Optimization

- The first thing we need to make clear about Optimization is that we aren't optimizing
  - We are going to use heuristics to come up with a good solution, not an optimal solution

- There are a bunch of different types of optimization
  - To a large extent, the entire purpose of a low-level language is optimization
    - Register allocation is optimization (could just leave in memory)
    - Code scheduling is optimization
    - Instruction selection is optimization
    - The phases we call Optimization are obviously optimization

Optimization

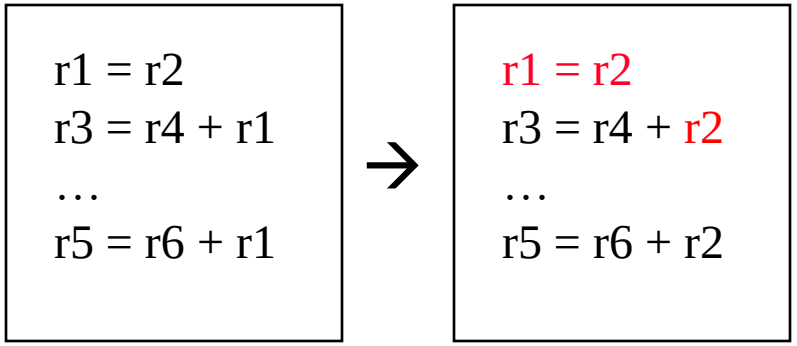
- Optimization is expensive, in terms of compilation time
  - Requires very costly Dataflow Analysis
  - Cannot be done indiscriminately
- "Biggest bang for the buck" is achieved by targeting the "hot" sections of the code; i.e., the inner loops
  - A program that runs even a few seconds will execute millions of instructions
  - Saving 2 or 3 cycles in main() makes no difference
  - But if you save 2 or 3 cycles in an inner loop executed 10M times, then you've had an impact
- Scope of optimization
  - Local: scope is just basic block; much cheaper
  - Global: scope is function-wide
  - Inter-procedural: Complicated and expensive

Optimization

- Some of the types of optimization
  - Jump optimization
    - e.g., target of a branch is a jump
  - Code-simplifying optimizations
    - e.g., Constant folding, common-subexpression elimination
  - Loop-based optimizations
    - Pull instructions out of loop
    - e.g., A[5] = b Address computation and load can be moved out
  - Peephole
    - Using a trick specific to target architecture, such as looking for opportunities to use X86 lea
  - Function inlining

Optimization

- Due to shortage of time, let's just look at one optimization in detail, just to get a feel for what the compiler has to do
  - Will look at forward copy propagation



- Forward copy propagation
  - Looks for a pattern
- Requirements
  - Find a mov between regs
  - Search rest of block to:
    - See where r1 used
    - Be sure r2 not redefined before last use r1
    - search ends if r1 redefined
  - To delete move, need to search all control flow paths exiting block, to see if r1 needs to be used again

Optimization

- Some notes on forward copy propagation
  - Probably this optimization doesn't eliminate the move
    - You just try to cause dead code, which an dead code elimination optimization will remove later
    - Note that optimization is an iterative process
      - One optimization often facilitates another
  - This was just a local version
    - Even if it doesn't eliminate the move, could improve the schedule
    - You could also do a global version, but the dataflow analysis required is complicated
  - May cause adverse effects
    - If you extend the live range of r2, without eliminating the live range of r1, you have just increased register pressure
      - Greater chance of spilling

Optimization

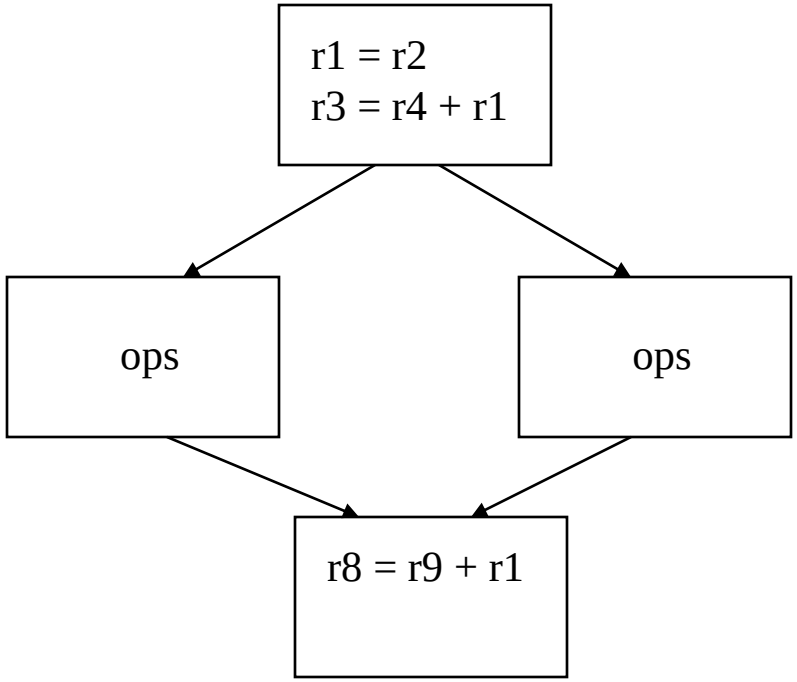
- Example pseudo-code for forward copy propagation

```
for (opA = bb.getFirstOp(); opA != null; opA = opA.getNextOp()) {
   if (opA.getType() != OPER_MOV)
      continue;
   srcReg = opA.getSrc(0);
   destReg = opA.getDest(0);
   for (opB = OpA.getNextOp(); opB != null; opB = opB.getNextOp) {

      if (opB.isDestReg(srcReg) )   // stop when redefine src of move
         break;
      if (opB.isDestReg(destReg) )   // stop when redefine dest of move
         break;
      if (opB.isSrcReg (destReg) {
         opB.substituteSrc (destReg, srcReg);
      }
   }
}
```

Optimization

- What would be required to make this optimization global?
  - Does the definition of r1 reach? Is it only definition reaching? Is r2 live in last block? Does the same definition of r2 reach the last block?
  - Dataflow analysis is very complex

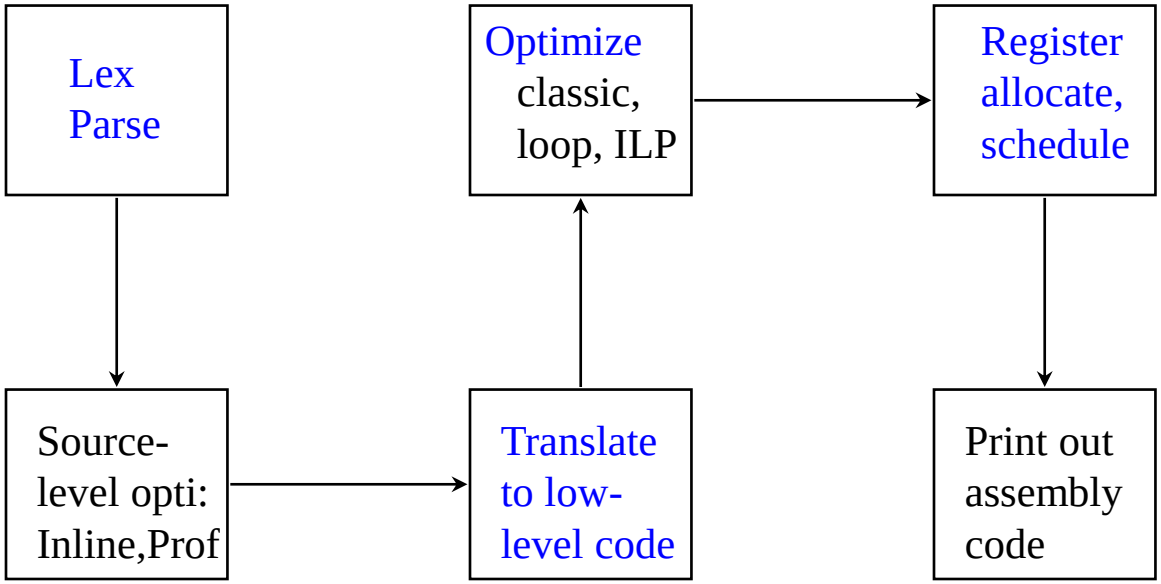


Optimization

- Final thoughts on optimization
  - Can result in dramatic improvements in execution time
    - Good versus bad compile can have 2-5 times difference in execution speed
  - Understanding optimization can make you a better programmer
    - Rather than relying on compiler, don't put loop invariant code inside a loop
    - Know what gives an optimizer problems
      - e.g., pointers are real tough
      - Eliminating a load of *p, and using previous load can only be done if compiler can prove no one wrote to same memory location in between
        - You can declare a local var to hold the value, and compiler doesn't have to figure it out

Summary

- We've hit the highlights of the compilation process
  - Hopefully you understand pretty well what goes into writing a compiler
- We've seen the power of objects for designing code
  - Can I say "polymorphism" one more time?
- We've seen interesting data structures



Compilers

Are

Cool !!!

CS 3510 – Compiler Theory and Practice