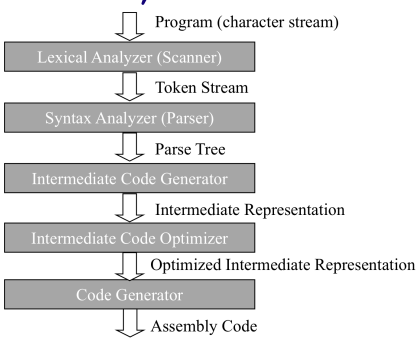
Control-Flow Analysis

Anatomy of a Compiler



Optimizations

```
int sumcalc(int a, int b, int N)
{
    int i;
    int x, y;
    x = 0;
    y = 0;
    for(i = 0; i <= N; i++) {
      x = x + (4*a/b)*i + (i+1)*(i+1);
      x = x + b*y;
    }
    return x;
}
```

Constant Propagation

```
int sumcalc(int a, int b, int N)
{
    int i;
    int x, y;
    x = 0;
    y = 0;
    for(i = 0; i <= N; i++) {
      x = x + (4*a/b)*i + (i+1)*(i+1);
      x = x + b*0;
    }
    return x;
}
```

Copy Propagation

```
int sumcalc(int a, int b, int N)
{
    int i;
    int x, y;
    x = 0;
    y = 0;
    for(i = 0; i <= N; i++) {
      x = x + (4*a/b)*i + (i+1)*(i+1);
    }
    return x;
}
```

Algebraic Simplification

```
int sumcalc(int a, int b, int N)
{
    int i;
    int x, y;
    x = 0;
    y = 0;
    for(i = 0; i <= N; i++) {
      x = x + (4*a/b)*i + (i+1)*(i+1);
      x = x;
    }
    return x;
}
```

Dead Code Elimination

```
int sumcalc(int a, int b, int N)
{
    int i;
    int x, t;
    x = 0;

    for(i = 0; i <= N; i++) {
      t = i+1;
      x = x + (4*a/b)*i + t * t;

    }
    return x;
}
```

Common Sub-Expression Elimination (CSE)

```
int sumcalc(int a, int b, int N)
{
    int i;
    int x, y, t;
    x = 0;
    y = 0;
    for(i = 0; i <= N; i++) {
      t = i+1;
      x = x + (4*a/b)*i + t * t;

    }
    return x;
}
```

Loop Invariant Removal

```
int sumcalc(int a, int b, int N)
{
    int i;
    int x, t, u;
    x = 0;
    u = (4*a/b);
    for(i = 0; i <= N; i++) {
      t = i+1;
      x = x + u *i + t * t;

    }
    return x;
}
```

Strength Reduction

```
int sumcalc(int a, int b, int N)
{
    int i;
    int x, t, u, v;
    x = 0;
    u = (4*a/b);
    v = 0;
    for(i = 0; i <= N; i++) {
      t = i+1;
      x = x + v + t*t;
      v = v + u;
    }
    return x;
}
```

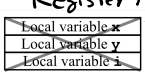
u*0, v=0,
u*1, v=v+u,
u*2, v=v+u,
u*3, v=v+u,
u*4, v=v+u,
... ...

Strength Reduction

Optimized

```
int sumcalc(int a, int b, int N)
{
    int i;
    int x, t, u, v;
    x = 0;
    u = ((a<<2)/b);
    v = 0;
    for(i = 0; i <= N; i++) {
      t = i+1;
      x = x + v + t*t;
      v = v + u;
    }
    return x;
}
```

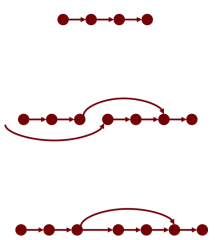
Register Allocation



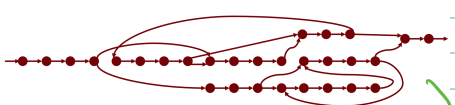

```
$t9 = x
$t8 = t
$t7 = u
$t6 = v
$t5 = i
```

Representing Control Flow

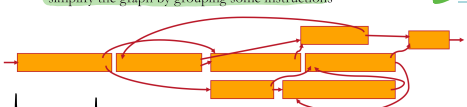
- Most instructions
  - execute the next instruction
  - straight line control-flow
- Jump instructions
  - execute form different location
  - jump in control-flow
- Branch instructions
  - execute either the next instruction or from a different location
  - fork in the control-flow



- Forms a Graph



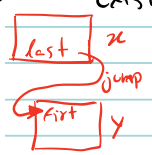
- A Very Large Graph
- Observations:
  - lots of straight-line connections
  - simplify the graph by grouping some instructions



Control Flow Graph (CFG)

- Control-Flow Graph G = <N, E>
- Nodes(N): Basic Blocks
- Edges(E): (x,y) ∈ E iff first instruction in the Basic Block y follows the last instruction in the basic block x
  - First instruction in y is the target of branch or jump instruction (last instruction) in the basic block x
  - first instruction of y is next after the last instruction of x in memory and the last instruction of x is not a jump instruction

condition to exist Edge



- Block with the first instruction of the procedure is the entry node (block with the procedure label)
- The blocks with the return instruction are exit nodes.
  - Can make a single exit node by adding a special node

Control Flow Importance

- Uncover Flow Structure:
  - Loops
  - Convergence and Divergence of Paths
- Loops are Important to Optimize
  - Programs spend a lot of times in loops and recursive cycles
  - Many special optimizations can be done on loops
- Programmers organize code using structured control-flow (if-then-else, for-loops *etc*)
  - Optimizer can exploit this
  - *but* need to discover them first

Challenges

- Unstructured Control Flow
  - Use of goto's by the programmer
  - Only way to build certain control structures

```
L1: x = 0
    if (y > 0) goto L3
L2: if (y < 0) goto L1
L3: y = y + z
    goto L2
```

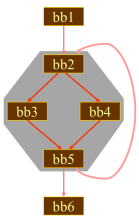
- Obscured Control Flow
  - Method Invocations
  - Procedure Variables
  - Higher-Order Functions
  - Jump Tables

```
Myobject->run()
```

need to perform analyses to identify true control-flow patterns.

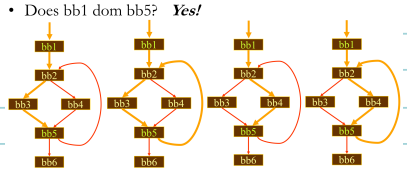
Identification of Recursive Structures Loops

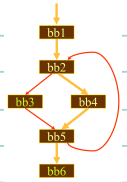
- Identify **Back** Edges
- Find the nodes and edges in the loop given by the Back Edge
- Other than the Back Edge
  - Incoming edges only to the basic block with the back edge head
  - one outgoing edge from the basic block with the tail of the back edge



Dominators

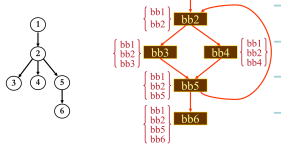
- Node x **Dominates** node y (x dom y) if every possible execution path from entry to node y includes node x
  - Does bb1 dom bb5? ***Yes!***
  - Does bb3 dom bb6? ***No!***





Dominators Sets and Tree

Dominance Relationship, direct (or immediate) and indirect represented as a tree:



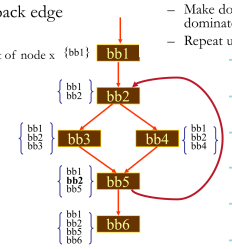
Computing Dominators

- a dom b iff
  - a = b or
  - a is the **unique immediate predecessor** of b or
  - a is a **dominator of all immediate predecessor** of b
- Algorithm
  - Make dominator set of the entry node itself
  - Make dominator set of the remainder node to be all graph nodes
  - Visit the nodes in any order
  - Make dominator set of the current node intersection of the dominator sets of the predecessor nodes + the current node
  - Repeat until no change

See Algorithm with the example in the slides 85 -

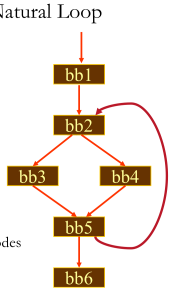
Back Edge

- An edge (x, y) ∈ E is a back edge iff y dom x
  - is node y in the dominator set of node x



Natural Loop

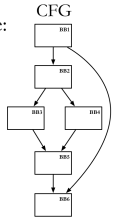
- In a CFG a Back Edge induces a Natural Loop
- Finding the Nodes of a Loop
  - Given Back Edge (s,d)
  - Traverse Backwards (against flow) from d
  - Until Reaching s
  - Collected nodes in CFG form Natural Loop
- In the Example: back edge is (5,2)
  - Natural Loop : { 2, 5, 3, 4 } why?
  - Trace Back Edge backward and collect all nodes
  - Until you reach head of the Back Edge



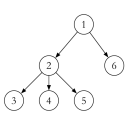
Dominance Frontier

- Definition:
  - The Dominance Frontier of a basic block N, DF(N), is the set of all blocks that are immediate successors to blocks dominated by N, but which aren't themselves strictly dominated by N
- Example:

CFG



Dominator Tree and Frontier



Node	Dominance Frontier
1	∅
2	6
3	5
4	5
5	6
6	∅

primeiro bloco que não é dominado

Dominance Frontier é quando a dominância de um certo bloco termina.

Summary

- Overview of Optimizations
- Control-Flow Analysis
- Dominators
- Back Edges and Natural Loops
- Dominance Frontier & SSA-Form

não sai no teste

está nos últimos slides