Optimization Overview

Most complexity in modern compilers is in the optimizer

- Also by far, the largest phase in terms of compile-time and in terms of source code size (recall: lexing, parsing, semantic analysis, optimization, code generation)

When should we perform optimizations?

- On AST?
  - Pro: Machine independent
  - Con: Too high level
    - Too abstract --- need to have more details to be able to express the "kind" of machine for which the AST needs to be compiled (e.g., register or stack machine or quantum computer)
- On assembly language
  - Pro: exposes optimization opportunities
  - Con: may be too low level, making optimization difficult (need to undo/redo certain decisions)
  - Con: machine dependent
  - Con: must reimplement optimizations when retargetting
- On IR
  - Pro: can be machine independent, if designed well (can represent a large family of machines)
  - Pro: can expose optimization opportunities, if designed well.
  - Con: IR design critical for exposing optimization opportunities.

We will be looking at optimizations on an IR which has the following grammar:

```
P --> S P | S
S --> id := id op id
       | id := op id
       | id := id
       | push id
       | id := pop
       | if id relop id goto L
       | L:
       | jump L
```

- id's are register names
- constants can replace ids
- typical operators: +, -, *

A basic bloack is a maximal sequence of instructions with

- no labels (except at the first instruction), and
- no jumps (except in the last instruction)

Idea:

- Cannot jump into a basic block (except at beginning)
- Cannot jump out of a basic block (except at end)
- A basic block is a single-entry, single-exit, straight-line code segment

Once we reach the start of a basic block, we are guaranteed to execute all instructions in the BB. Furthermore, the only way into the basic block is through the first statement.

Consider the basic block:

```
1. L:
2.   t := 2*x
3.   w := t + x
4.   if w > 0 goto L'
```

(3) executes only after (2)

- We can change (3) to w := 3 * x
- Can we eliminate (2) as well? Need to be sure that `t` is not used in other basic blocks.

A control-flow graph is a directed graph with

- Basic blocks as nodes
- An edge from block A to block B if the execution can pass from the last instruction in A to the first instruction in B
  - e.g., the last instruction in A is `jump Lb`
  - e.g., the last instruction in A is `if id relop id then goto Lb`
  - e.g., execution can fall-through from block A to block B

Example control-flow graph:

```
BB1:
  x := 1
  i := 1

BB1-->BB2

BB2:
L:
  x := x * x
  i := i + 1
  if i < 10 goto L

BB2 --> BB2

BB2 --> BB3
```

The body of a method (or procedure) can be represented as a control-flow graph. There is one initial node (entry node). All "return" nodes are terminal.

Optimization seeks to improve a program's resource utilization

- Execution time (most often)
- Code size
- Memory usage
- Network messages sent, disk accesses, etc.
- Power consumption

Optimization should not alter what the program computes

- The answer must still be the same.

For languages like C, there are typically three granularities of optimization

- Local optimizations
  - Apply to a basic block in isolation
- Global optimizations
  - Apply to a control-flow graph (method body) in isolation
- Inter-procedural optimizations
  - Apply across method boundaries

Production compilers do all these types of optimizations. In general, easies to implement local optimizations and hardest to implement inter-procedural optimizations.

Criteria for evaluating an optimization

- Payoff. What is a good payoff?

- Frequency of occurrence
- Ease of Implementation
- Compilation Time

In practice, often a conscious decision is made not to implement the fanciest optimization known. Why?

- Some optimizations are hard to implement
- Some optimizations are costly in compilation time
- Some optimizations have low payoff. But hard to establish the payoff. Often one optimization may trigger another one, and this is hard to predict in advance.
- Many fancy optimizations are all three!

Current state-of-the-art: the goal is "Maximum benefit for minimum cost"