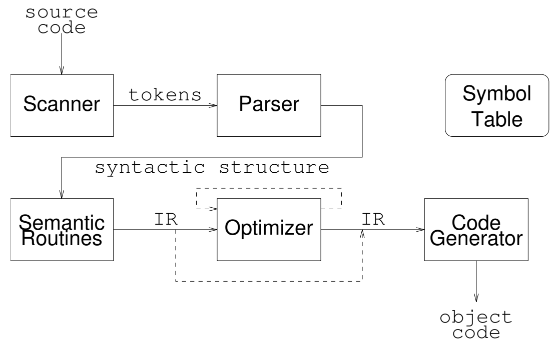
CS6013 - Modern Compilers: Theory and Practise

Control flow analysis

V. Krishna Nandivada

IIT Madras

Control flow analysis



- Code optimization requires that the compiler has a global "understanding" of how programs use the available resources.
- It has to understand how the control flows (control-flow analysis) in the program and how the data is manipulated (data-flow analysis)
- Control-flow analysis: flow of control within each procedure.
- Data-flow analysis: how the data is manipulated in the program.

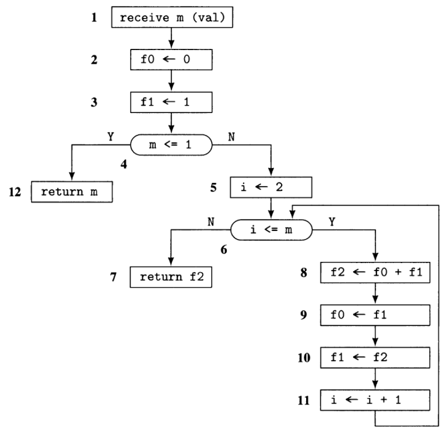
Example

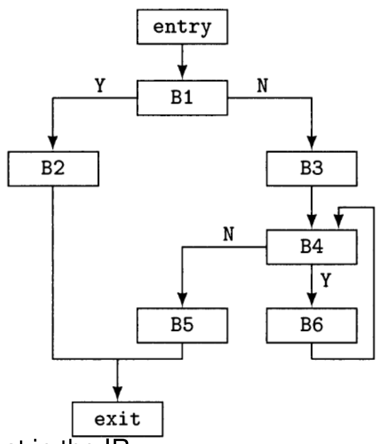
```
unsigned int fib(m)
   unsigned int m;
{  unsigned int f0 = 0, f1 = 1, f2, i;
   if (m <= 1) {
      return m;
   }
   else {
      for (i = 2; i <= m; i++) {
         f2 = f0 + f1;
         f0 = f1;
         f1 = f2;
      }
      return f2;
   }
```

```
      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
```

- IR for the C code (in a format described in Muchnick book)
- `receive` specifies the reception of a parameter and the parameter-passing discipline (by-value, by-result, value-result, reference). Why do we want to have an explicit receive instruction?– Gives a point of definition for the args.
- What is the control structure? Obvious?

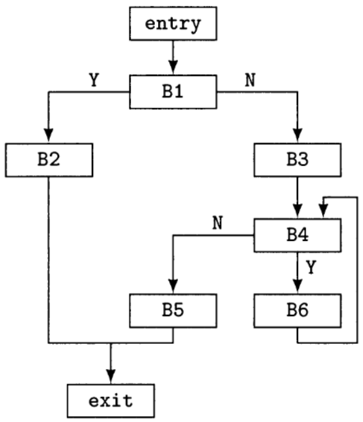
Example - flow chart and control-flow





- The high-level abstractions might be lost in the IR.
- Control-flow analysis can expose control structures not obvious in the high level code. Possible? Loops constructed from `if` and `goto`
- A basic block is informally a straight-line sequence of code that can be entered only at the beginning and exited only at the end.

Basic blocks - what do we get?



- `entry` and `exit` are added for reasons to be explained later.
- 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.
  - This relations is antisymmetric, reflexive, and transitive.
- back edge: An edge in the flow graph, whose head dominates its tail (example - edge from `B6` to `B4`.
- A loop consists of subset of nodes dominated by its entry node (head of the back edge) and having exactly one back edge in it.

Deep dive - Basic block

Basic block definition

- A basic block is a maximal sequence of instructions that can be entered only at the first of them
- The basic block can be exited only from the last of the instructions of the basic block.
- Implication:First instruction can be a) entry point of a routine,b) item target of a branch, c) item instruction following a branch or a return.
- 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?

- In most cases it is not considered as a branch+return. Why?
- Problem with setjmp() and longjmp()? [self-study]

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`.
- And edge connects two basic blocks b_1 and b_2 if control can pass from b_1 to b_2.
- An edge(s) from `entry` node to the initial basic block(s?)
- From each final basic blocks (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:

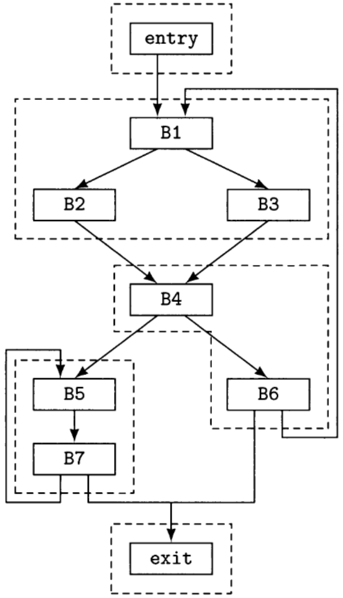
$$Succ(b) = \{n \in N | \exists e \in E \text{ such that } e = b \to n\}$$
$$Pred(b) = \{n \in N | \exists e \in E \text{ such that } e = n \to b\}$$

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

Extended basic block

Extended basic block

- a maximal sequence of instructions beginning with a leader that contains no join nodes other than its first node.
- Has a single entry, but possible multiple exit points.
- Some optimizations are more effective on extended basic blocks.
- Why EBBs? Extending "local" optimizations to EBBs is straightforward.
- How to build an EBB, for a given basic block?



Dominators and Postdominators

- Goal: To determine loops in the flowgraph.

Dominance relation:

- Node d dominates node i (written $d\ dom\ i$), if every possible execution path from `entry` to i includes d.
- This relations is antisymmetric ($a\ dom\ b$, $b\ dom\ a \Rightarrow a = b$), reflexive ($a\ dom\ a$), and transitive (if $a\ dom\ b$ and $b\ dom\ c$, then $a\ dom\ c$.
- We write $dom(a)$ to denote the dominators of a.

Immediate dominance:

- A subrelation of dominance.
- For $a \neq b$, we say $a\ idom\ b$ iff $a\ dom\ b$ and there does not exist a node c such that $c \neq a$ and $c \neq b$, for which $a\ dom\ c$ and $c\ dom\ b$.
- We write $idom(a)$ to denote the immediate dominator of a – note it is unique.

Strict dominance:

- $d\ sdom\ i$, if d dominates i and $d \neq i$.

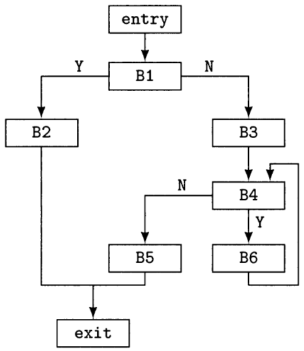
Post dominance:

- $p\ pdom\ i$, if every possible execution path from i to `exit` includes p.
- Opposite of dominance ($i\ dom\ p$), in the reversed CFG (edges reversed, `entry` and `exit` exchanged).

Computing all the dominators

```
procedure Dom_Comp(N,Pred,r) returns Node ⟶ set of Node
   N: in set of Node
   Pred: in Node ⟶ set of Node
   r: in Node
begin
   D, T: set of Node
   n, p: Node
   change := true: boolean
   Domin: Node ⟶ set of Node
   Domin(r) := {r}
   for each n ∈ N - {r} do
      Domin(n) := N
   od
   repeat
      change := false
*     for each n ∈ N - {r} do
         T := N
         for each p ∈ Pred(n) do
            T ∩= Domin(p)
         od
         D := {n} ∪ T
         if D ≠ Domin(n) then
            change := true
            Domin(n) := D
         fi
      od
   until !change
   return Domin
end    || Dom_Comp
```

* Order makes the difference.



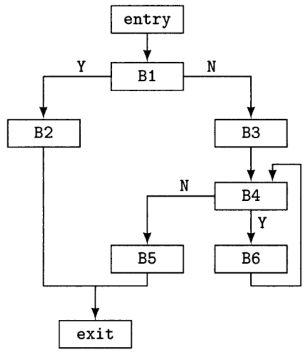
Compute the dominators.

i	Domin(i)
entry	{entry}
B1	{entry,B1}
B2	{entry,B1,B2}
B3	{entry,B1,B3}
B4	{entry,B1,B3,B4}
B5	{entry,B1,B3,B4,B5}
B6	{entry,B1,B3,B4,B6}
exit	{entry,B1,exit}

Computing all the immediate dominators

```
procedure Idom_Comp(N,Domin,r) returns Node ⟶ Node
   N: in set of Node
   Domin: in Node ⟶ set of Node
   r: in Node
begin
   n, s, t: Node
   Tmp: Node ⟶ set of Node
   Idom: Node ⟶ Node
   for each n ∈ N do
      Tmp(n) := Domin(n) - {n}
   od
*  for each n ∈ N - {r} do
      for each s ∈ Tmp(n) do
         for each t ∈ Tmp(n) - {s} do
            if t ∈ Tmp(s) then
               Tmp(n) -= {t}
            fi
         od
      od
   od
   for each n ∈ N - {r} do
      Idom(n) := ◆Tmp(n)
   od
   return Idom
end    || Idom_Comp
```

* Order makes the difference.

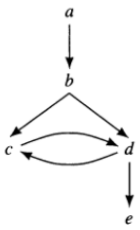


immediate dominators.

i	Tmp(i)
entry	∅
B1	{entry}
B2	{B1}
B3	{B1}
B4	{B3}
B5	{B4}
B6	{B4}
exit	{B1}

Identifying loops

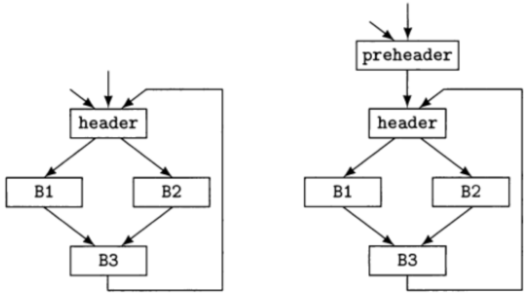
- Back edge: an edge in the flowgraph, whose head dominates its tail.(Counter example)



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. the edge set connecting all the nodes in its node set.
  3. Node n is called the loop header.

Loops (contd.)

- preheader: a new (initially empty) block is placed just before the header of a loop
- all the edges that previously went to the header from outside the loop now go to the preheader, and there is a single new edge from the preheader to the header.



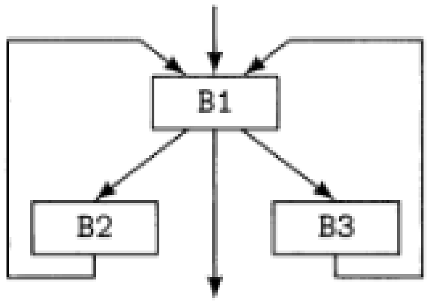
- Adv: helps optimizations that move code from inside a loop to just before its header – preheader guarantees that such a place is available – the code will be put in the pre-header

Algorithm to compute natural loops

```
procedure Nat_Loop(m,n,Pred) returns set of Node
   m, n: in Node
   Pred: in Node ⟶ set of Node
begin
   Loop: set of Node
   Stack: sequence of Node
   p, q: Node
   Stack := []
   Loop := {m,n}
   if m ≠ n then
      Stack ⊕= [m]
   fi
   while Stack ≠ [] do
      || add predecessors of m that are not predecessors of n
      || to the set of nodes in the loop; since n dominates m,
      || this only adds nodes in the loop
      p := Stack↓-1
      Stack ⊖= -1
      for each q ∈ Pred(p) do
         if q ∉ Loop then
            Loop ∪= {q}
            Stack ⊕= [q]
         fi
      od
   od
   return Loop
end    || Nat_Loop
```


Loops (contd.)

- Unless two natural loops have the same header – they are either disjoint or one is nested inside other.
- What about the other way? Given two loops with the same header – can we guarantees that either a) one is nested inside other, or b) they constitute the same loop?



Two natural Loops with same header (contd.)

```
    i = 1;
B1: if (i >= 100)
        goto b4;
    else if ((i % 10) == 0)
        goto B3;
    else
B2:     ...
        i++;
        goto B1;
B3: ...
    i++;
    goto B1;
B4: ...
```

```
B1: if (i < j)
       goto B2;
    else if (i > j)
       goto B3;
    else goto B4;
B2: ...
    i++;
    goto B1;
B3: ...
    i--;
    goto B1;
B4: ...
```

- Can be fixed – disallow if-then-else?
- What about loops with multiple entry points?
- A loop can be most generally described by a strongly connected component of a flowgraph.
- Self reading – Algorithm to compute SCCs.

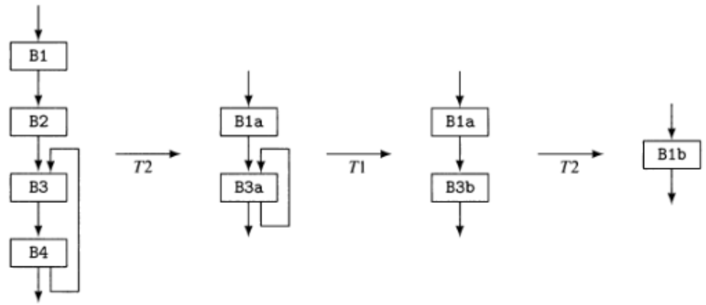
Reducibility

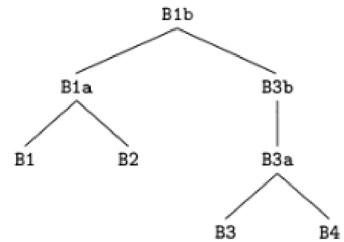
- "Reducibility" – a property of the flowgraphs.
- A reducible transformation is one that collapses subgraphs into single nodes (and hence "reduces" the graph).
- A flow graph is reducible if applying a sequence of such transformations ultimately reduces it to a single node.
- A flow graph $G = (N, E)$ is reducible (or well structured) iff
  - E can be partitioned into disjoint sets E_F – set of forward edges; and E_B – set of backward edges; such that
  - (N, E_F) forms a DAG in which every node can be reached from the entry node.
  - E_B has all the back edges.
- A flowgraph is reducible if all the loops in it are natural loops (characterized by their back edges) and vice versa.
- Implication: A reducible flowgraph has no jumps into the middle of the loops – makes the analysis easy.
- Read yourself – irreducible flow graphs.

Interval analysis

- An alternative approach to do control flow analysis.
- Overall three steps:
  - Divide the flowgraph into "regions" of varisous sorts (depending on the particular approach),
  - consolidating each region into a new node (called an abstract node – as it abstracts away what's inside the node), and
  - replace "entering" and "leaving" edges.
- Resulting graph is called a abstract flowgraph.
- The above transformations can be applied in sequence or in parallel.
- Each abstract node corresponds to a subgraph.
- The result of applying such transformations on a abstract flowgraph is also called control tree.

Example T1-T2 analysis





Control tree

Control tree:

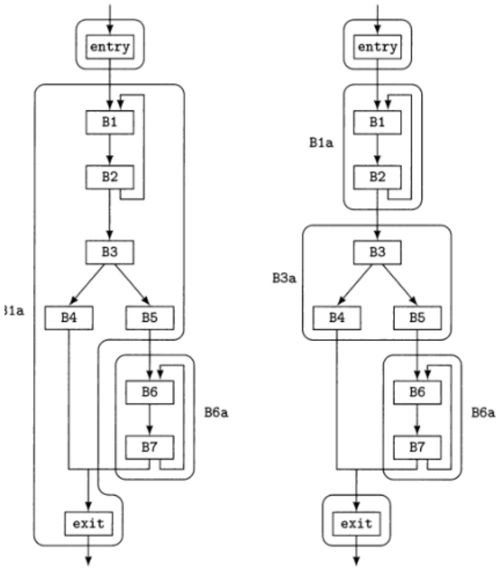
- Root of the control tree is an abstract graph representing the orignal graph.
- The leaves are individual basic blocks.
- The nodes between the root and the leaves are the abstract nodes representing regions of the flowgraph.
- The edges represent the relationship between each abstract node and the node regions.

Interval analysis

Interval analysis

- Ignores irreducible regions.
- Uses maximal intervals: A maximal interval, with a leader h is the single entry subgraph with entry h, may contain a natural loop and some acyclic structure dangling from its exits.
- minimal interval: A minimal interval is defined to be
  1. a natural loop.
  2. a maximal acyclic subgraph.
  3. a minimal irreducible region.
- It is used to identify loops in the flowgraph.

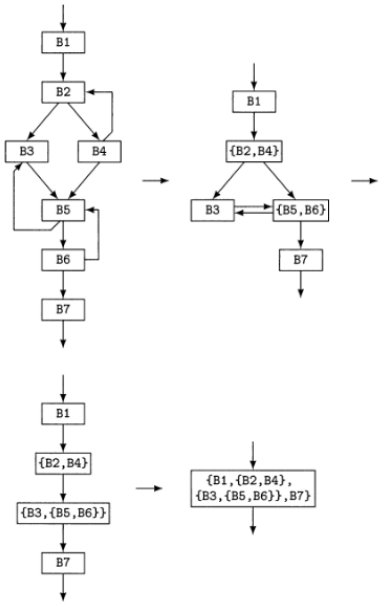
Example: Maximal and minimal intervals



Steps to perform interval analysis

- Perform a postorder traversal of the flowgraph – look for loop headers, and headers of improper regions.
- For each loop header found, construct its natural loop; and then reduce it (T1).
- For each improper region – construct a minimal SCC and reduce.
- For the `entry` node and the immediate descendent of a node in a natural loop, construct a maximal acyclic graph with that node as its root; may reduce it (T2) if it has more than one node in it.
- Iterate till it terminates.

Example: Interval analysis



Approaches to Control flow Analysis

Two main approaches to control-flow analysis of single routines.

- Both start by determining the basic blocks that make up the routine.
- Construct the control-flowgraph.

First approach:

- Use dominators to discover loops; to be used in later optimizations.
- Sufficient for many optimizations (ones that do iterative data-flow analysis, or ones that work on individual loops only).

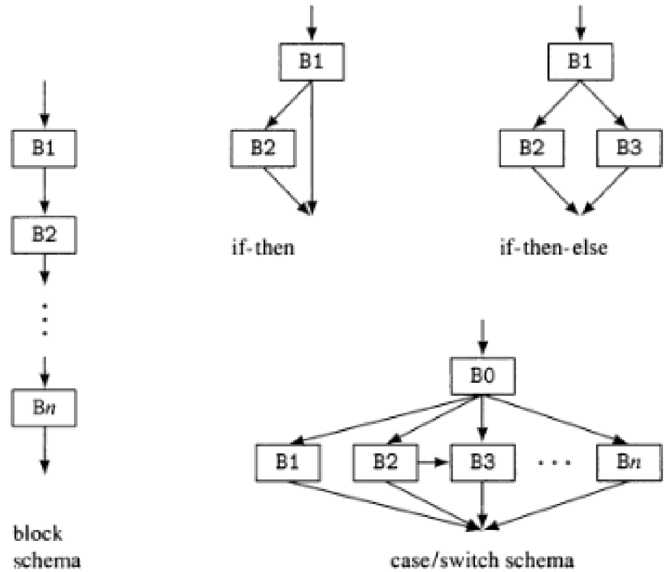
Second approach (interval analysis):

- Analyzes the overall structure of the routine.
- Decomposes the routine into nested regions - called intervals.
- The resulting nesting structure is called a control tree.
- A sophisticated variety of interval analysis is called structural analysis.

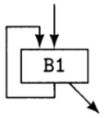
Structural analysis

- A more refined form of interval analysis.
- Differs from basic interval analysis in that it identifies many types of control structures than just loops.
- Each such structure is turned into a region and provides a basis for doing efficient data-flow analysis on each of the different regions.
- Output - a control tree.
  Typically larger than that we find for interval analysis.
  But the individual regions are simpler and simpler.
- Region – has exactly one entry point –
  How to include an irreducible or improper region? (coming soon).

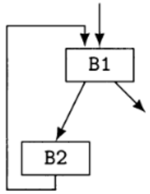
Examples of (Acyclic) regions



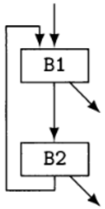
Examples of (Cyclic) regions



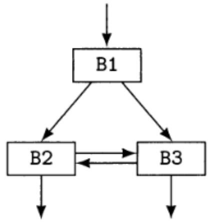
self loop



while loop



natural loop schema

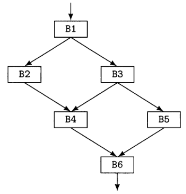


improper interval schema

Beyond the discussed regions

- The patterns for the control-flow constructs are determined by the syntax and semantics of the language.
- The presented patterns are schematic in nature.
  - For example - switch case may or not have a free fall to the next branch.
  - “natural loop” talks about loops that neither a self or a while loop.
- Will the presented patterns cover all types of intervals seen in practise?
- Another type of pattern is called a proper interval – an arbitrary acyclic structure; contains no cycles and cannot be reduced to any of the simple acyclic cases.

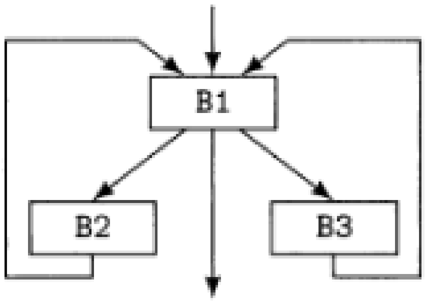
Example:



Structural analysis - computation

- Process is similar to that of interval analysis – except that there are more patterns.

Example:

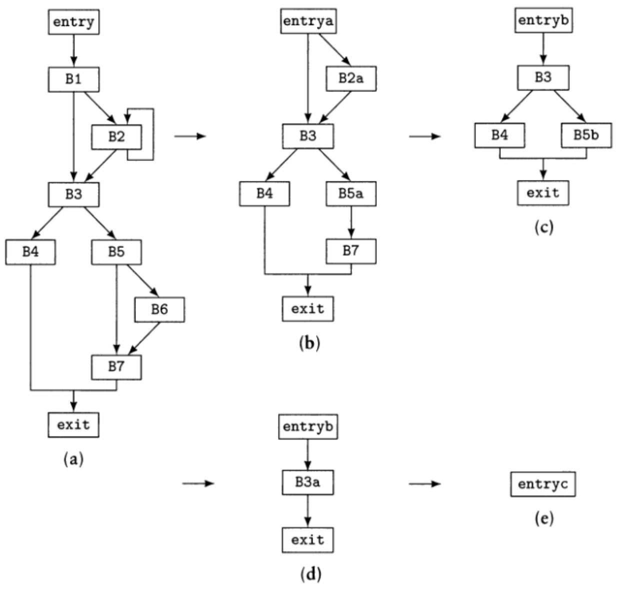


Structural analysis - algorithm

- Construct a depth-first spanning tree for the flowgraph.
- Examine the flowgraphs’s nodes in postrder, for instances of the various regions.
  - Form abstract nodes for each region.
  - Collapse the connecting edges.
- Build the control tree in the proces.

Self reading: how to identify these intervals?

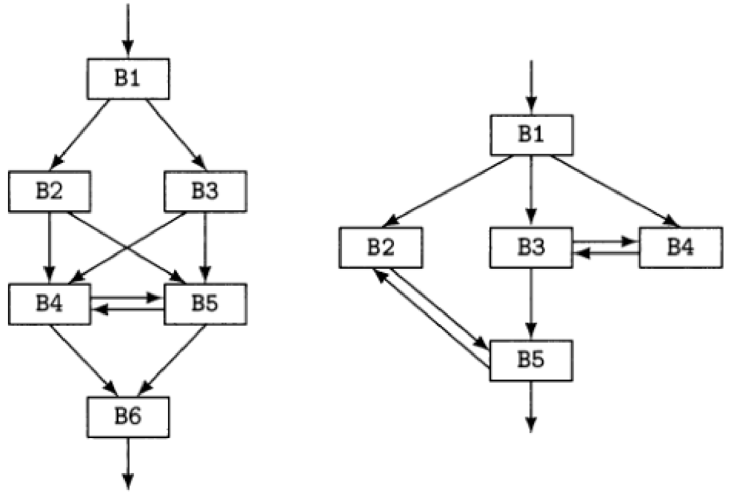
Example - structural analysis



How to detect Improper regions

- Add the lowest common dominator (`ncd`) of the set of entry points (`I`) for the multiple-entry cycle.
- Find a node that is reachable from `ncd`. Say n.
- If there exists a path from n to any element of I – add n.

Improper regions and Importance of order



1. interval = (improper) {B1, B2, B3, B4, B5} {B6}
2. Say, we choose B3 before B2: a) {B1, B3, B4}, {B2, B5}; Otherwise, b) {B1, B2, B3, B4, B5}

Who uses what?

We studied two techniques: dominators based and interval analysis based. Which is used in practise?

- Most optimizing compilers dominators and iterative data flow analysis – its easy/quick to write.

But

- The interval-based approaches are faster.
- The interval-based approaches help easy update of computed data (don't need to recompute from scratch).

Uses of Structural analysis

- Structural control flow analysis to the aid of Constant propagation.
- Control flow optimizations

Control flow optimization

- Goal: produce longer basic blocks. What is it good for?
  - Can help increase instruction-level parallelism.
- Reduce code size.

Unreachable code elimination

Straightening

- Fuses basic blocks if the predecessor has only one successor and the successor has only one predecessor.

If simplification

- Simplify if conditions:
  - Say the `then` part is empty – reverse the condition.
  - If both `then` and `else` are empty – remove both and keep the condition. Why?
  - 'Predicate' evaluates to a constant – throw away `then` or `else` part. What about the predicate evaluation?
  - Nested if-then-else statements where the outer predicate ⇒ inner predicate.

Loop Inversion

- Transforming a `while` loop to a `do-while` or `repeat-until` loop.
- Adv:
  - Only one jump to end the loop.
  - Gives a guarantee that the loop will be executed for sure.

```
x = 3;
while (cond) {
     S1;
     x = 4;
}
// Q: Is x a constant here?
```


Examples - loop inversion

- Where the loop condition is known to hold for the first iteration.
- Where the loop condition is not guaranteed to hold for the first iteration.

Closing remarks

What have we done?

- Control flow analysis (identifying loops and interval analysis).
- Control flow optimizations.

To read

- Muchnick - Ch 7, (parts of) Ch 18.

Next:

- Data flow analysis