Program Representations

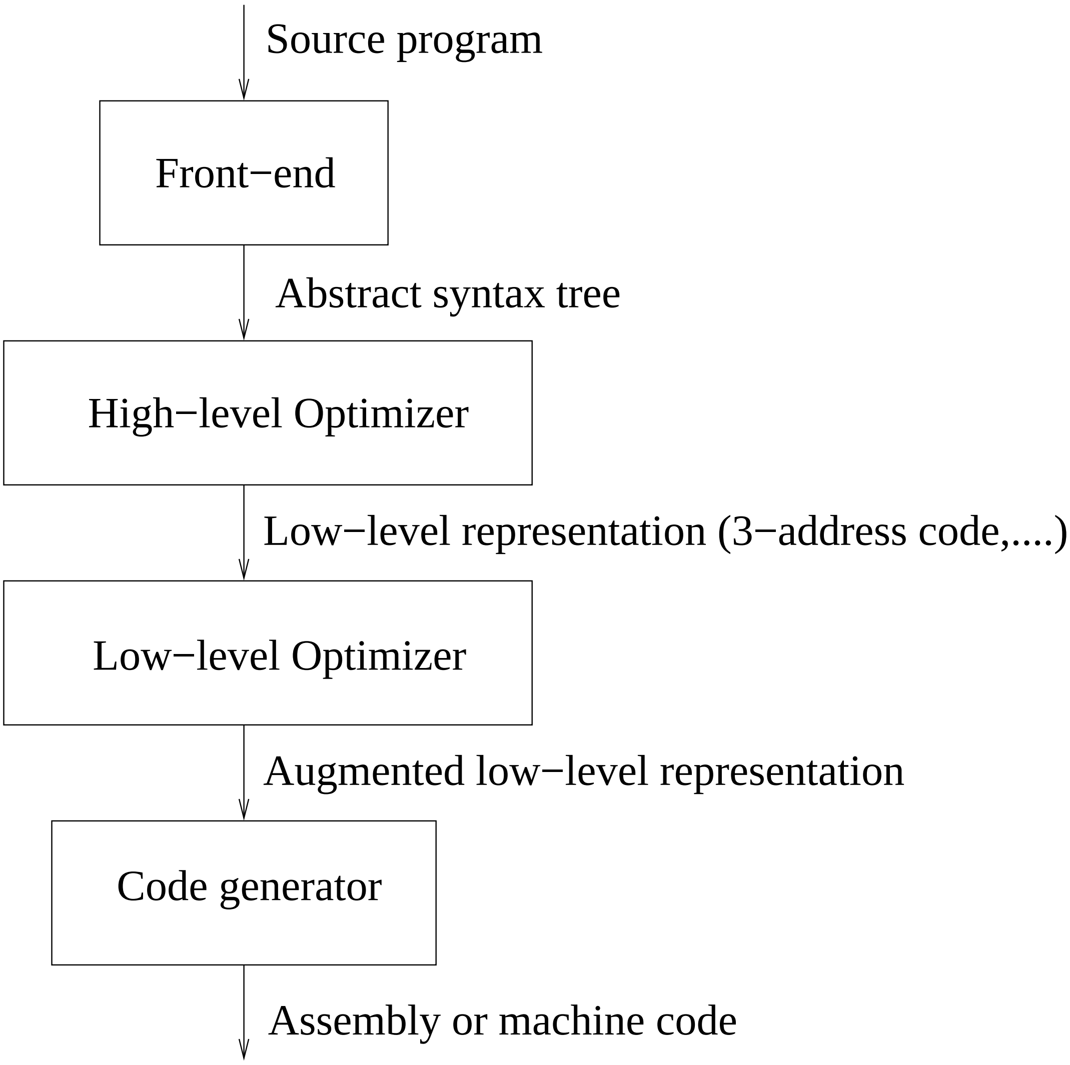
Announcements

Assignment 1 is posted on the course web-site.

Today

- Organization of modern compilers
- Control flow graphs
- Opportunities for scalar optimization
- Dataflow analysis

Major phases of a modern compiler



Source program: collection of files, each of which is a sequence of characters
Output of compiler: assembly/machine for actual machine or virtual machine like JVM

Front-end

Goal: convert linear structure of input program into hierarchical structure

Input: source program

Output: abstract syntax tree + symbol table

Tasks:

- **lexical analysis**: convert sequence of characters in a file into sequence of tokens
- **parsing**: convert sequence of tokens into a hierarchical representation of program structure (abstract syntax tree)
- **auxiliary tasks**:
  - macro/template expansion
  - produce a symbol table
  - perform type-checking
  - ....

High-level Optimizer

Goal: perform high-level analysis and optimization of program

Input: AST + symbol table from front-end

Output: Low-level program representation such as 3-address code

Tasks:

- procedure/method inlining
- array dependence analysis
- loop transformations: unrolling, permutation, tiling, jamming, distribution,...
- ....

Low-level Optimizer

Goal: perform scalar optimizations on low-level representation of program

Input: low-level representation of program (3-address code)

Output: optimized low-level representation of program + auxiliary information (def-use chains,SSA etc.)

Tasks

- dataflow analysis: reaching definitions, live variable analysis,...
- generation of auxiliary information: conversion to SSA form,...
- scalar optimizations: constant folding, partial redundancy elimination, strength reduction,...

Code Generator

Goal: produce assembly/machine code from optimized low-level representation of program

Input: optimized low-level representation of program from low-level optimizer

Output: assembly/machine code for real or virtual machine

Tasks:

- Register allocation
- Instruction selection
- Data movement instruction generation
- ....

Notes

- Front-ends may generate low-level representation directly if compiler does not do high-level optimizations.
- Some compilers may regenerate high-level representation from low-level representation (many Java compilers).
- Nowadays, compilation can be performed *off-line* or *on-line* while program is executing (just-in-time compilation).

Focus of next couple of weeks: low-level optimizer

How does a compiler analyze and optimize low-level representation of program?

Low-level representation

We will use a simple 3-address representation:

- Statement:
  - performs one arithmetic/logical operation, or
  - compute boolean expression and jump conditionally
  - unconditional jump
- Any statement can be given a label and jump targets are labels.
- Book-keeping statements: denote start and end of procedures/functions/etc.

Targets of jump statement: statements whose execution may immediately follow execution of jump statement

Explicit targets of jump statement: targets mentioned explicitly in jump statement

Implicit target of jump statement: statement that follows jump statement

For now, focus on one procedure. Assume statements are numbered sequentially.

Running example

```
          A = 4
          t1 = A * B
L1:       t2 = t1 / C
          if t2 < W goto L2
          M = t1 * k
          t3 = M + I
L2:       H = I
          M = t3 - H
          if t3 ≥ 0 goto L3
          goto L1
L3:       halt
```

First task is to make flow of control explicit: produce **control-flow graph**

Control Flow Graph

- Divides statements into **basic blocks**
- Basic block: a maximal sequence of statements $I_0, I_1, \ldots, I_n$ such that if I_j and I_{j+1} are two adjacent statements in this sequence, then
  - execution of I_j is always followed immediately by the execution of I_{j+1}, and
  - execution of I_{j+1} is immediately preceded by execution of I_j.
- Edges between basic blocks represent potential flow of control.

More formally, CFG $= \langle V, E, Entry \rangle$, where

V = vertices or nodes, representing a statement or basic block (group of statements).
E = edges, potential flow of control
$E \subseteq V \times V$
$Entry \in V$, unique program entry

For convenience, assume all V are reachable from $Entry$,

$$(\forall v \in V)[Entry \xrightarrow{*} v]$$

Control Flow Graph Construction

Constructing *CFG*s with basic blocks (sets of statements)

- Identify *Leaders* - first statement of a basic block
- In lexicographic order, construct a block by appending subsequent statements up to, but not including, the next leader.

Leader identification:

1. first statement in the program, or
2. explicit target statement of any conditional or unconditional branch, or
3. statement immediately following a conditional or unconditional branch (this statement is an *implicit* target).

Basic Block Partition Algorithm

Input: set of statements,
stat(i) = i^{th} statement in input program

Output: set of *leaders*, set of basic blocks where *block(x)* is the set of statements in the block with *leader* x.

Algorithm:

```
leaders = {1}                  // Leaders, first statement
for i = 1 to |n|               // n = number of statements
    if stat(i) is a branch then
        leaders = leaders ∪ all potential targets of stat(i)
endfor
worklist = leaders             // Basic blocks
while worklist not empty do
    x = smallest numbered stat in worklist
    worklist = worklist - {x}
    block(x) = {x}
    for (i = x + 1; i ≤ |n| and i ∉ leaders; i++ )
        block(x) = block(x) ∪ {i}
    endfor
endwhile
```

Basic Block Example

```
 1          A = 4
 2          t1 = A * B

 3   L1:    t2 = t1 / C
 4          if t2 < W goto L2

 5          M = t1 * k
 6          t3 = M + I

 7   L2:    H = I
 8          M = t3 - H
 9          if t3 ≥ 0 goto L3

10          goto L1

11   L3:    halt
```

Leaders =

Blocks =

Determining the Edges in a Control Flow Graph

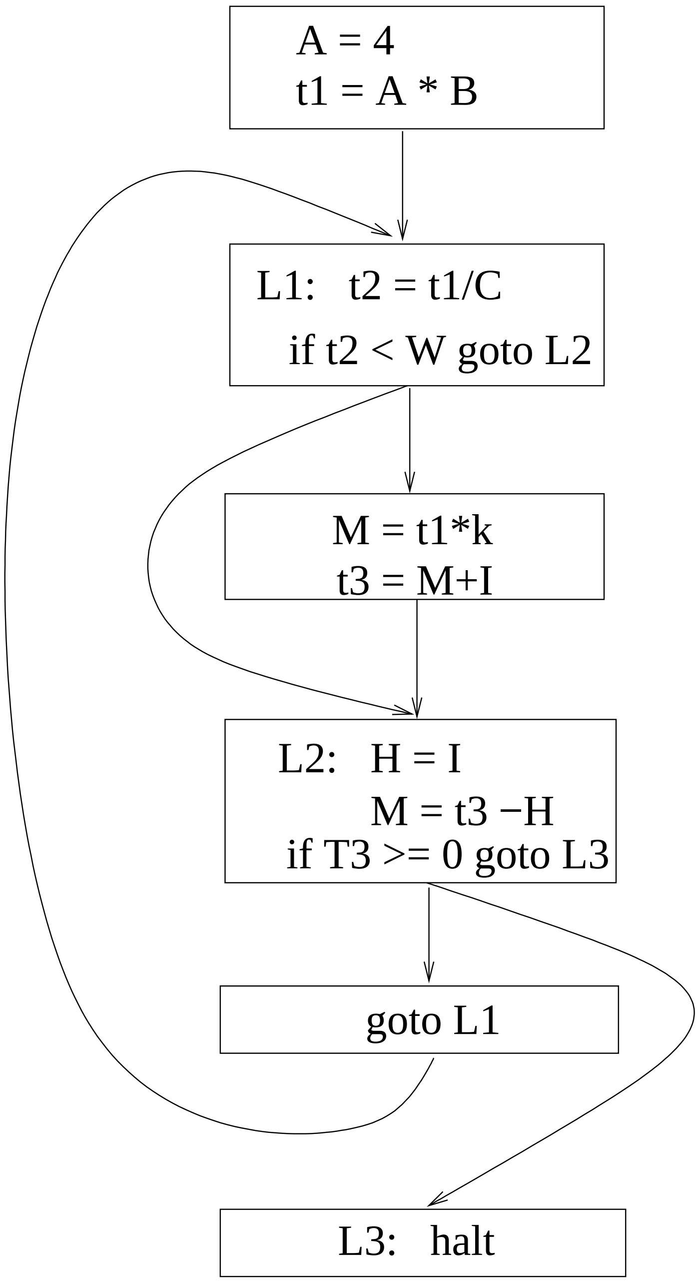
$\exists$ directed edge from B_1 to B_2 if:

1. $\exists$ a branch from the last statement of B_1 to the first statement B_2 (B_2 is a leader).
2. B_2 immediately follows B_1 in program order and B_1 does not end with an unconditional branch.

Input: *block()*, a sequence of basic blocks
Output: *CFG* where nodes are basic blocks
Algorithm:

```
for i = 1 to the number of blocks do
    x = last statement of block(i)
    if stat(x) is a branch then
        for each explicit target y of stat(x)
            create edge from block i to block y
        endfor
    if stat(x) is not an unconditional branch then
        create edge from block i to block i + 1
endfor
```

CFG Example



A = 4
t1 = A * B
L1: t2 = t1/C
if t2 < W goto L2
M = t1*k
t3 = M+I
L2: H = I
M = t3 −H
if T3 >= 0 goto L3
goto L1
L3: halt

Discussion

- We defined CFG as a graph in which nodes represent basic blocks. In some situations, we will also consider the *statement-level CFG* in which nodes are individual statements. We will use the term **CFG** to mean either kind of graph.

- In statement-level CFG, it is sometimes convenient to use a node to explicitly represent merging of control in the control flow graph as shown in the next slide.

- If the input language is structured, the front-end can generate the basic block CFG directly.

Statement-level CFG Example