a\ dom\ b$, $a \neq_{\beta}$ and $c \in Pred(b)$, what can be say about a and c? $a\ dom\ c$.
- If $c \in Pred(b)$ and $a\ dom\ c$, what can be say about a and b?

Dominators

Dominance relation:

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

Questions:

- If $a\ dom\ b$, $a \neq_{\beta}$ and $c \in Pred(b)$, what can be say about a and c? $a\ dom\ c$.
- If $c \in Pred(b)$ and $a\ dom\ c$, what can be say about a and b? Nothing, since $a\ dom\ b$ may not hold.

Dominators

Dominance relation:

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

Questions:

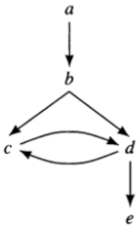
- If $a\ dom\ b$, $a \neq_{\beta}$ and $c \in Pred(b)$, what can be say about a and c? $a\ dom\ c$.
- If $c \in Pred(b)$ and $a\ dom\ c$, what can be say about a and b? Nothing, since $a\ dom\ b$ may not hold.
  - If $a \in \bigcap_{c \in Pred(b)} dom(c)$, then $a\ dom\ b$.

Identifying loops

- Back edge: an edge in the flowgraph, whose destination dominates its source.

Identifying loops

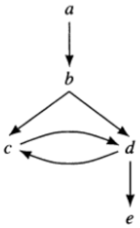
- Back edge: an edge in the flowgraph, whose destination dominates its source.(Counter example)



Has a loop, but no back edge – hence not a natural loop.

Identifying loops

- Back edge: an edge in the flowgraph, whose destination dominates its source.(Counter example)



Has a loop, but no back edge – hence not a natural loop.

- Given a back edge $m \to n$, the natural loop of $m \to n$ is
  1. the subgraph consisting of the set of nodes containing n and all the nodes from which m can be reached in the flowgraph without passing through n, and
  2. Node n is called the loop header.

Instruction Selection

- Performing the actual translation targeting the Instruction Set Architecture of the underlying machine.
  - More difficult for CISC machines which are rich in addressing modes, resulting in many possible translations.
- We can design SDD schemes for generating machine code, just like what we did for 3-address IR.
  - The underlying grammar would be for 3-address code (or other similar IRs).

Instruction Selection

- Performing the actual translation targeting the Instruction Set Architecture of the underlying machine.
  - More difficult for CISC machines which are rich in addressing modes, resulting in many possible translations.
- We can design SDD schemes for generating machine code, just like what we did for 3-address IR.
  - The underlying grammar would be for 3-address code (or other similar IRs).
- There are efficient schemes for translating arithmetic expressions guaranteeing the use of minimum number of registers for storing temporaries.
  - Can also take into account different addressing modes available in CISC machines, guaranteeing minimum execution time.
  - Only applicable for translation of an individual (albeit arbitrarily long) arithmetic expression. For global optimization, we have to rely on register allocation.