

3.36pt

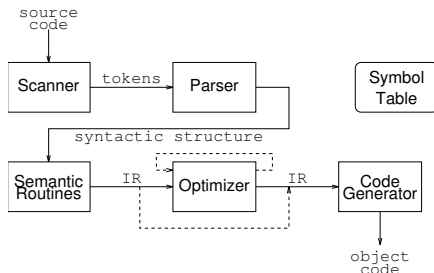


# Challenges in the back end

- The input to the backend (What?).
- The target program – instruction set, constraints, relocatable or not (adv/disadv?), machine code or assembly?
- Instruction selection (undecidable): maps groups of IR instructions to one or more machine instructions. Why not say each IR instruction maps to one more more machine level instructions?
  - Easy, if we don't care about the efficiency.
  - Choices may be involved (add / inc); may involve understanding of the context in which the instruction appears.
- Register Allocation (NP-complete): Intermediate code has temporaries. Need to translate them to registers (fastest storage).
  - Finite number of registers.
  - If cannot allocate on registers, store in the memory – will be expensive.
  - Sub problems: Register allocation, register assignment, spill location, coalescing. All NP-complete.
- Evaluation order: Order of evaluation of instructions may impact the code efficiency (e.g., distance between load and use).



# Compiler 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: understanding how the data is manipulated the program.



# Basic blocks

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

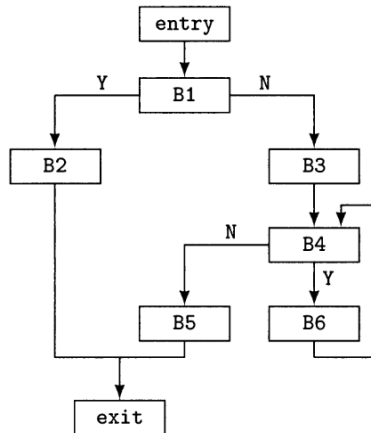
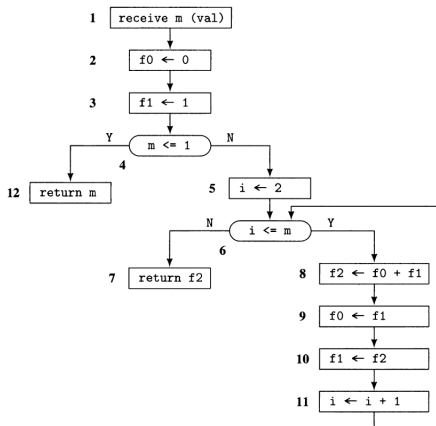
```
unsigned int fib(m)
    unsigned int m;
{ unsigned int f0 = 0, f1 = 1, f2, i;
  if (m <= 1) {
    return m;
  }
  else {
    for (i = 2; i <= m; i++) {
      f2 = f0 + f1;
      f0 = f1;
      f1 = f2;
    }
    return f2;
  }
}
```

```
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
```

- `receive` specifies the reception of a parameter. Why do we want to have an explicit receive instruction? To specify the parameter name and the parameter-passing discipline (by-value, by-result, value-result, reference); also gives a definition point.
- 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`



# 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) first instruction of a routine, b) target of a branch, c) 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 ]



## 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)
```





# 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`.
- An 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  $b$ :

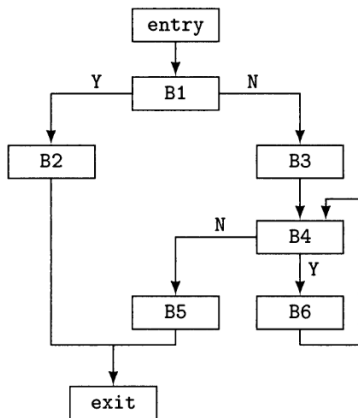
$$Succ(b) = \{n \in N \mid \exists e \in E \text{ such that } e = b \rightarrow n\}$$

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

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



# 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 all nodes dominated by its entry node (head of the back edge) and having exactly one back edge in it.



# Dominators and Postdominators

- Goal: To determine loops in the flowgraph.

Dominance relation:

- Node  $d$  dominates node  $i$  (written  $d \text{ dom } i$ ), if every possible execution path from `entry` to  $i$  includes  $d$ .
- This relation is antisymmetric ( $a \text{ dom } b, b \text{ dom } a \Rightarrow a = b$ ), reflexive ( $a \text{ dom } a$ ), and 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  $a$ .

Immediate dominance:

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

Strict dominance:

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

Post dominance:

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



# Computing all the dominators

procedure Dom\_Comp(N,Pred,r) returns Node  $\rightarrow$  set of Node

N: in set of Node

Pred: in Node  $\rightarrow$  set of Node

r: in Node

begin

D, T: set of Node

n, p: Node

change := true: boolean

Domin: Node  $\rightarrow$  set of Node

Domin(r) := {r}

for each  $n \in N - \{r\}$  do

    Domin(n) := N

od

repeat

    change := false

\* for each  $n \in N - \{r\}$  do

    T := N

    for each  $p \in \text{Pred}(n)$  do

        T  $\cap$ = Domin(p)

    od

    D := {n}  $\cup$  T

    if  $D \neq \text{Domin}(n)$  then

        change := true

        Domin(n) := D

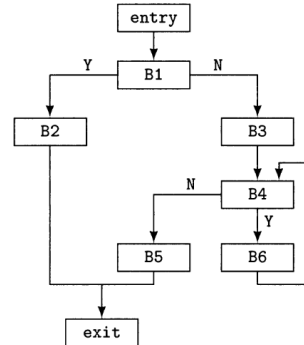
    fi

od

until !change

return Domin

end || Dom\_Comp



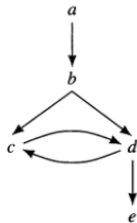
Compute the dominators.

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}



# 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  $m \rightarrow n$ , the natural loop of  $m \rightarrow n$  is
  - 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
  - the edge set connecting all the nodes in its node set.
  - Node  $n$  is called the loop header.



# Algorithm to compute natural loops

```
procedure Nat_Loop(m,n,Pred) returns set of Node
  m, n: in Node
  Pred: in Node  $\rightarrow$  set of Node
begin
  Loop: set of Node
  Stack: sequence of Node
  p, q: Node
  Stack := []
  Loop := {m,n}
  if m  $\neq$  n then
    Stack  $\oplus=$  [m]
  fi
  while Stack  $\neq$  [] 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  $\ominus=$  -1
    for each q  $\in$  Pred(p) do
      if q  $\notin$  Loop then
        Loop  $\cup=$  {q}
        Stack  $\oplus=$  [q]
      fi
    od
  od
  return Loop
end || Nat_Loop
```



# 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 and structural analysis):

- Analyzes the overall structure of the routine.
- Can handle all types of loops.
- Out of syllabus. :-)

