CS 526

Advanced

Compiler

Construction

http://misailo.cs.Illinois.edu/courses/cs526

Goals of the Course

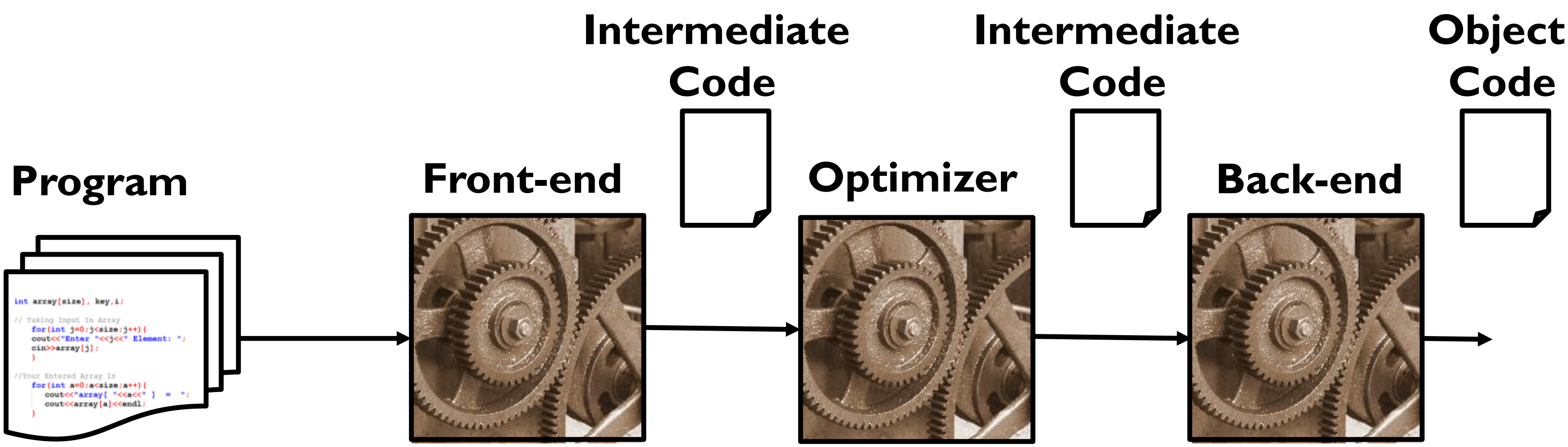
Develop a fundamental understanding of the major approaches to program analysis and optimization

Understand published research on various novel compiler techniques

Solve a significant compiler problem by reading the literature and implementing your solution in LLVM

Learn about current research in compiler technology

Compiler Overview



Preprocessing Source
- Automatic Parallelization
- Vectorization
- Cache Management
- Performance Modeling

Code Generation
- Source Code Portability
- Back-end Optimizations
- Static Profiling
- Power Management

Linking/Loading
- Interprocedural optimization
- Load-time optimization
- Security checking

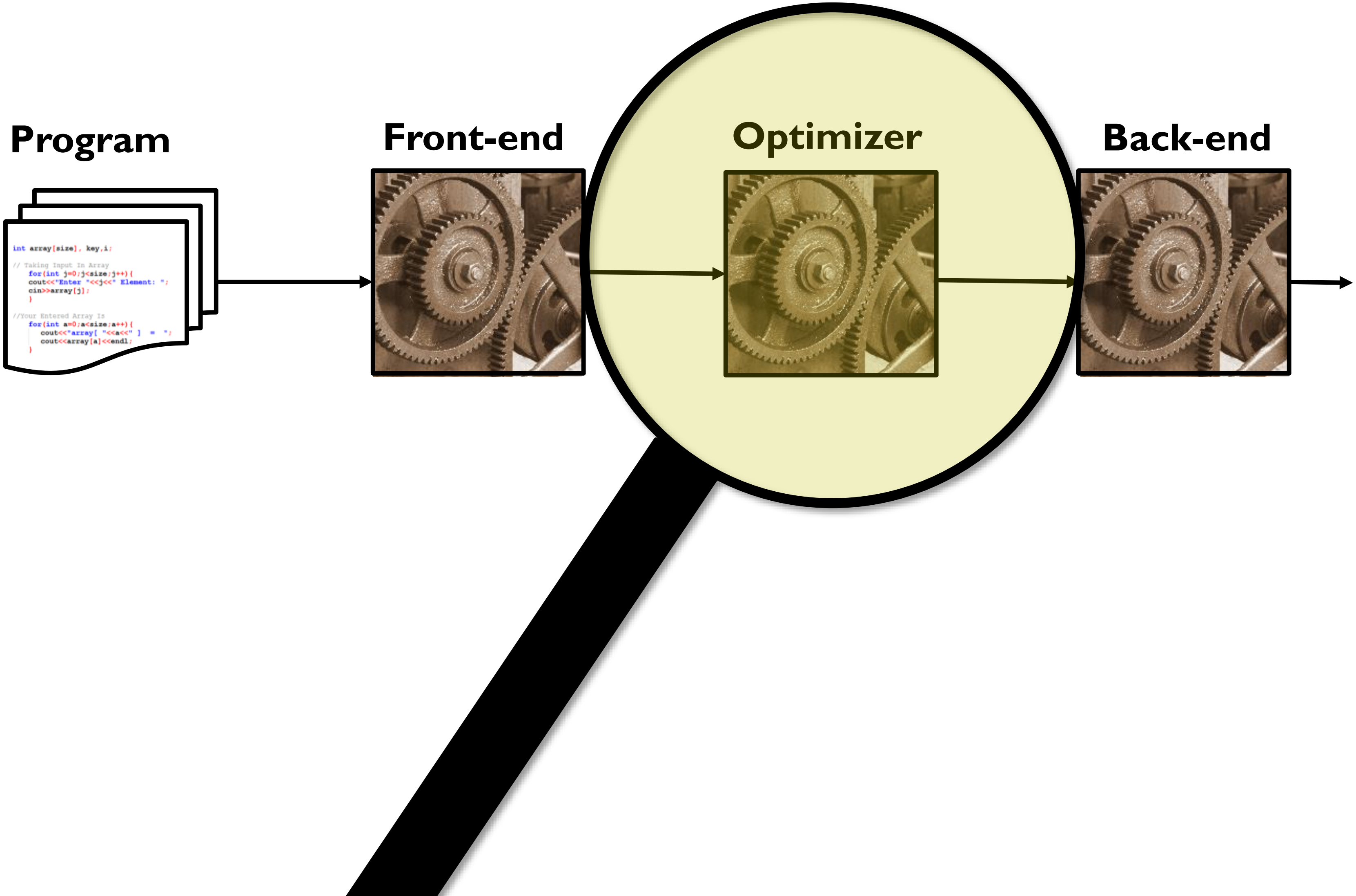
Runtime compilation
- JIT code generation
- Runtime optimization
- Fault tolerance

COMPILER =

Program Analysis +

Program Transformation

Compiler Overview



CS 526 SM

COURSE LOGISTICS

Schedule

Twice a week – Tuesdays and Thursdays 3:30-4:45 pm

Course Format

- Lectures – most of the weeks (sometimes guest)
- Projects – two programming assignments (LLVM)
- Exams – midterm and final exams
- Mini-quizzes – before (almost) every lecture

Prerequisites

Helpful (I will assume you took it):

Basic **compilers** course (e.g., CS 426)

Also helpful:

Basic **programming languages** course (e.g., CS 421)

Basic **computer architecture** (e.g., CS 233)

Most important: commitment to learn as you go

Grading

Miniquizes	10%
Midterm Exam	15%
Final Exam	25%
Projects	50%

Miniquizzes

Test background knowledge (like the one today)

- **5 minutes** at the beginning of the class
- Concept from compiler theory, something that was covered in previous courses or lectures
- We will discuss the solution immediately afterwards

Each miniquiz is **worth ~0.5%** (up to 10%).

- **Self-graded**, the main purpose is to bring everyone to the same page before we start the discussions
- In total 25 quizzes; can miss 5 without penalty

Exams

Midterm

- In class (March 16; before the break)
- Focuses on analysis (SSA, dataflow, dependency)
- 75 minutes

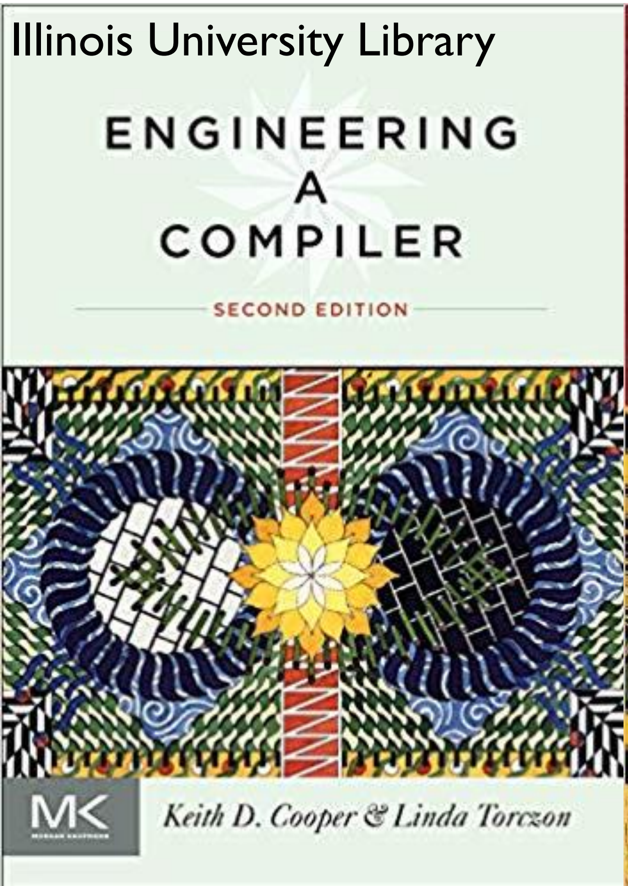
Final

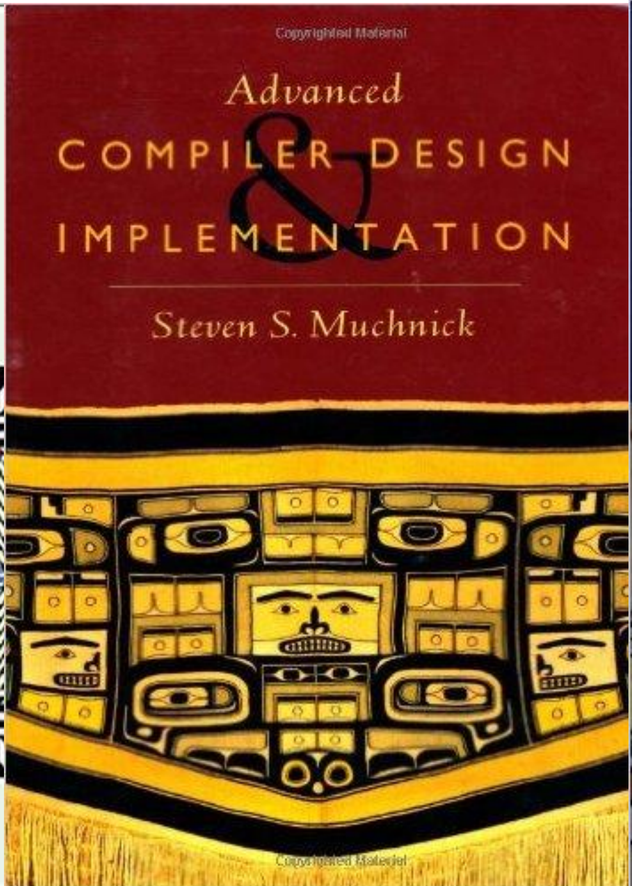
- In the finals week
- Pointer analysis, optimization and special topics
- Also include the materials from the midterm
- Up to 3 hours

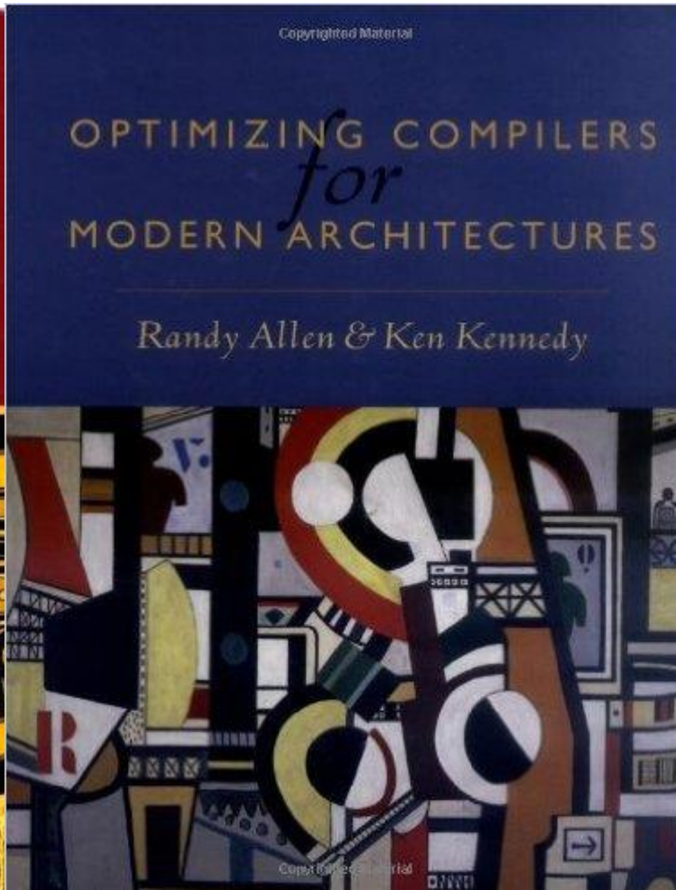
Books

No official book, but many times you will **need** to look into one of these:

Available online via
Illinois University Library



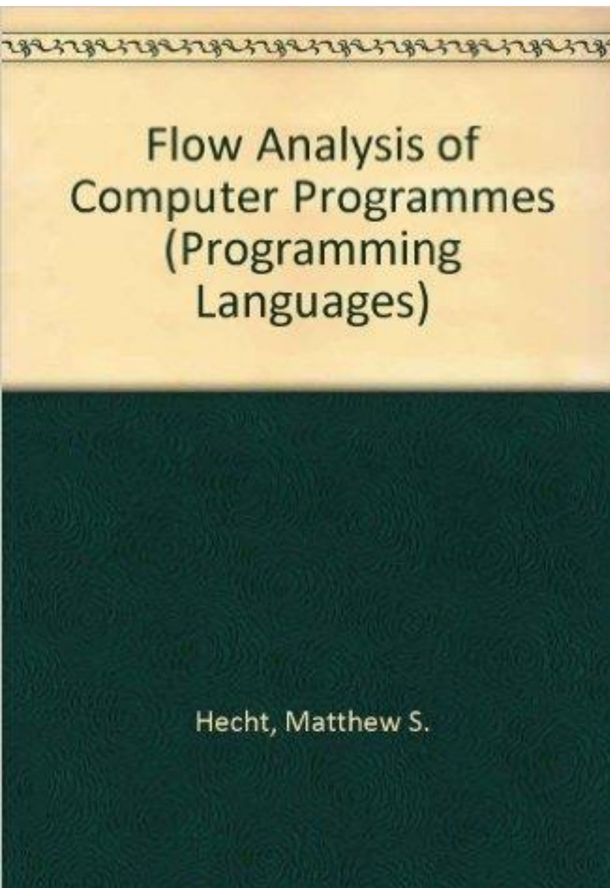


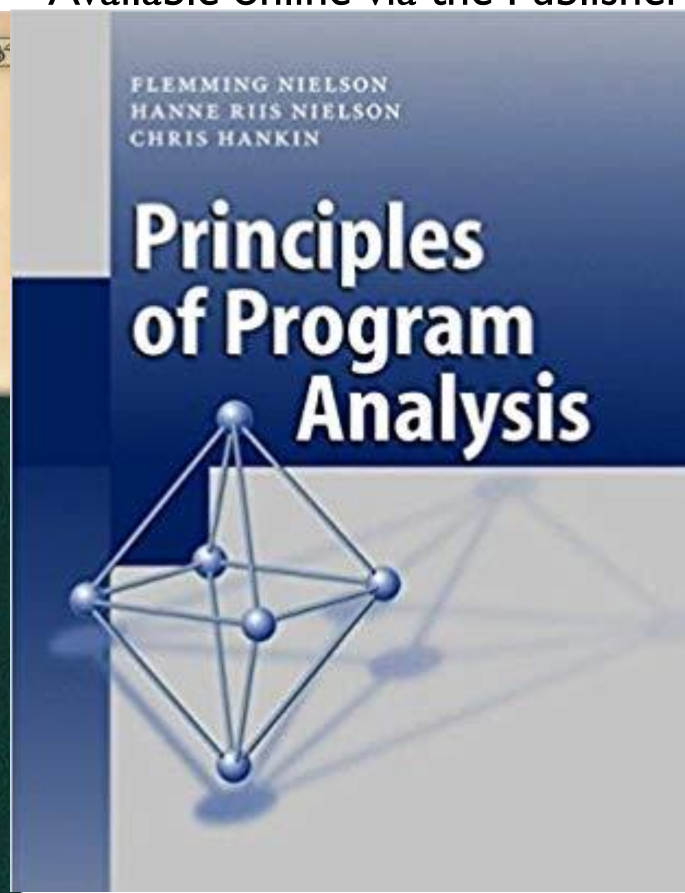


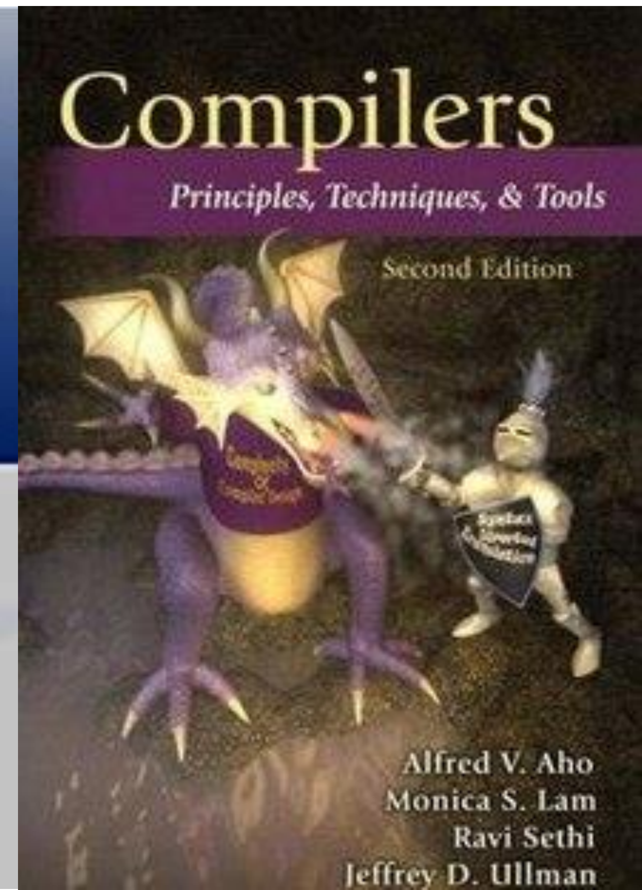
And More Books

No official book, but many times you will **need** to look into one of these:

Available online via the Publisher







Projects

Gain experience solving existing compiler problems

- Read the literature for the problems
- Find or develop a solution
- Implement the solution in a realistic compiler
- Test it on realistic benchmarks

Projects

P1 – Warm-up exercise:

- ***Individual,*** 3 weeks but do it sooner
- Scalar replacement of aggregates via SSA (Muchnick, Chapter 12)
- Goal: become familiar with the infrastructure

P2 – Main problem

- ***Groups of two,*** 12 weeks, also do it sooner!
- Choose and solve a harder problem (Suggestions coming soon)

Infrastructure

LLVM: Low Level Virtual Machine http://llvm.org

- Virtual instruction set: RISC-like, SSA-form
- Powerful link-time (interprocedural) optimization system
- Many front-ends: C/C++, D, Fortran, Julia, Haskell, Objective-C, OpenMP, OpenCL, Python, Swift, ...
- Software: 1.3M+ lines of C++
- Open source: In use at many universities and major companies

Infrastructure

Prepare for the project, during this week:

Read LLVM Documentation at http://llvm.org/docs:

Introduction to the LLVM Compiler Infrastructure

Follow instructions in the ***Getting Started*** and ***Writing an LLVM Pass*** guides to:

(a) Download LLVM 7.0.1, with Clang and test-suite

(b) Do a full build (no need to run "make install")

(c) Compile and run the "Hello" pass

Install on **your EWS**: `ssh <netid>@linux.ews.illinois.edu`

Get in Touch

Email: misailo@illinois.edu

- Please include "CS 526" in the subject line

Office: Siebel Center, office 4110

Office Hours:

- By appointment (send me an email)
- I am typically free right after the class
- We can organize dedicated office hours before the exams

CS 526

QUESTIONS SO FAR?

CONTROL FLOW ANALYSIS

The slides adapted from Vikram Adve

Flow Graphs

Flow Graph: A triple G=(N,A,s), where (N,A) is a (finite) directed graph, s ∈ N is a designated "initial" node, and there is a path from node s to every node n ∈ N.

- An *entry node* in a flow graph has no predecessors.
- An *exit node* in a flow graph has no successors.
- There is exactly one entry node, s. We can modify a general DAG to ensure this. *How?*

Control Flow Graph (CFG)

Flow Graph: A triple G=(N,A,s), where (N,A) is a (finite) directed graph, s ∈ N is a designated "initial" node, and there is a path from node s to every node n ∈ N.

Control Flow Graph (CFG) is a flow graph that represents all ***paths*** (sequences of statements) that might be traversed during program execution.

- Nodes in CFG are program statements, and edge (S_1,S_2) denotes that statement S_1 can be followed by S_2 in execution.
- In CFG, a node unreachable from s can be safely deleted. *Why?*
- Control flow graphs are usually *sparse*. I.e., $|A| = O(|N|)$. In fact, if only binary branching is allowed $|A| \leq 2|N|$.

Control Flow Graph (CFG)

Basic Block is a sequence of statements S_1 ... S_n such that execution control must reach S_1 before S_2, and, if S_1 is executed, then $S_2 \ldots S_n$ are all executed in that order

- Unless a statement causes the program to halt

Leader is the first statement of a basic block

Maximal Basic Block is a basic block with a maximum number of statements (n)

Control Flow Graph (CFG)

Let us refine our previous definition

CFG is a directed graph in which:

- Each node is a single basic block
- There is an edge b1 → b2 if block b2 *may* be executed after block b1 in *some* execution

We typically define it for a single procedure

A CFG is a conservative approximation of the control flow! Why?

Example

Source Code

```
unsigned fib(unsigned n) {
   int i;
   int f0 = 0, f1 = 1, f2;

   if (n <= 1) return n;

   for (i = 2; i <= n; i++) {
      f2 = f0 + f1;
      f0 = f1;
      f1 = f2;
   }

   return f2;
}
```

LLVM bitcode (ver 3.9.1)

```
define i32 @fib(i32) {
  %2 = icmp ult i32 %0, 2
  br i1 %2, label %12, label %3

; <label>:3:
  br label %4

; <label>:4:
  %5 = phi i32 [ %8, %4 ], [ 1, %3 ]
  %6 = phi i32 [ %5, %4 ], [ 0, %3 ]
  %7 = phi i32 [ %9, %4 ], [ 2, %3 ]
  %8 = add i32 %5, %6
  %9 = add i32 %7, 1
  %10 = icmp ugt i32 %9, %0
  br i1 %10, label %11, label %4

; <label>:11:
  br label %12

; <label>:12:
  %13 = phi i32 [ %0, %1 ], [ %8, %11 ]
  ret i32 %13
}
```

See You Next Time!

Review in the next few weeks:

Muchnick, Chapter 21: Case Studies of Compilers

Review by next Tuesday:

Cytron, Ferrante, Rosen, Wegman, and Zadeck,

"Efficiently Computing Static Single Assignment Form and the Control Dependence Graph,"

ACM Trans. on Programming Languages and Systems, 13(4), Oct. 1991, pp. 451–490.

If you see this, I clicked a wrong button