RICE

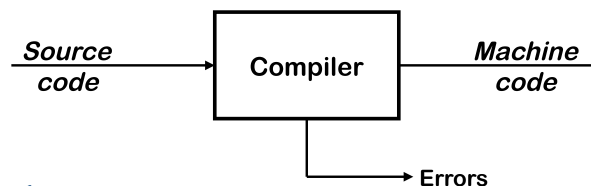
COMP 412
FALL 2011

The View from 35,000 Feet
Comp 412

This lecture has expanded to fill two classes.

Copyright 2011, Keith D. Cooper & 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.
Faculty from other educational institutions may use these materials for nonprofit educational purposes, provided this copyright notice is preserved.

High-level View of a Compiler

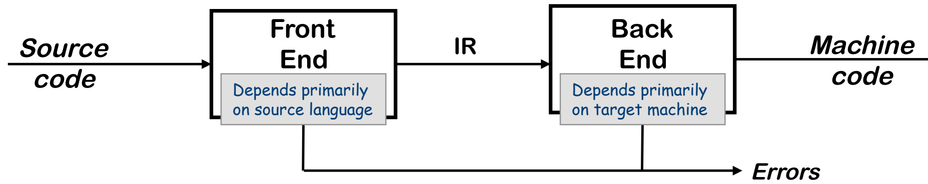


Implications

- Must recognize legal (and illegal) programs
- Must generate correct code
- Must manage storage of all variables (and code)
- Must agree with OS & linker on format for object code

Big step up from assembly language—use higher level notations

Traditional Two-pass Compiler



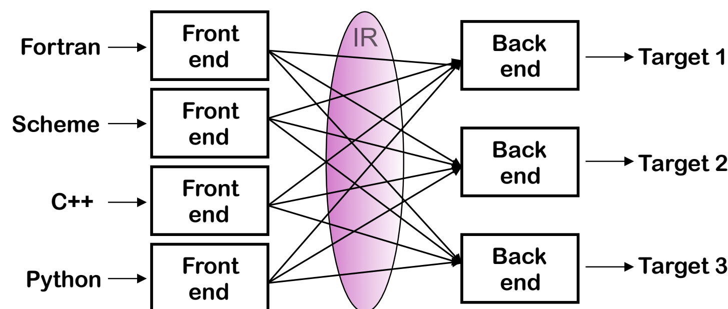
Implications

- Use an intermediate representation (IR)
- Front end maps legal source code into IR
- Back end maps IR into target machine code
- Admits multiple front ends & multiple passes *(better code)*

Classic principle from software engineering: Separation of concerns

Typically, front end is O(n) or O(n log n), while back end is NPC

A Common Fallacy



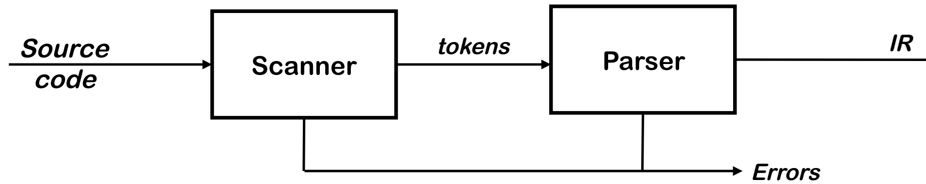
Can we build *n x m* compilers with *n+m* components?

- Must encode all language specific knowledge in each front end
- Must encode all features in a single IR
- Must encode all target specific knowledge in each back end

Some success in systems with assembly level (or lower) IRs

e.g., gcc's rtl or llvm ir

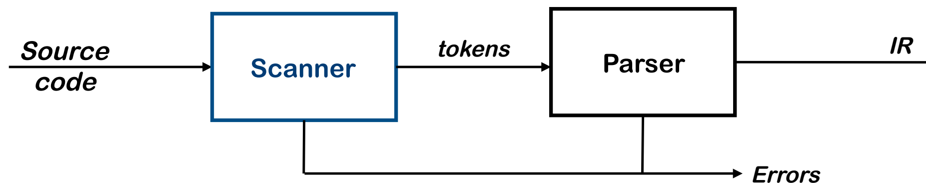
The Front End



Responsibilities

- Recognize legal (& illegal) programs
- Report errors in a useful way
- Produce IR & preliminary storage map
- Shape the code for the rest of the compiler
- Much of front end construction can be automated

The Front End



Scanner

- Maps character stream into words—the basic unit of syntax
- Produces pairs — a word & its part of speech
  x = x + y ; *becomes* <id,x> = <id,x> + <id,y> ;
  — *word ≅ lexeme, part of speech ≅ token type, pair ≅ a token*
- Typical tokens include *number*, *identifier*, +, –, new, *while*, *if*
- Speed is important

Textbooks advocate automatic scanner generation

Commercial practice appears to be hand-coded scanners

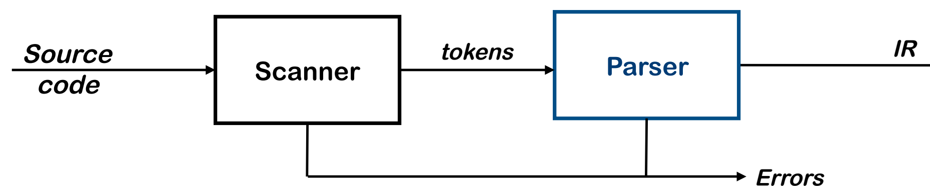
The Front End

The syntax recognized in a scanner is typically specified using the theory of regular languages

- While that sounds esoteric, the notation, called regular expressions (REs), that we use will seem familiar.
  - The "wildcard" character in file name matching is Linux shell
  - Same ideas are used in "net nanny" applications
- The theory leads to sound techniques for automatic generation of implementations.
  - Write a regular expression; tool generates a scanner
  - For a hand-coded scanner, I would still write the RE

Underlying theory is taught in 481 and used across CS

The Front End



Parser

- Recognizes context-free syntax & reports errors
- Guides context-sensitive ("semantic") analysis (*type checking*)
- Builds IR for source program

Hand-coded parsers are fairly easy to build

Most books advocate using automatic parser generators

In lab 2, you will build both a small hand-coded parser and a simple parser generator.

The Front End

Context-free syntax is specified with a grammar

SheepNoise → *SheepNoise* baa
| baa

This grammar defines the set of noises that a sheep makes under normal circumstances

It is written in a variant of Backus–Naur Form (BNF)

Formally, a grammar $G = (S,N,T,P)$

- S is the *start symbol*
- N is a set of *non-terminal symbols*
- T is a set of *terminal symbols* or *words*
- P is a set of *productions* or *rewrite rules* ($P : N \rightarrow N \cup T$)

(Example due to Dr. Scott K. Warren)

The Front End

Context-free syntax can be put to better use

1. *Goal* → *Expr*
2. *Expr* → *Expr Op Term*
3. | *Term*
4. *Term* → number
5. | id
6. *Op* → +
7. | -

S = *Goal*

T = { number, id, +, - }

N = { *Goal, Expr, Term, Op* }

P = { 1, 2, 3, 4, 5, 6, 7 }

- This grammar defines simple expressions with addition & subtraction over "number" and "id"
- This grammar, like many, falls in a class called "context-free grammars", abbreviated CFG

The grammar for a full language, such as Java, can contain hundreds of productions.

The Front End

Given a CFG, we can *derive* sentences by repeated substitution

Production	Result
	Goal
1	*Expr*
2	*Expr Op Term*
5	*Expr Op* y
7	*Expr* - y
2	*Expr Op term* - y
4	*Expr Op* 2 - y
6	*Expr* + 2 - y
3	*Term* + 2 - y
5	x + 2 - y

A derivation

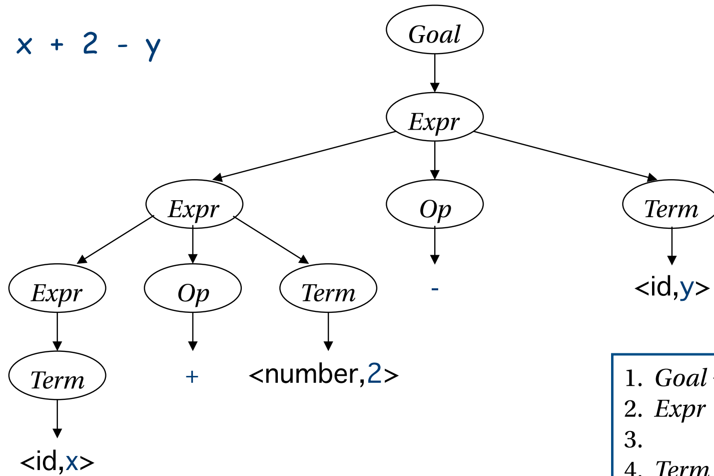
1. *Goal* → *Expr*
2. *Expr* → *Expr Op Term*
3. | *Term*
4. *Term* → number
5. | id
6. *Op* → +
7. | -

To recognize a valid sentence in some CFG, we reverse this process and build up a *parse*

The Front End

A parse can be represented by a tree (*parse tree* or *syntax tree*)

x + 2 - y

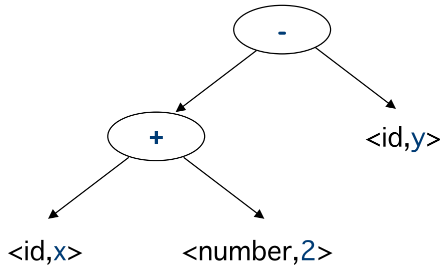


1. *Goal* → *Expr*
2. *Expr* → *Expr Op Term*
3. | *Term*
4. *Term* → number
5. | id
6. *Op* → +
7. | -

The parse tree contains a lot of unneeded information

The Front End

Compilers often use an *abstract syntax tree* instead of a parse tree



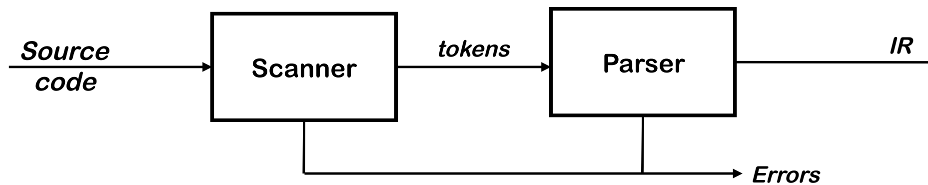
The AST summarizes grammatical structure, without including detail about the derivation

This is much more concise

ASTs are one kind of *intermediate representation* (IR)

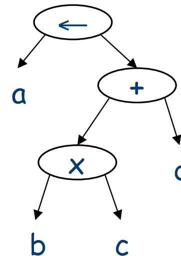
- Lisp s-expressions are, essentially, ASTs
- Some people think that the AST is the "natural" IR

The Front End



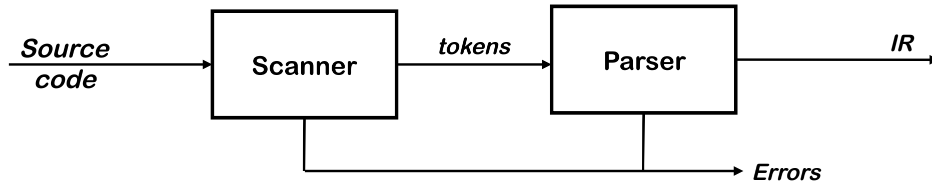
Code shape determines many properties of resulting program

a ← b x c + d



Recall the speed difference between different ways of writing a simple array initialization in Lecture 1

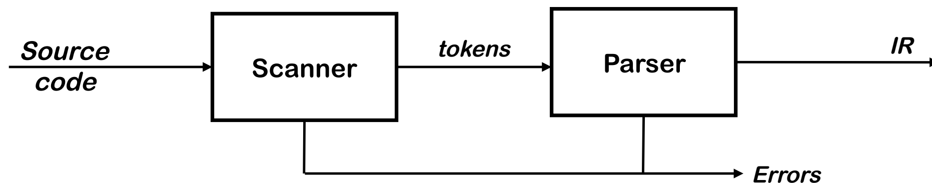
The Front End



The front end must generate IR that can be used by the optimizer and the back end to complete the translation process.

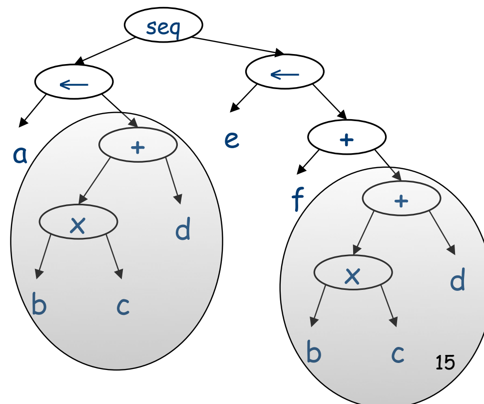
- Keep in mind that the point of the compiler is translation
- Front end performs initial stages of translation
  - Represents all of the computation
  - Needs some representation of the data (e.g., locations)
- This translation step determines, to a large extent, how the final code looks

The Front End



Code shape determines many properties of resulting program

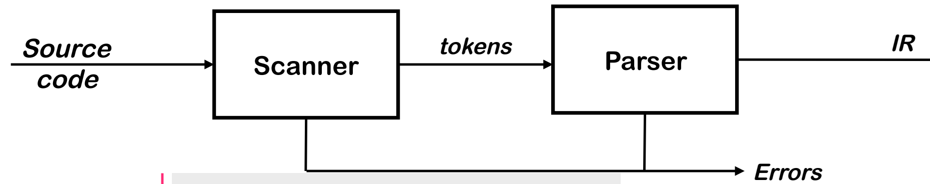
a ← b x c + d
e ← f + b x c + d



If you turn this AST into code, you will likely get duplication

For those of you who are EE's, this kind of translation artifact shows up in VHDL systems

The Front End



Is "a" distinct from b, c, & d ?

Code shape determines many properties of resulting program

a ← b x c + d
e ← f + b x c + d

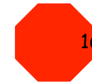
becomes

load @b ⇒ r_1
load @c ⇒ r_2
mult r_1,r_2 ⇒ r_3
load @d ⇒ r_4
add r_3,r_4 ⇒ r_5
store r_5 ⇒ @a
load @f ⇒ r_6
add r_5,r_6 ⇒ r_7
store r_7 ⇒ @e

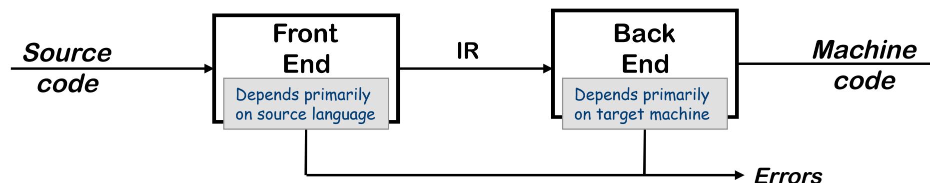
computes b x c + d (first six instructions)

reuses b x c + d (last three instructions)

We would like to produce this code, but getting it right takes a fair amount of effort ….



Traditional Two-pass Compiler



Implications

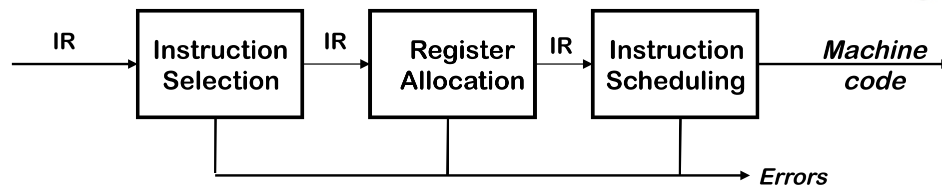
- Use an intermediate representation (IR)
- Front end maps legal source code into IR
- Back end maps IR into target machine code
- Admits multiple front ends & multiple passes (*better code*)

Classic principle from software engineering: Separation of concerns

Typically, front end is O(n) or O(n log n), while back end is NPC

2nd appearance for this slide.

The Back End

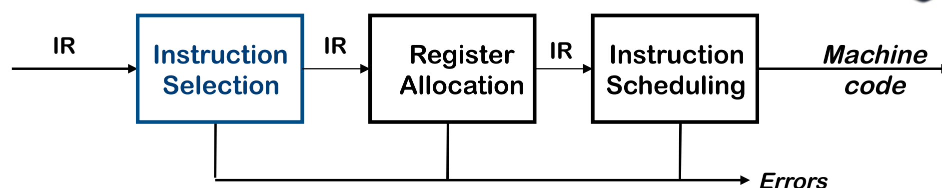


Responsibilities

- Translate IR into target machine code
- Choose instructions to implement each IR operation
- Decide which value to keep in registers
- Ensure conformance with system interfaces

Automation has been *less* successful in the back end

The Back End



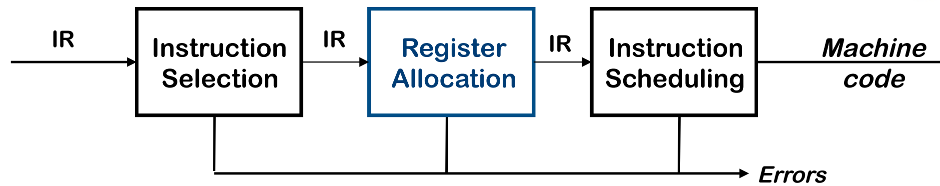
Instruction Selection

- Produce fast, compact code
- Take advantage of target features such as addressing modes
- Usually viewed as a pattern matching problem
  - *ad hoc* methods, pattern matching, dynamic programming
  - Form of the IR influences choice of technique

This was the problem of the future in 1978
  - Spurred by transition from PDP-11 to VAX-11
  - Orthogonality of RISC simplified this problem

Standard goal has become "locally optimal" code.

The Back End



Register Allocation

- Have each value in a register when it is used
- Manage a limited set of resources
- Can change instruction choices & insert LOADs & STOREs
- Optimal allocation is NP-Complete in most settings

Compilers approximate solutions to NP-Complete problems

You will become experts over next 3 weeks…

gcc 4.1, O3, PowerPC G4 target

The Back End

Local Register Allocation

→ An example that came to my attention in 2009 …

$a_0 \leftarrow a_1 + a_2$	$a_0 \leftarrow a_1 + a_2$
$a_1 \leftarrow a_2 + a_0$	$a_1 \leftarrow a_2 + a_0$
$a_2 \leftarrow a_0 + a_1$	$a_2 \leftarrow a_0 + a_1$
$a_3 \leftarrow a_1 + a_2$	$a_0 \leftarrow a_1 + a_2$
$a_4 \leftarrow a_2 + a_3$	$a_1 \leftarrow a_2 + a_0$
$a_5 \leftarrow a_3 + a_4$	$a_2 \leftarrow a_0 + a_1$
…	…
This block allocates into 30 registers on the PowerPC G4	This block allocates into 20 registers on the PowerPC G4 and takes more time to run

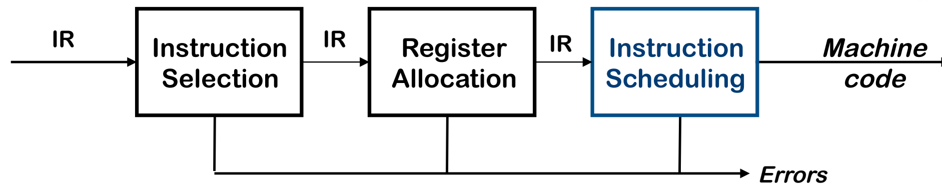
What is the difference?

- Naming
  - Block on left reuses names & confuses its notion of live range

The point of this example is that the allocator confused the distinct live ranges with the same name (e.g., a_0) with the result that it saw the problem as more constrained that it should have been. As a result it began spilling when it had at least 10 spare registers.

Renaming the "virtual registers" so that each definition point targets a unique name would avoid this problem.

The Back End



Instruction Scheduling

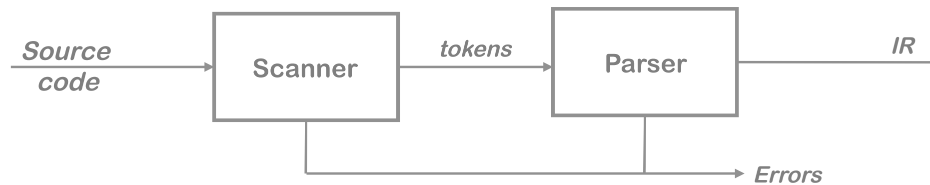
- Avoid hardware stalls and interlocks
- Use all functional units productively
- Can increase lifetime of variables (changing the allocation)

Optimal scheduling is NP-Complete in nearly all cases

Heuristic techniques are well developed

In Lab 3, you will build a pair of schedulers.

The Front End



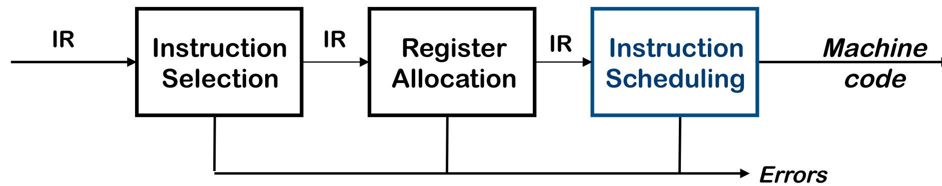
$a \leftarrow b \times c + d$
$e \leftarrow f + b \times c + d$

load @b $\Rightarrow r_1$
load @c $\Rightarrow r_2$
mult $r_1, r_2 \Rightarrow r_3$
load @d $\Rightarrow r_4$
add $r_3, r_4 \Rightarrow r_5$
store $r_5 \Rightarrow$ @a
load @f $\Rightarrow r_6$
add $r_5, r_6 \Rightarrow r_7$
store $r_7 \Rightarrow$ @e

(first six lines) computes $b \times c + d$; (last three lines) reuses $b \times c + d$

Recall this example ILOC program from earlier in the lecture?

The Back End



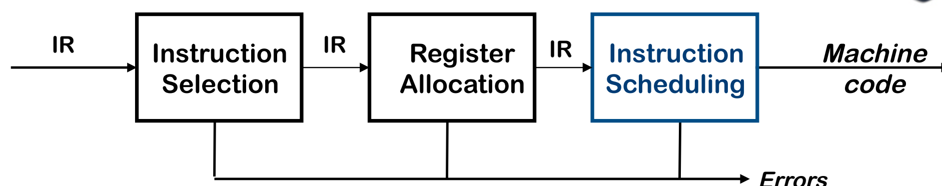
Instruction Scheduling

unit 1	unit 2
load @b ⇒ r_1	load @c ⇒ r_2
load @d ⇒ r_4	load @f ⇒ r_6
mult r_1,r_2 ⇒ r_3	nop
add r_3,r_4 ⇒ r_5	nop
store r_5 ⇒ @a	nop
add r_5,r_6 ⇒ r_7	nop
store r_7 ⇒ @e	nop

This schedule aggressively loads values into registers to cover the memory latency.

It finishes the computation as soon as possible (assuming 2 cycles for load & store, 1 cycle for other operations.

The Back End



Instr [covered] load)

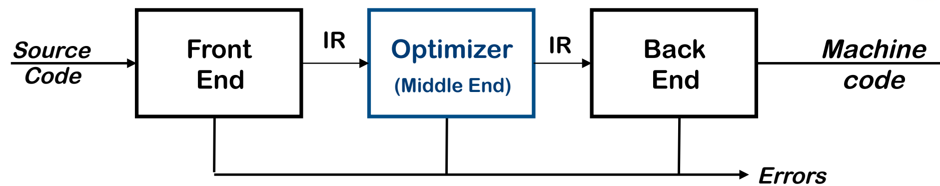
This schedule needs fewer registers but the code is no slower

unit 1	unit 2
load @b ⇒ r_1	load @c ⇒ r_2
load @d ⇒ r_4	load @f ⇒ r_6
mult r_1,r_2 ⇒ r_3	nop
add r_3,r_4 ⇒ r_5	nop
store r_5 ⇒ @a	nop
add r_5,r_6 ⇒ r_7	nop
store r_7 ⇒ @e	nop

unit 1	unit 2
load @b ⇒ r_1	load @c ⇒ r_2
load @d ⇒ r_4	nop
mult r_1,r_2 ⇒ r_3	nop
add r_3,r_4 ⇒ r_5	load @f ⇒ r_6
store r_5 ⇒ @a	nop
add r_5,r_6 ⇒ r_7	nop
store r_7 ⇒ @e	nop

Same set of names, fewer of them are simultaneously live

Traditional Three-part Compiler

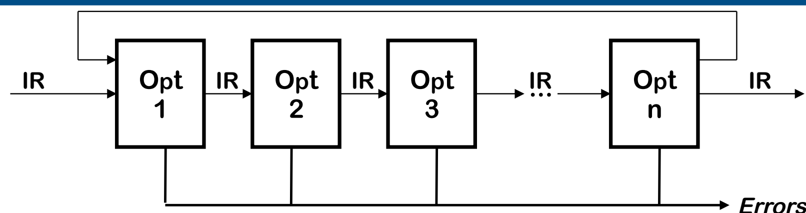


Code Improvement (or *Optimization*)

- Analyzes IR and rewrites (or *transforms*) IR
- Primary goal is to reduce running time of the compiled code
  - May also improve space, power consumption, …
- Must preserve "meaning" of the code
  - Measured by values of named variables

Subject of COMP 512, 515, maybe final weeks of 412

The Optimizer (or Middle End)



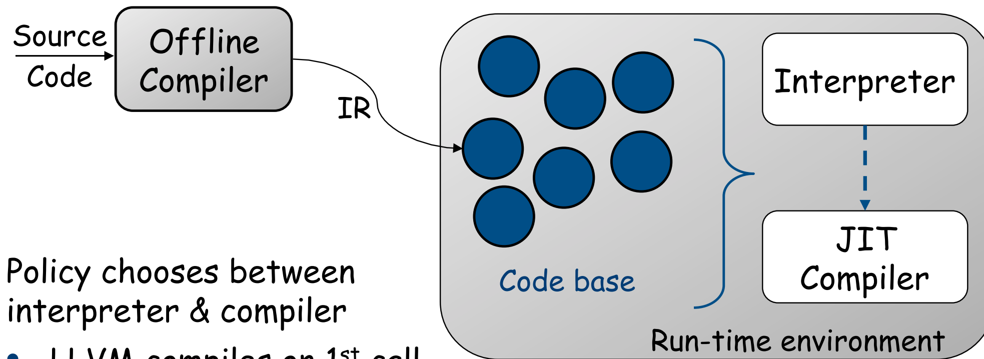
Modern optimizers are structured as a series of passes

Typical Transformations

- Discover & propagate some constant value
- Move a computation to a less frequently executed place
- Specialize some computation based on context
- Discover a redundant computation & remove it
- Remove useless or unreachable code
- Encode an idiom in some particularly efficient form

Run-time Compilation

Systems such as HotSpot, Jalapeno, and Dynamo deploy compiler and optimization techniques *at run-time*



Policy chooses between interpreter & compiler

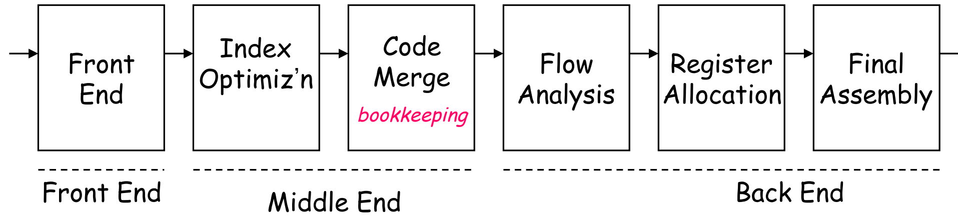
- LLVM compiles on 1st call
- Dynamo optimizes on 50th execution
- HotSpot uses a heuristic to decide

Role of the Run-time System

- Memory management services
  - Allocate
    - → In the heap or in an activation record (*stack frame*)
  - Deallocate
  - Collect garbage
- Run-time type checking
- Error processing
- Interface to the operating system
  - Input and output
- Support of parallelism
  - Parallel thread initiation
  - Communication and synchronization

Classic Compilers

1957: The FORTRAN Automatic Coding System

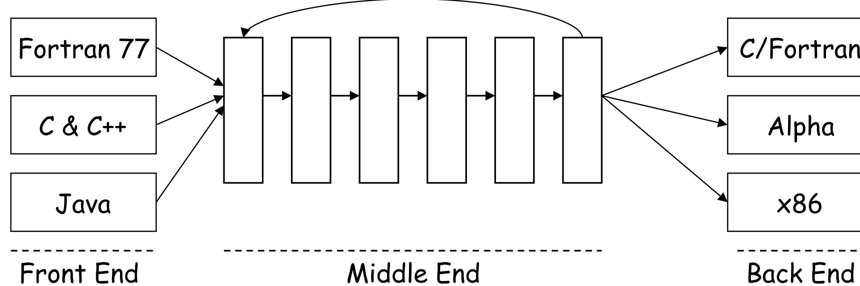


- Six passes in a fixed order
- Generated good code
  - Assumed unlimited index registers
  - Code motion out of loops, with ifs and gotos
  - Did flow analysis & register allocation

This compiler is widely recognized as the first commercial compiler. It fits the description.

Classic Compilers

1999: The SUIF Compiler System

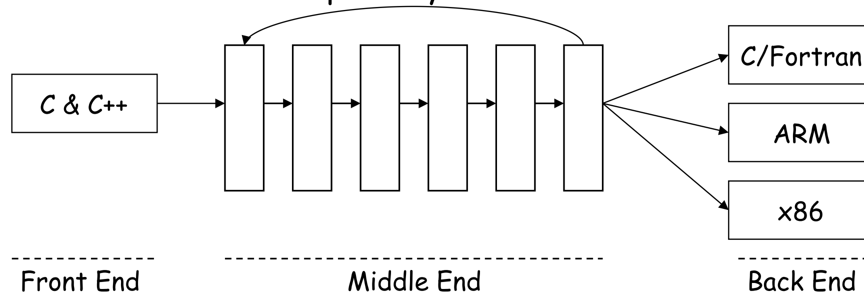


Another classically-built compiler

- 3 front ends, 3 back ends
- 18 passes, configurable order
- Two-level IR (High SUIF, Low SUIF)
- Influential piece of research infrastructure

Classic Compilers

2005: The LLVM Compiler System



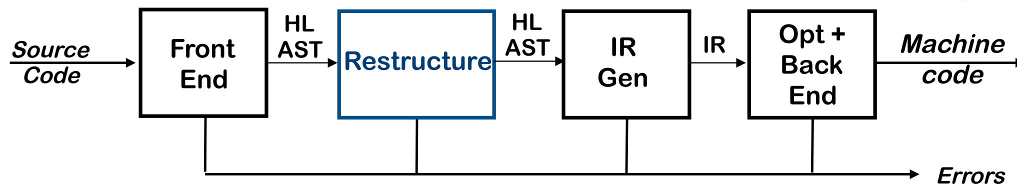
Often touted as replacement for gcc

- Many optimization passes
- Linear IR in SSA form
- Back ends for a growing collection of machines
- Decent compiler used in both academia and industry

Next Class

- Introduction to Local Register Allocation
- Announcements:
  - Specs for Lab 1 available Monday(8/29/2011)
    - → Due Sept 14 (Lab report due 1 day later)
    - → Practice blocks and simulator will be available
    - → Grading blocks will be hidden from you

Modern Restructuring Compiler



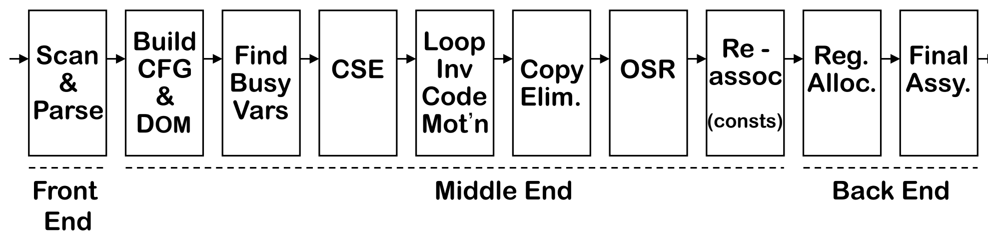
Typical Restructuring Transformations:

- Blocking for memory hierarchy and register reuse
- Vectorization
- Parallelization
- All based on dependence
- Also full and partial inlining

Subject of COMP 515

Classic Compilers

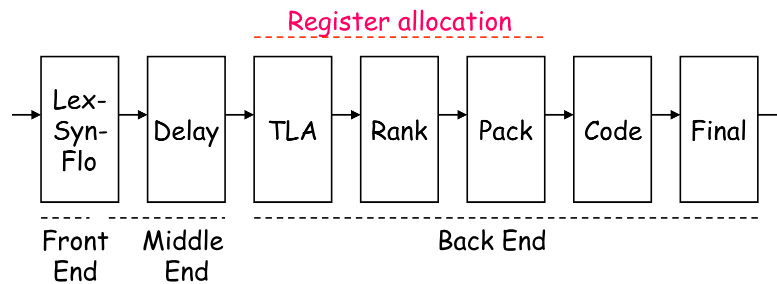
1969: IBM's FORTRAN H Compiler



- Used low-level IR (quads), identified loops with dominators
- Focused on optimizing loops ("inside out" order)
  Passes are familiar today
- Simple front end, simple back end for IBM 370

Classic Compilers

1975: BLISS-11 compiler (Wulf *et al.*, CMU)

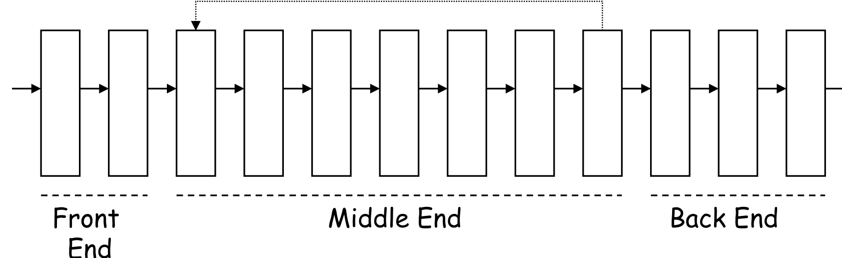


- The great compiler for the PDP-11
- Seven passes in a fixed order
- Focused on code shape & instruction selection
  - LexSynFlo did preliminary flow analysis
  - Final included a grab-bag of peephole optimizations

Basis for early VAX & Tartan Labs compilers

Classic Compilers

1980: IBM's PL.8 Compiler



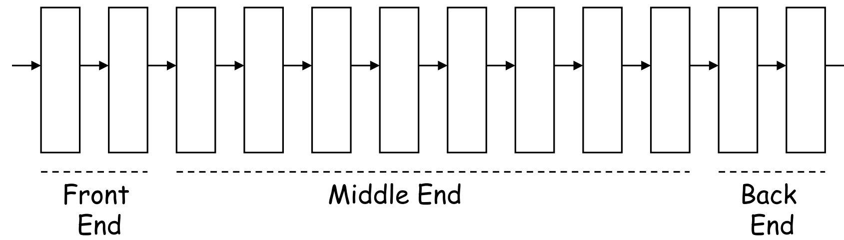
- Many passes, one front end, several back ends
- Collection of 10 or more passes
  - Repeat some passes and analyses
  - Represent complex operations at 2 levels
  - Below machine-level IR

Dead code elimination
Global cse
Code motion
Constant folding
Strength reduction
Value numbering
Dead store elimination
Code straightening
Trap elimination
Algebraic reassociation

Multi-level IR has become common wisdom

Classic Compilers

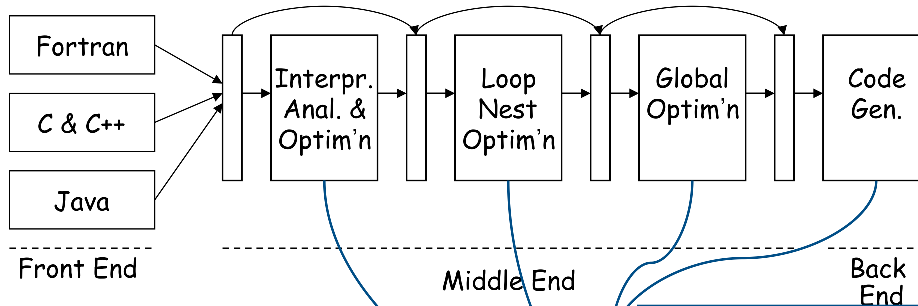
1986: HP's PA-RISC Compiler



- Several front ends, an optimizer, and a back end
- Four fixed-order choices for optimization (9 passes)
- Coloring allocator, instruction scheduler, peephole optimizer

Classic Compilers

2000: The SGI Pro64 Compiler (now Open64)

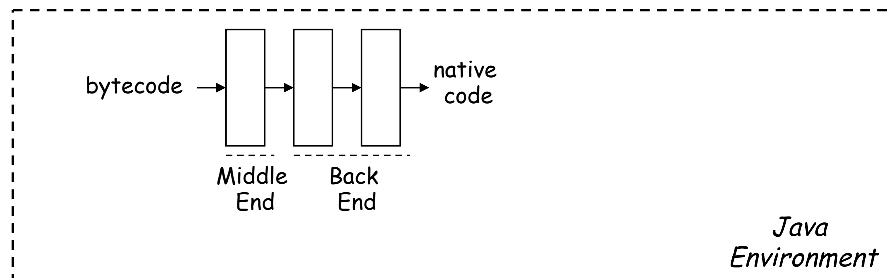


Open source compiler for IA 64

- 3 front ends, 1 back end
- Five-levels of IR
- Gradual lowering of abstraction level

Classic Compilers

Even a modern JIT fits the mold, albeit with fewer passes



- Front end tasks are handled elsewhere
- Few (if any) optimizations
  - Avoid expensive analysis
  - Emphasis on generating native code
  - Compilation must be a priori profitable