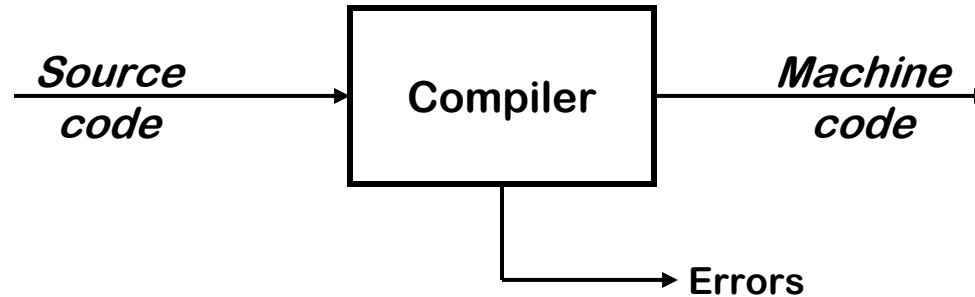


Wrapping Up

Copyright 2003, Keith D. Cooper, Ken Kennedy & Linda Torczon, all rights reserved.
Students enrolled in Comp 412 at Rice University have explicit permission to make copies of these materials for their personal use.

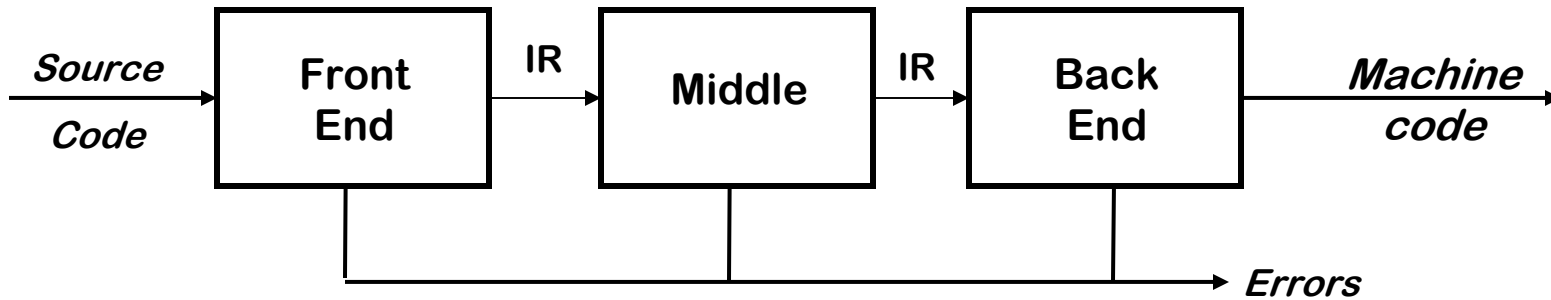
High-level View



Definitions

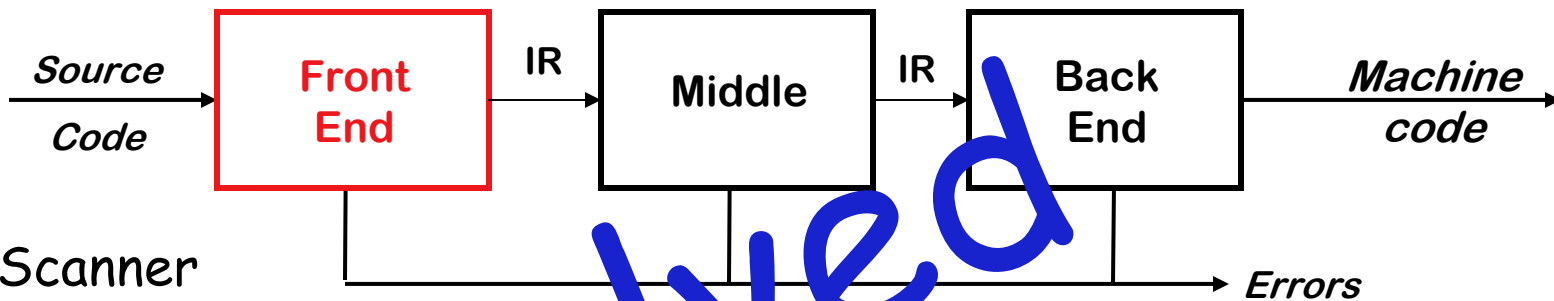
- Compiler consumes code & produces code
- Interpreter consumes executable & produces results

High-level View



- Front end consumes source code & produces IR
 - Determines code shape for rest of compiler
- Middle reads & writes IR
 - Analyzes & transforms IR to "improve" it
 - Myriad techniques at several distinct scopes
- Back end consumes optimizes IR & produces target code
 - Maps code to target ISA & handles resource issues

Front End



Scanner

- Applies DFA technology to classify words
- Use RE-based tools to generate fast scanners

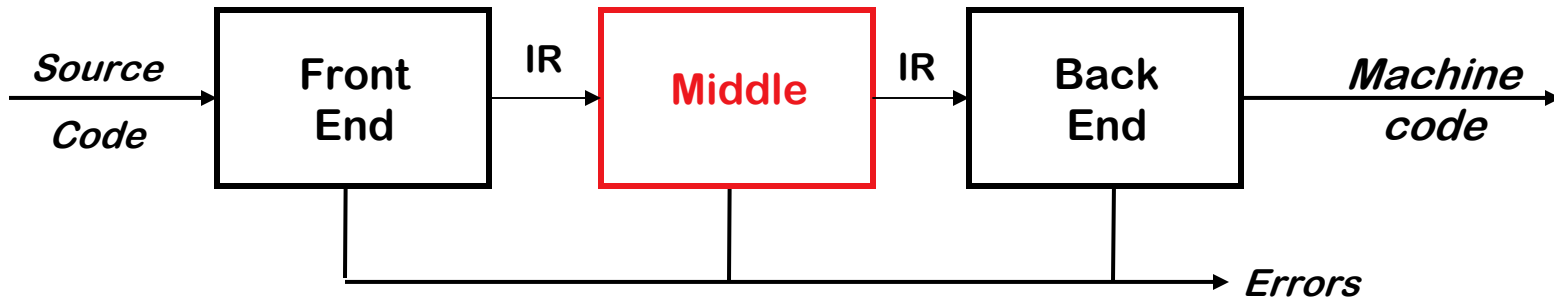
Parser

- Takes stream of classified words & computes structure
- Many techniques (LR(0), LL(1), LR(1) + others)

Context-sensitive Analysis

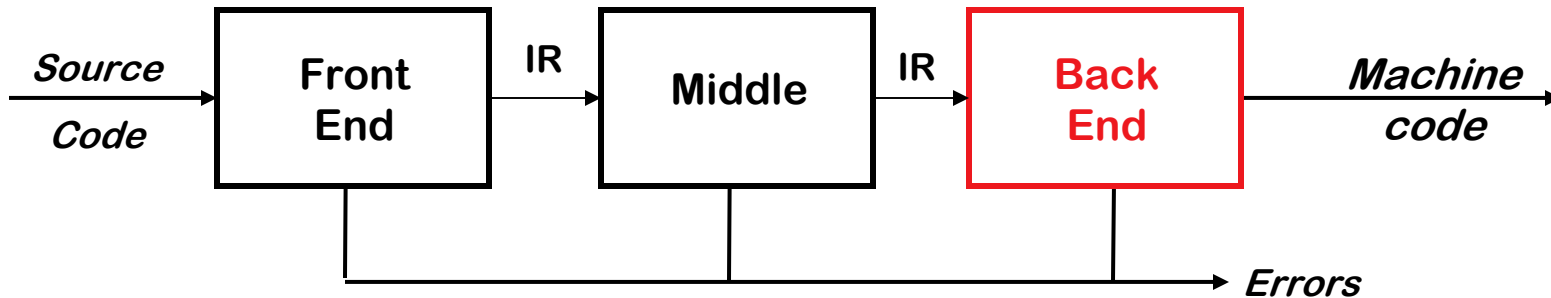
- Type-checking & IR generation (AGs & Ad-hoc SDT)
- Problems are harder & tools not as well developed

Middle (Optimizer)



- Typically structured as a series of passes
 - Allows for separation of concerns
 - Simplifies development & debugging
- Rewrites IR to improve running time, code space, ...
- We studied redundancy elimination
 - Other optimizations include code motion, dead-code elimination, constant propagation, strength reduction, ... (see Ch 10)

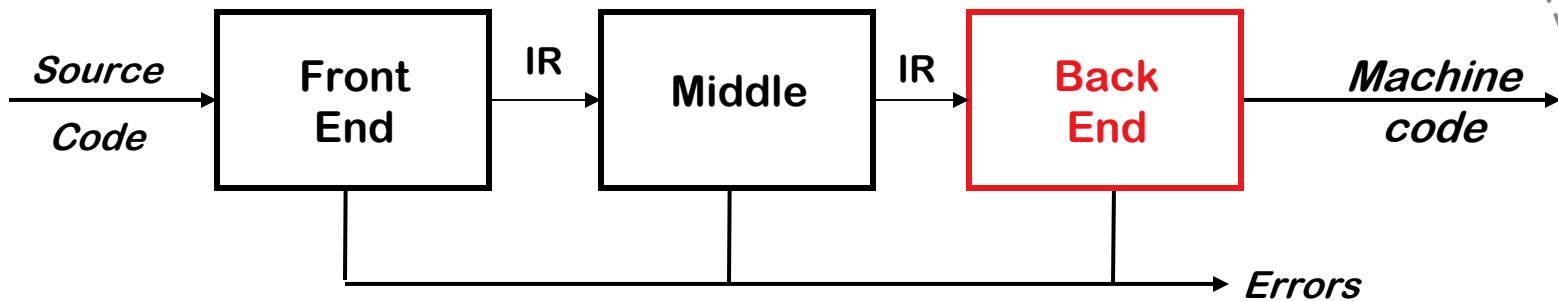
Back End



Three Problems

- **Instruction Selection**
 - Mapping optimized IR to target machine's ISA
- **Instruction Scheduling**
 - Reordering operations to hide latencies & speed execution
- **Register Allocation**
 - Mapping name space of optimized IR to target's register set

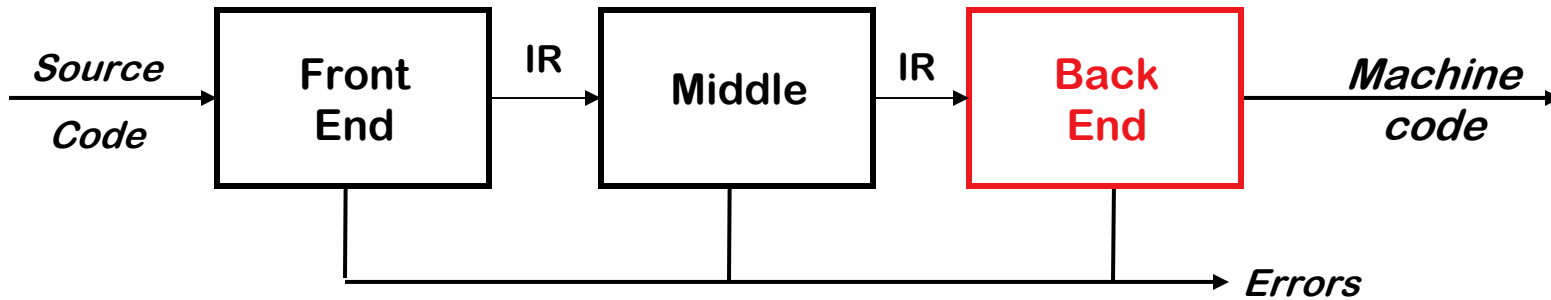
Back End



Instruction Selection

- This is a pattern-matching problem
 - We looked at tree pattern matching, peephole optimization
 - Other ideas: string matching, ad-hoc matching, parsing, ...
- Criterion is "local optimality"
 - Map immediate context to ISA in an efficient way
 - Leave "global optimality" to optimizer
- My algorithm of choice: peephole matching

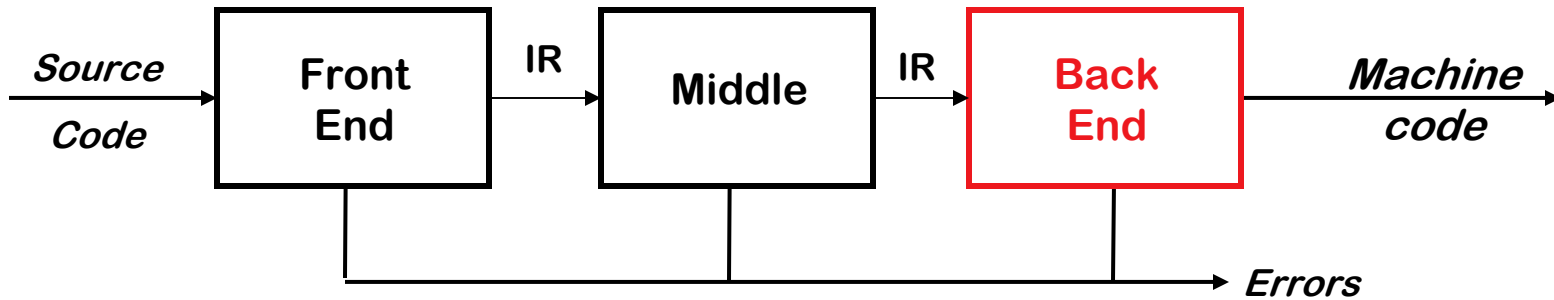
Back End



Instruction Scheduling

- Fixed set of operations, can vary the order
 - Must preserve inter-operation dependences
 - May change demand for registers
- List scheduling is dominant paradigm
 - Applied to blocks, extended blocks, regions in code
- My algorithm of choice: Schielke's RBF algorithm on EBBs

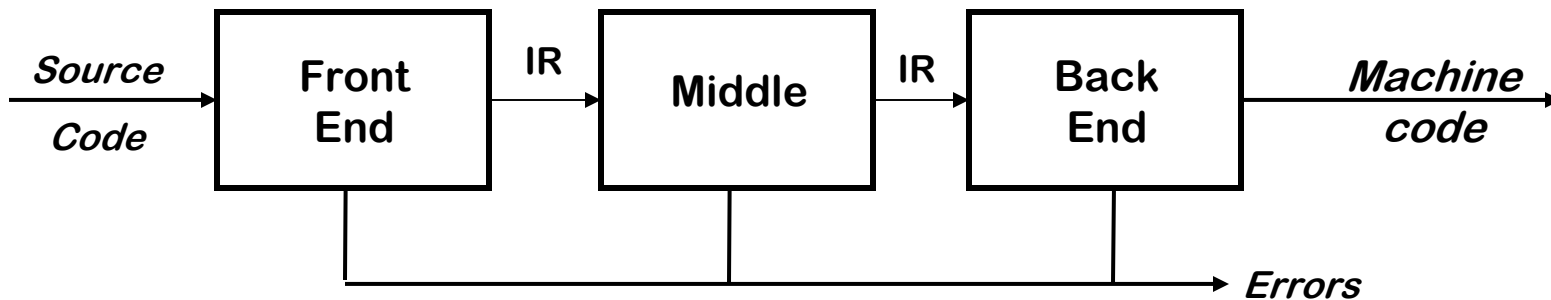
Back End



Register Allocation

- Handle disconnect between optimizer's assumption of unlimited registers and chip's limited set
 - Insert loads and stores to reconcile the two models (spills)
 - Minimize the slowdown
- Global allocation via graph coloring is the dominant paradigm
 - Build interference graph and color it
 - Many minor variations, but one strong theme

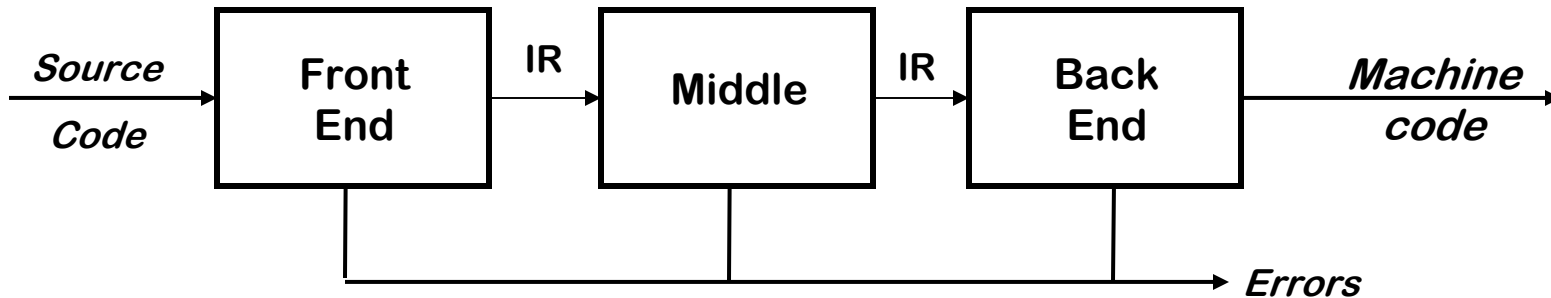
Perspective



Different concerns in each phase

- Front end worries about shaping code for optimization
 - Expose facts that can be improved & tailored
- Middle worries about global issues that affect speed, space
 - Redundancy, constants, code motion, locality, ...
- Back end worries about using the target's resources
 - Address modes, functional units, registers, issue slots, ...

Perspective

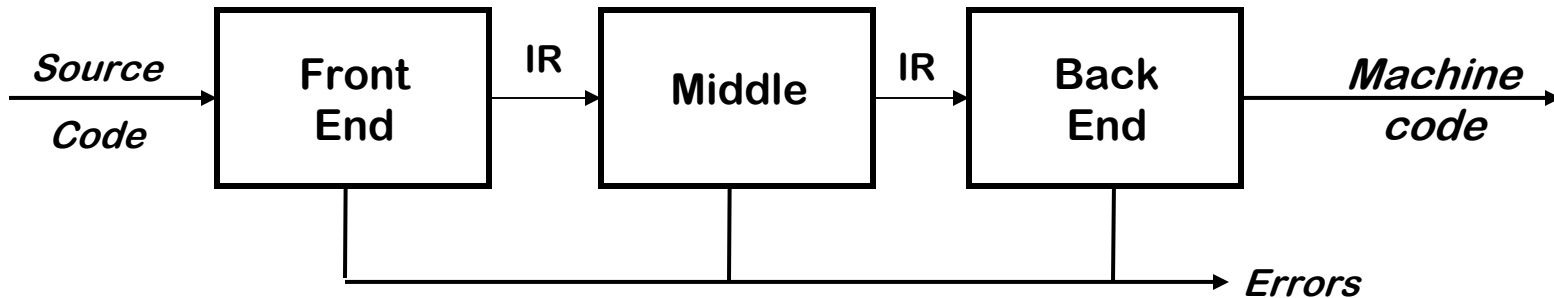


Different complexities in each phase

- Front end is mostly linear time
 - DFAs, parsers take time proportional to size of input stream
- Middle is low-order polynomial time
 - Analysis & optimization are $O(n)$ to $O(n^2)$, in general
- Back end solves hard problems with approximations
 - Heuristics that typically take $O(n)$ to $O(n^2)$
 - Underlying problems are seriously hard

hard

Perspective



Remaining work

- Front end is solved
 - No more theses on parsing, I hope
- Middle has some room for work *(not a wide open frontier)*
 - New machines, new languages present new problems
- Back end has room for work
 - Complications from new machines
 - Longer latencies, wider processors