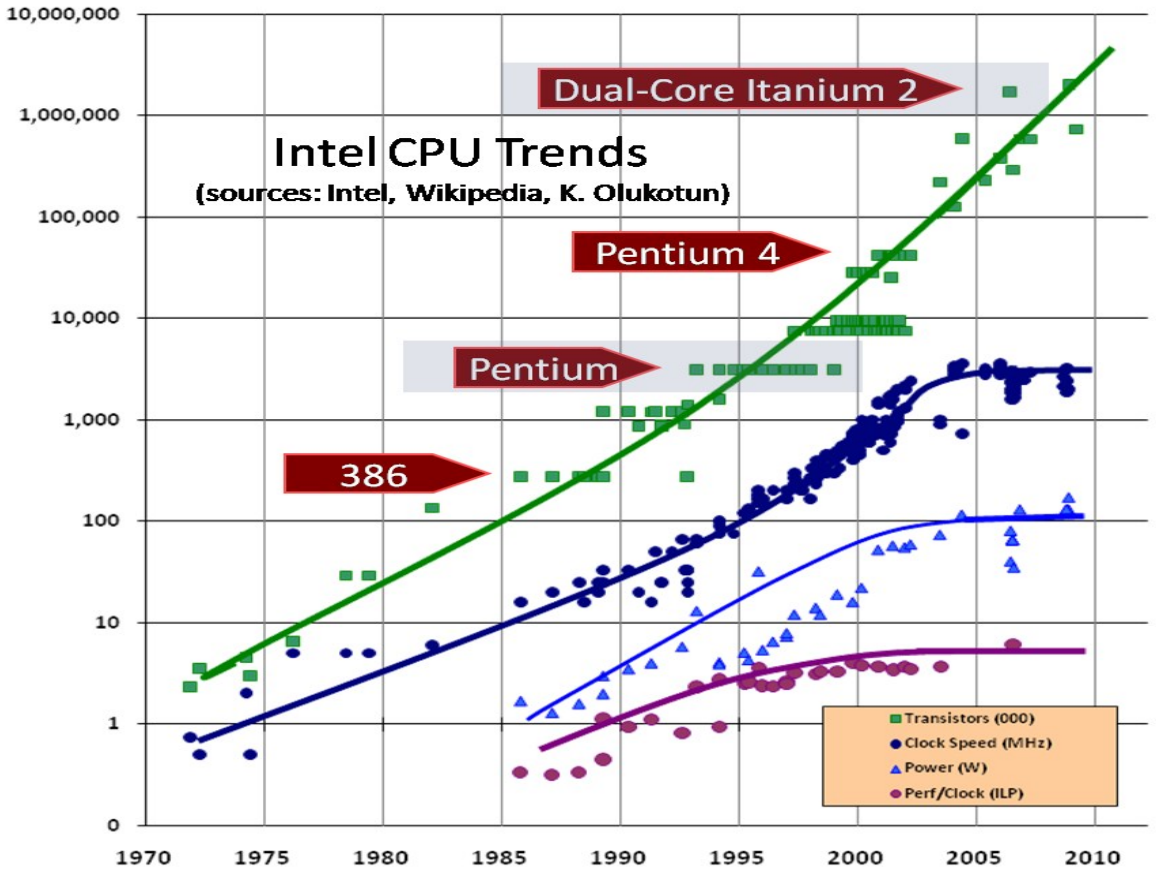
Learning Objectives

- **Identify the reasons why we study compiler design**
- List the components of a traditional 2-pass compiler/3-pass and their functionality
- Provide a brief summary of the history of compilers and the state of the art today

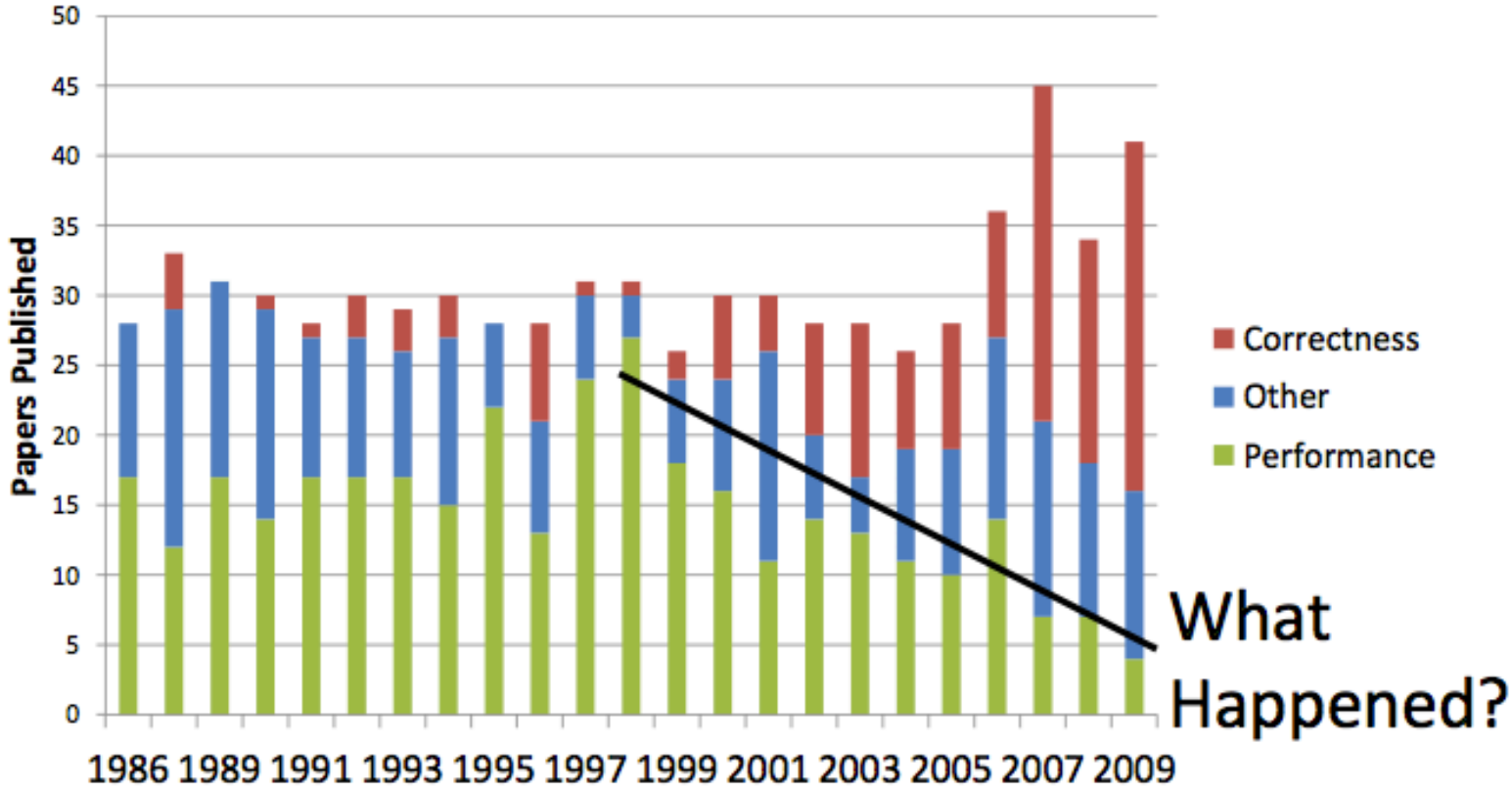
Why study compilers - 1

- Single core performance has plateuaed – need to extract performance through parallelism (i.e., parallel code)



Why study compilers - 2

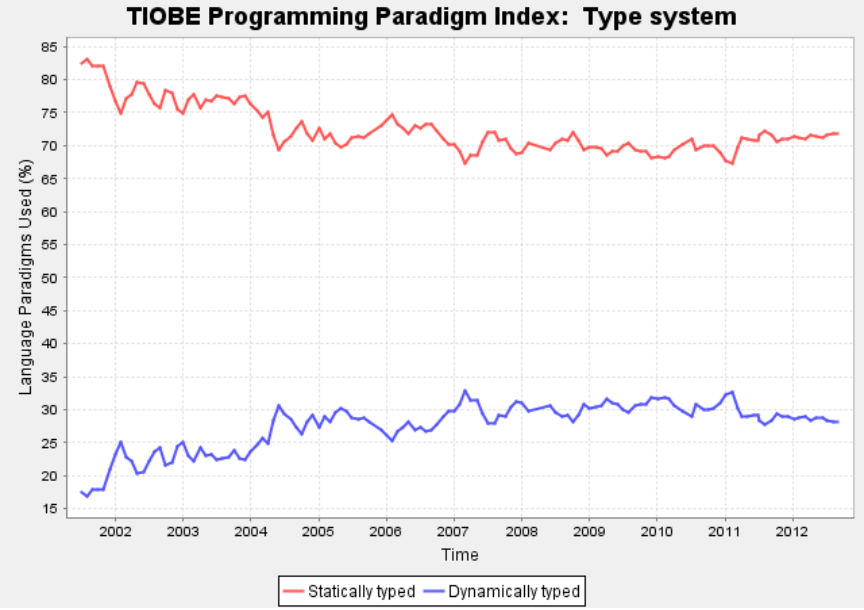
- Static analysis for reliability and security violations

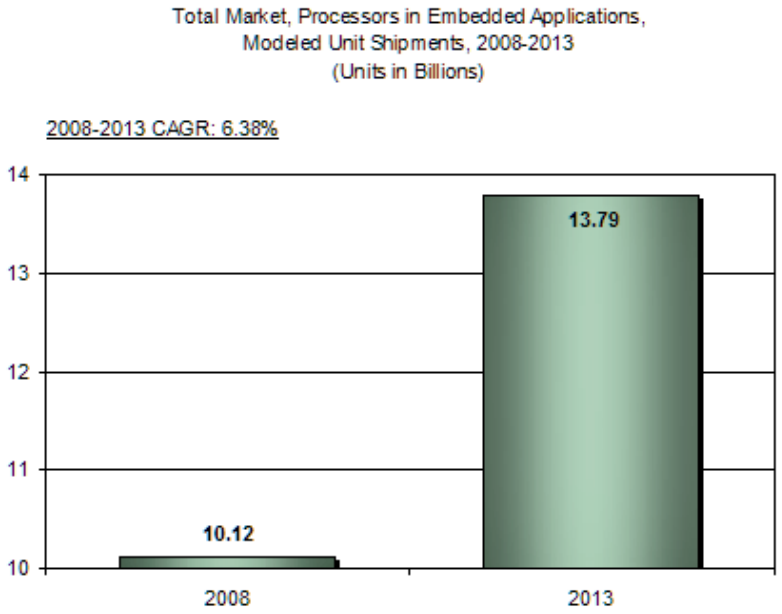


Slide courtesy: Ben Zorn, MSR

Why study compilers - 3

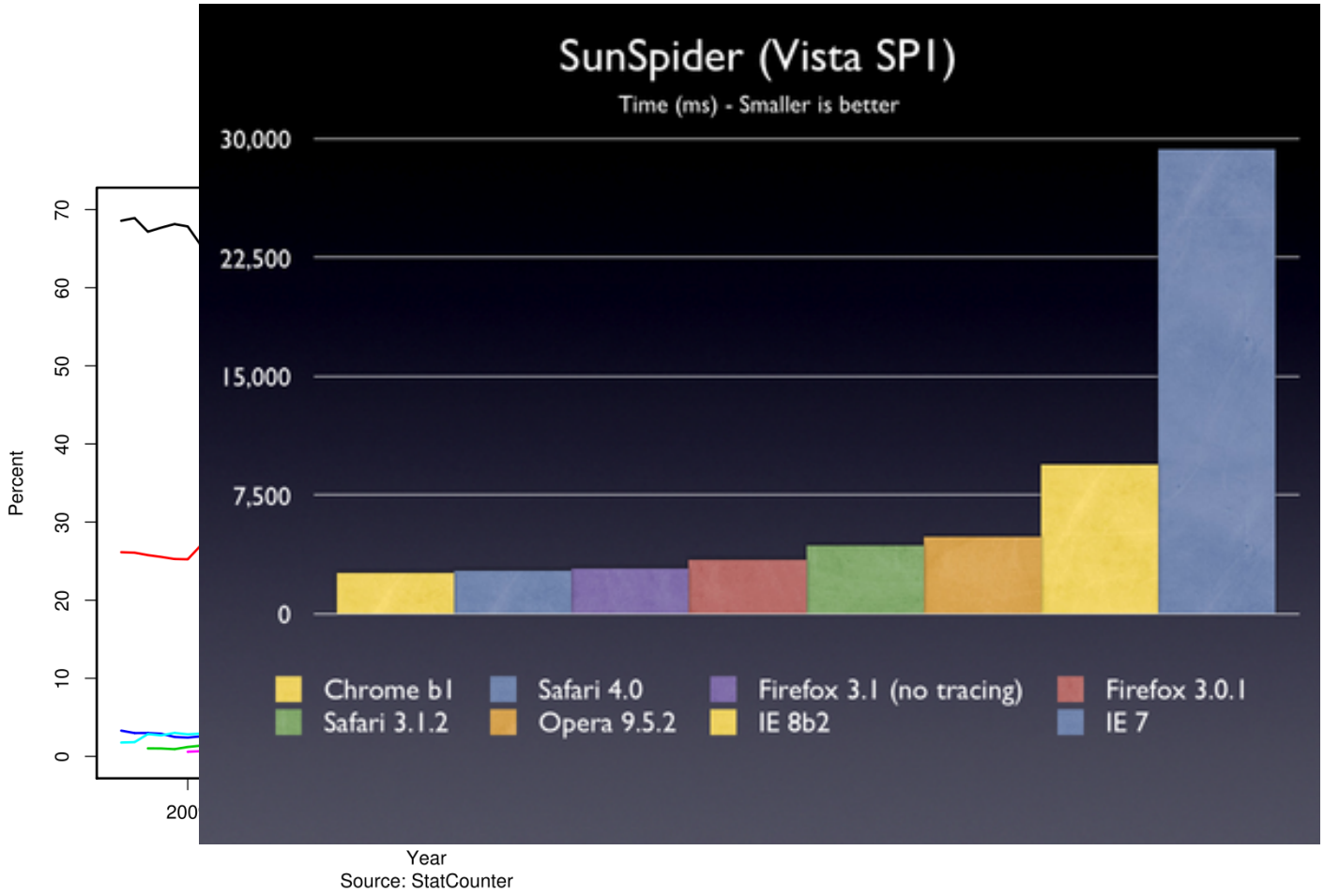
- New languages that are dynamic – challenging to compile
- Embedded applications – need specialized compilers (energy, space, cost are the constraints)





Why Study Compilers - 4

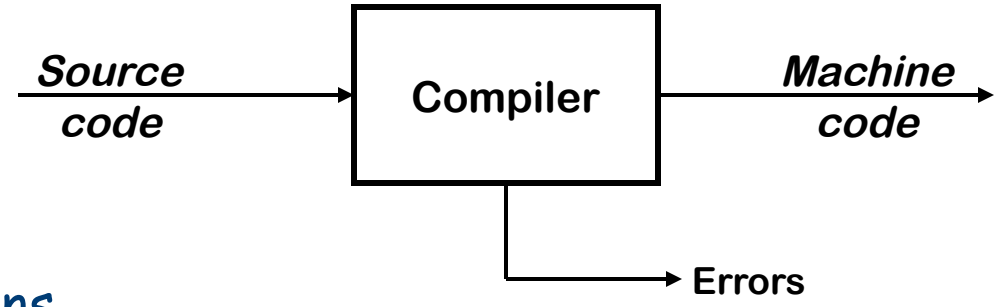
- What is the main factor influencing browser adoption ?



Learning Objectives

- Identify the reasons why we study compiler design
- **List the components of a traditional 2-pass compiler/3-pass and their functionality**
- Provide a brief summary of the history of compilers and the state of the art today

High-level View of a Compiler

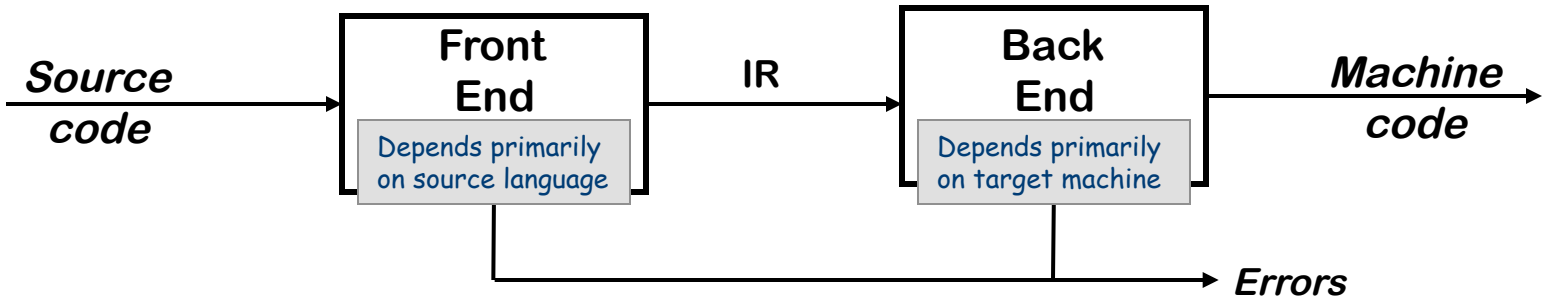


Implications

- 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



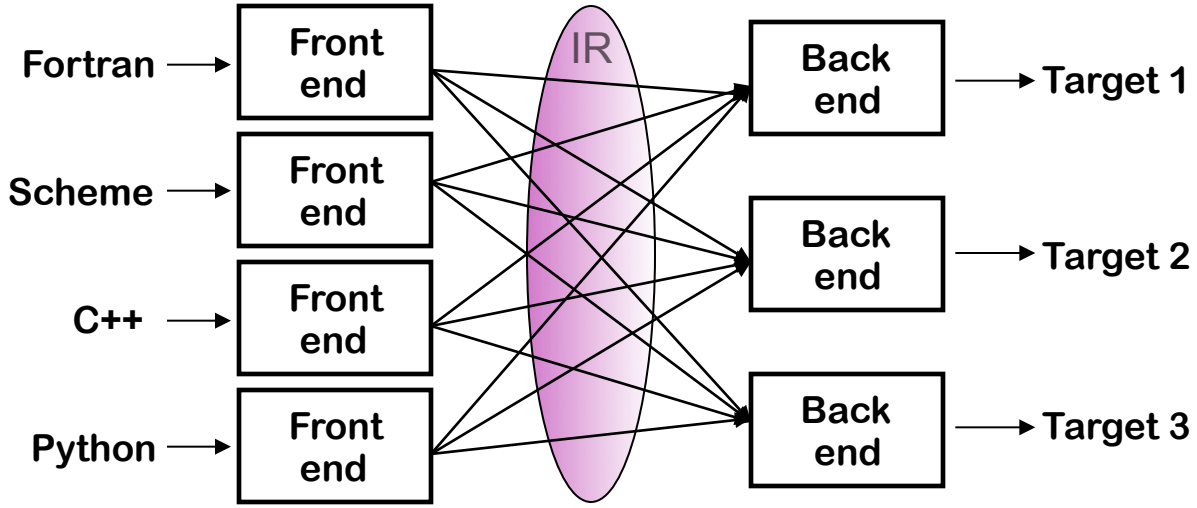
Implications

- 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



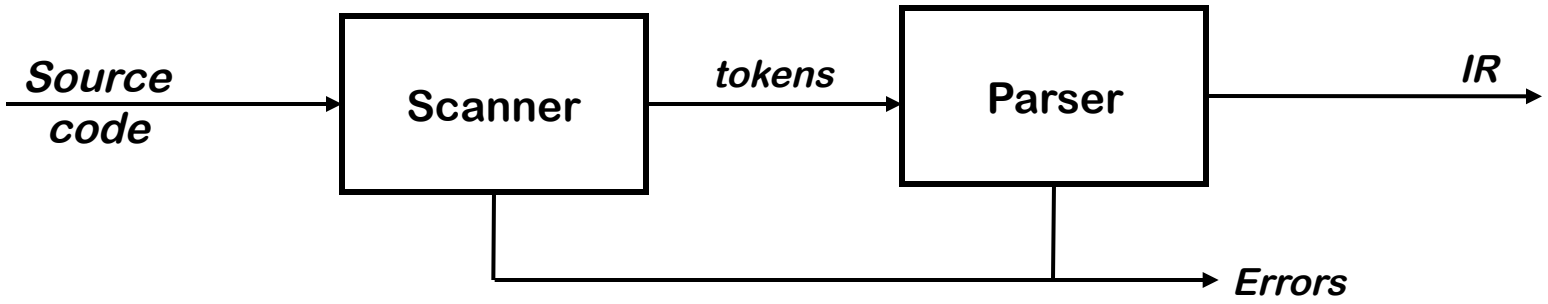
Can we build *n x m* compilers with *n+m* components?

- 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

Successful in systems with assembly level (or lower) IRs

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

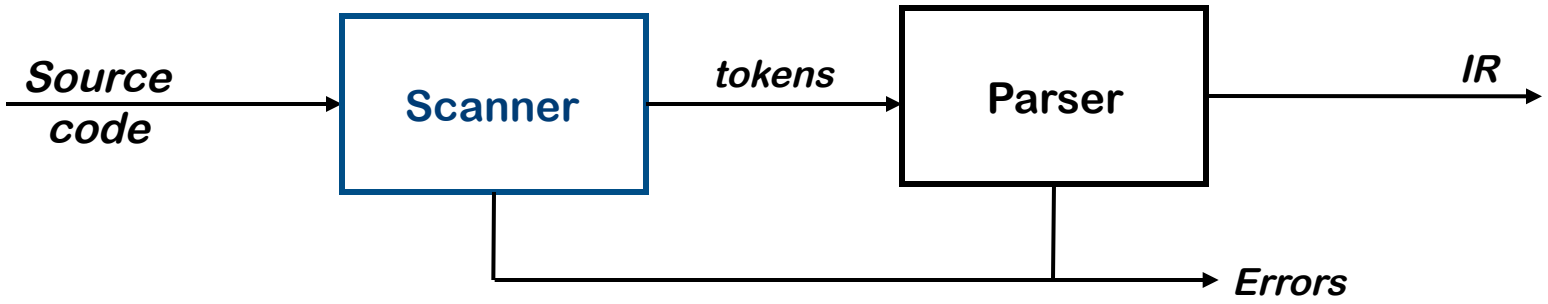
The Front End



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



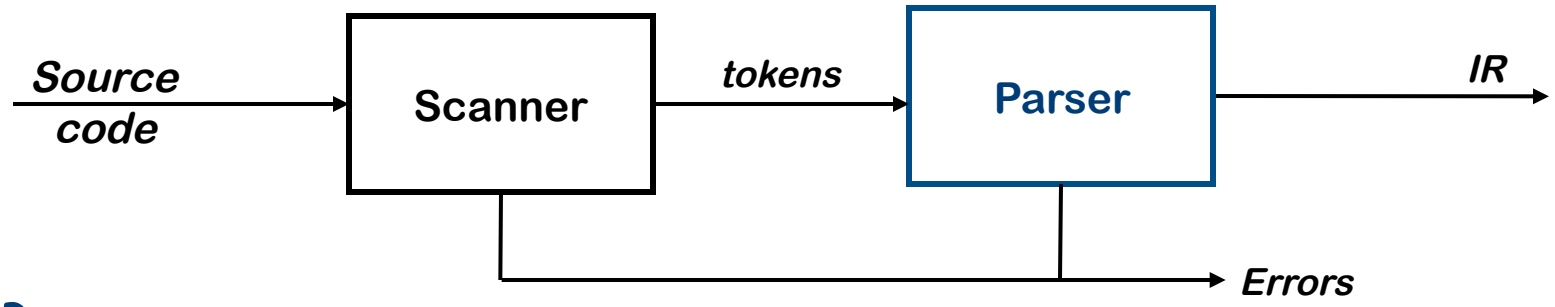
Scanner

- 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



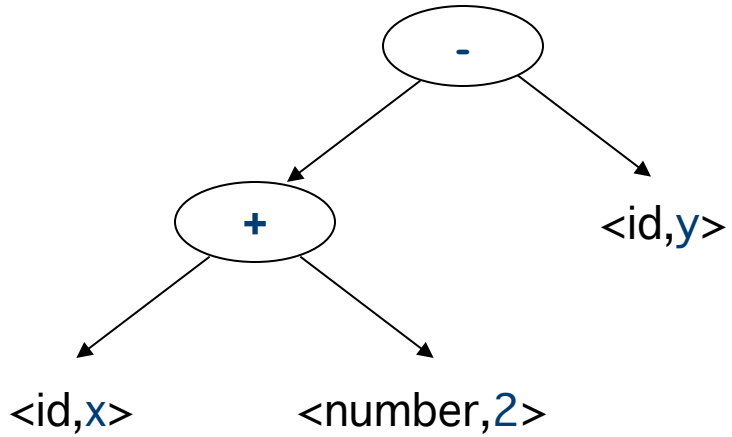
Parser

- Recognizes context-free syntax & reports errors
- Guides context-sensitive ("semantic") analysis (*type checking*)
- Builds IR for source program (e.g., Abstract Syntax Tree)

Hand-coded parsers are fairly easy to build

Most books advocate using automatic parser generators

Abstract Syntax Tree (AST)



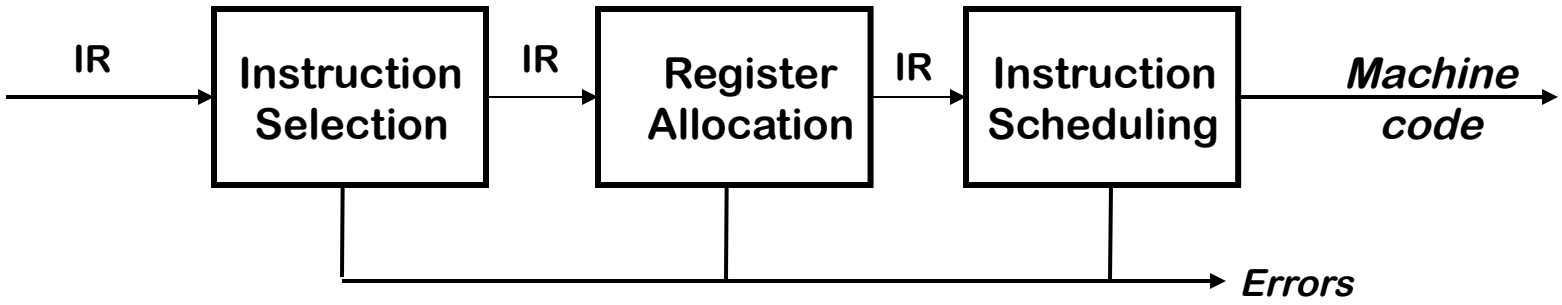
The AST summarizes grammatical structure, without including detail about the derivation

This is much more concise

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

Some people think that the AST is the "natural" IR.

The Back End

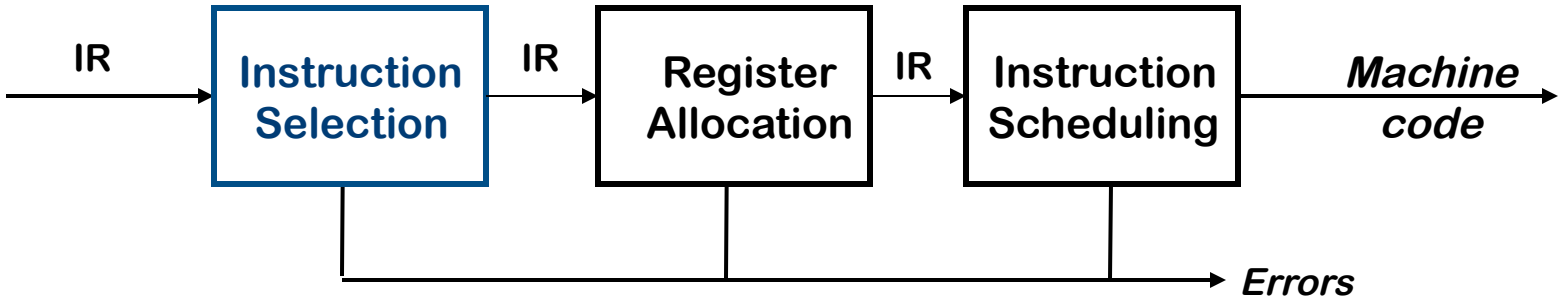


Responsibilities

- 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

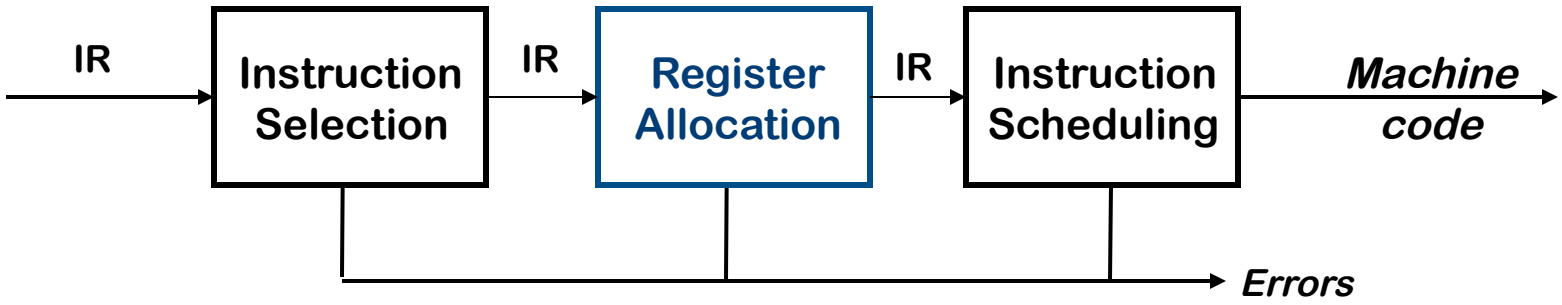
- Produce fast, compact code
- Take advantage of target features such as addressing modes
- Usually viewed as a pattern matching problem
  - *ad hoc* methods, pattern matching, dynamic programming
  - Form of the IR influences choice of technique

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

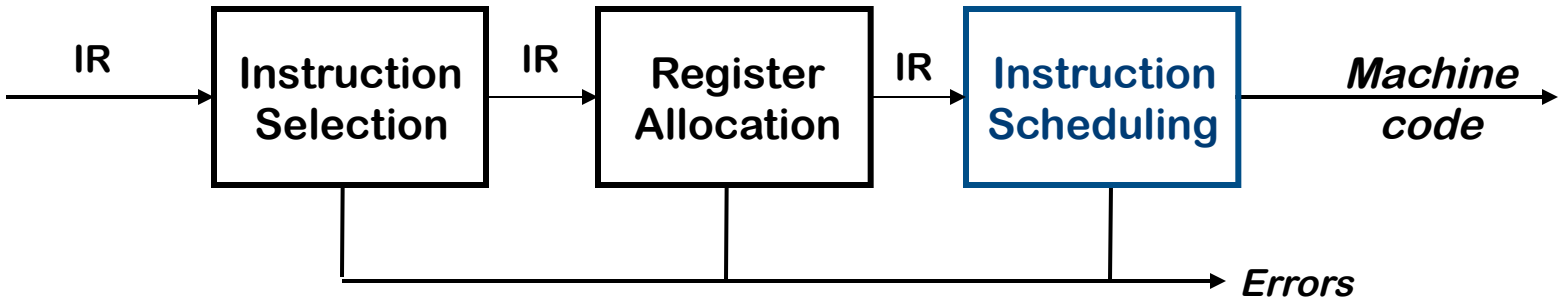


Register Allocation

- 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

The Back End



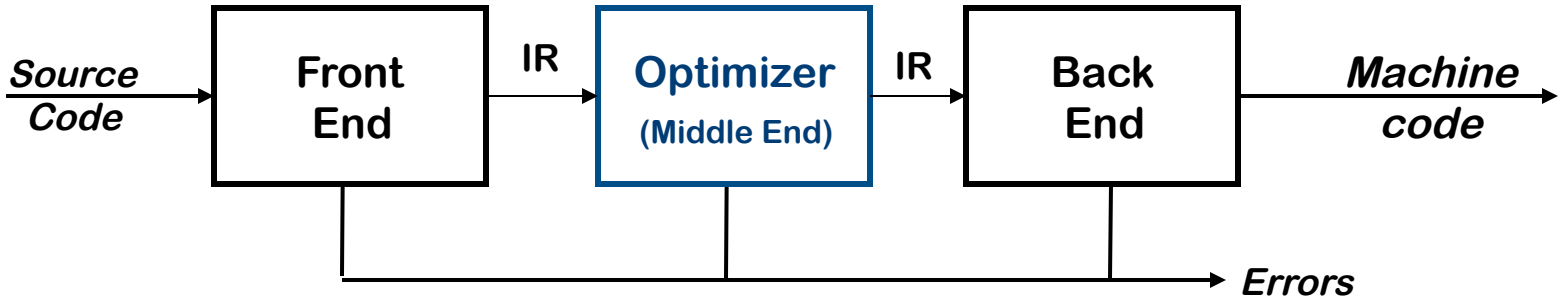
Instruction Scheduling

- 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

Traditional Three-part Compiler

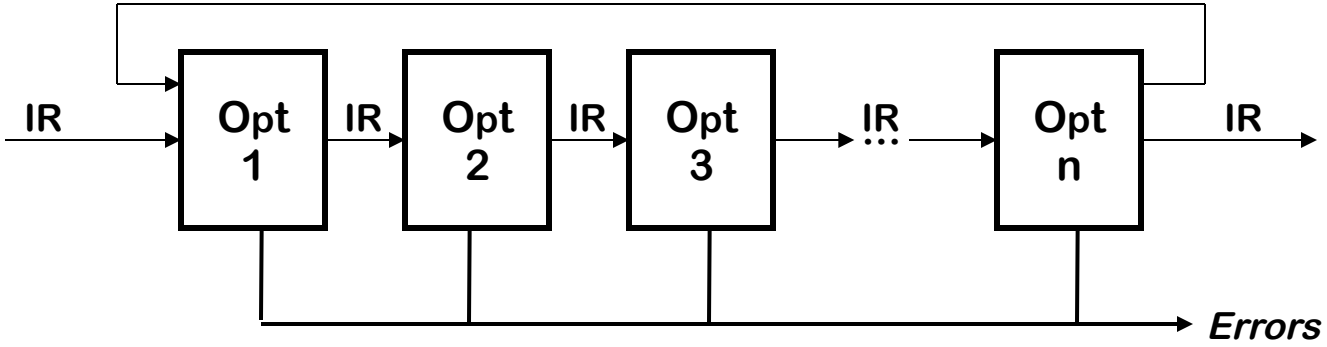


Code Improvement (or *Optimization*)

- 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 this course

The Optimizer (or Middle End)

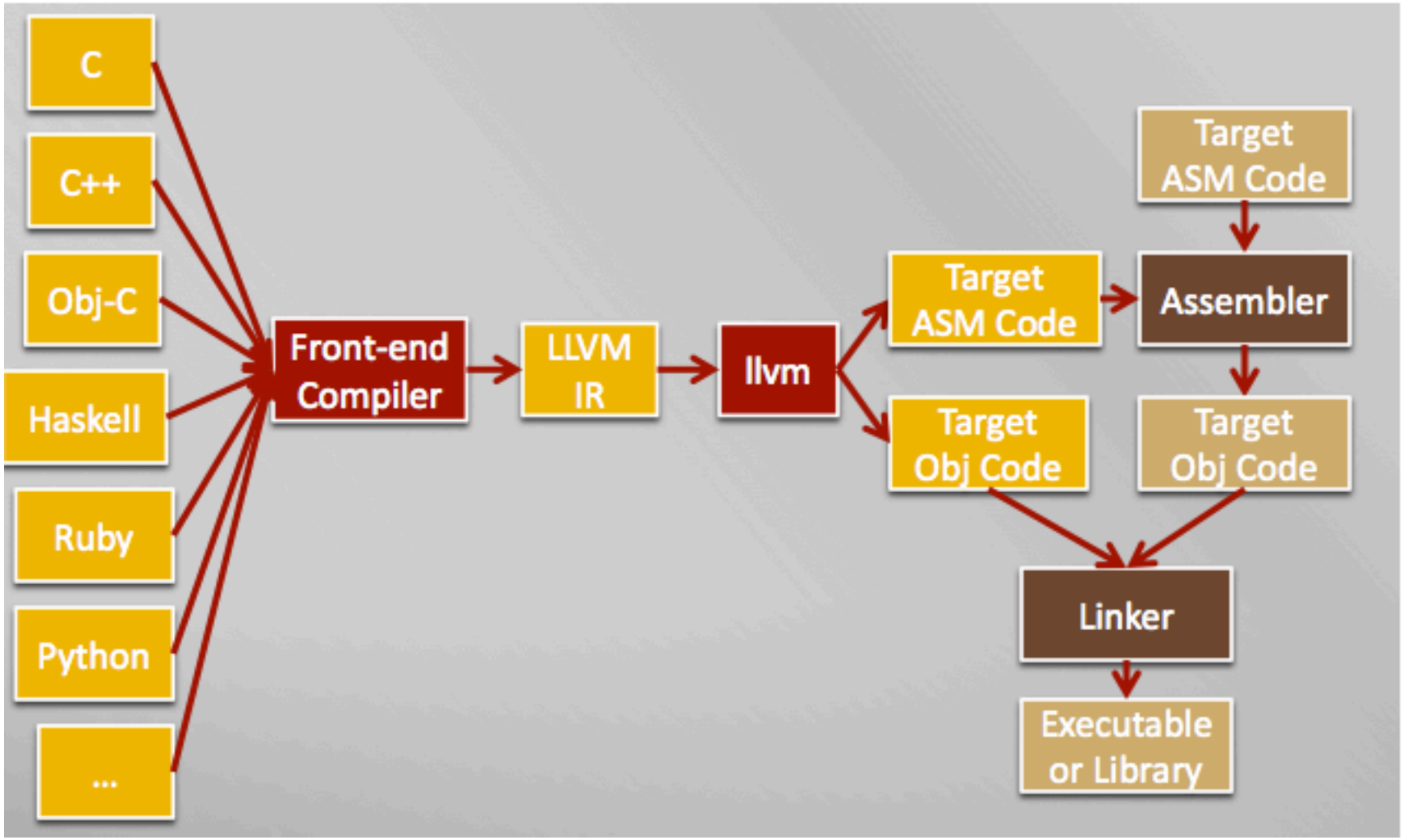


Modern optimizers are structured as a series of passes

Typical Transformations

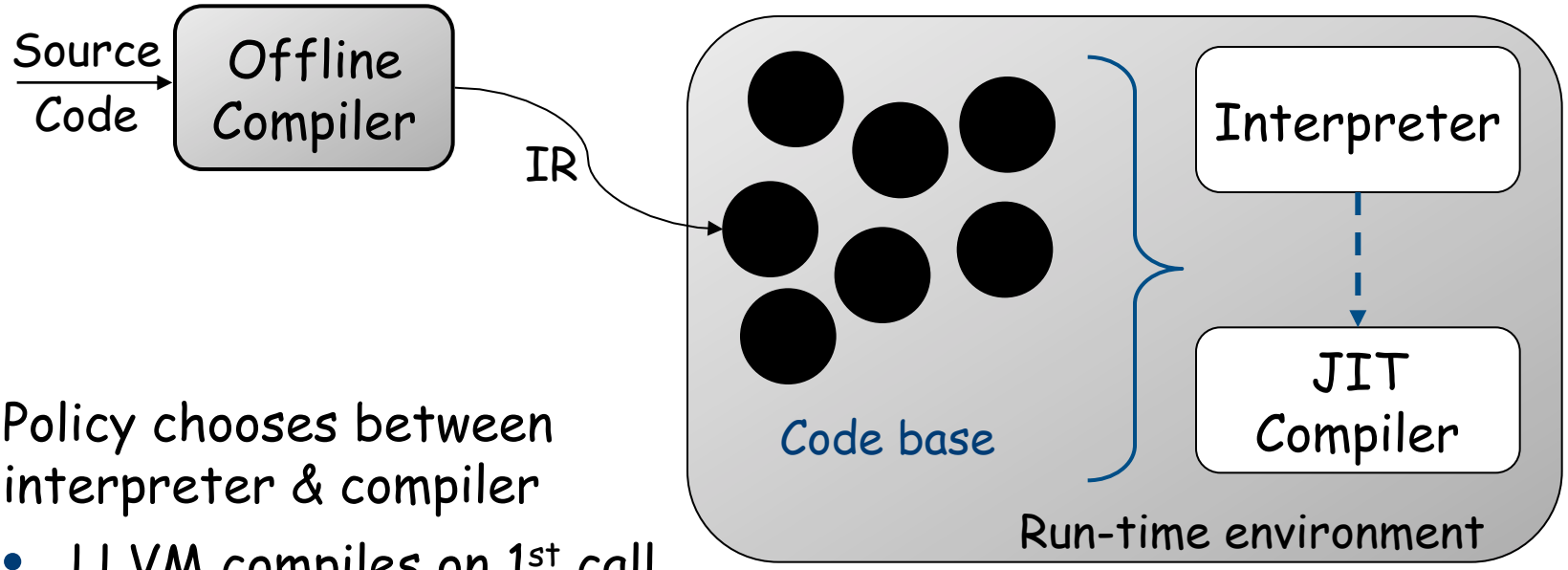
- Discover & propagate some constant value
- Move a computation to a less frequently executed place
- Specialize some computation based on context
- Discover a redundant computation & remove it
- Remove useless or unreachable code
- Encode an idiom in some particularly efficient form

LLVM Compiler Structure



Run-time Compilation

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

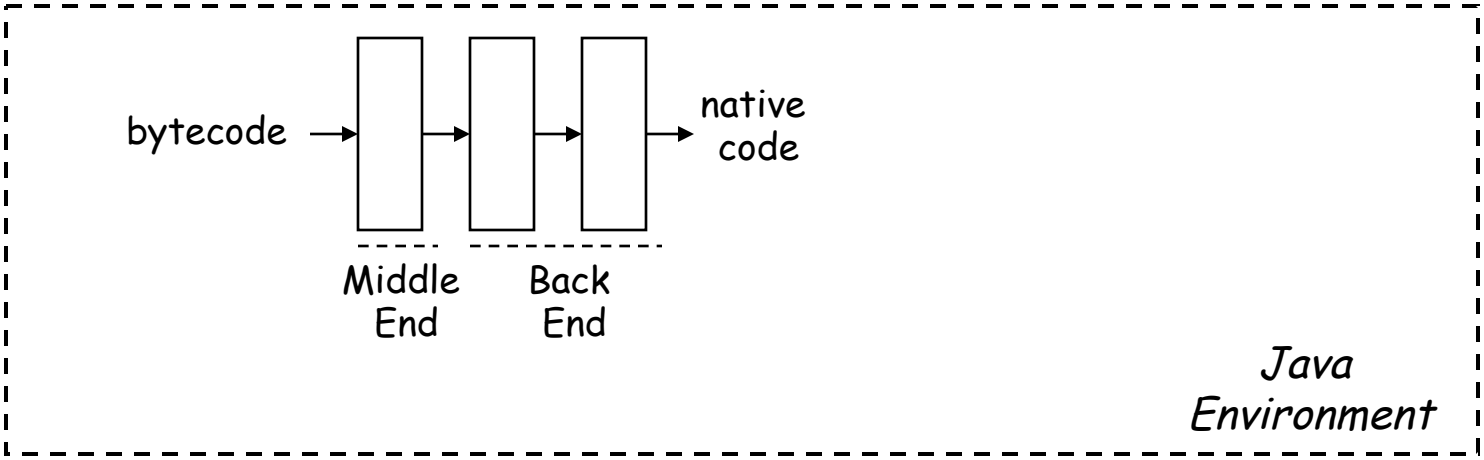


Policy chooses between interpreter & compiler

- LLVM compiles on 1st call
- Dynamo optimizes on 50th execution

JIT 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

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

Learning Objectives

- Identify the reasons why we study compiler design
- List the components of a traditional 2-pass compiler/3-pass and their functionality
- **Provide a brief summary of the history of compilers and the state of the art today**

First FORTRAN compiler

- John Backus on the Fortran Compiler (around 1958)

- "It is our belief that if FORTRAN, during its first months, were to translate any reasonable scientific program into an object program only half as fast as its hand-coded counterpart, then acceptance of our system would be in serious danger. ...
- "To this day I believe that our emphasis on object program efficiency rather than on language design was basically correct. I believe that had we failed to produce efficient programs, the widespread use of languages like FORTRAN would have been seriously delayed."

- – John Backus, Fortran I, II and III, *Annals of the History of Computing,* vol. 1, no. 1, July 1979.

A Sense of History

1955-1959	*Commercial compilers generated good code*
Fortran	Separation of concerns (Backus, 1956)
Cobol	Control-flow graph, register allocation (Haibt, 1957)
1960–1964	*Academics try to catch up with industrial trade secrets*
Algol 60	Early algorithms for "code generation" (1960, 1961)
	Relating theory to practice (Lavrov, 1962)
	Alpha project at Novosibirsk (Ershov, 1963 & 1965)
1965-1969	*Technology begins to spread*
PL/I	Fortran H (Medlock & Lowry, 1967)
Algol 68	Value numbering (Balke, 1967 ?)
Simula 67	Literature begins to emerge (Allen, 1969)

A Sense of History

1970-1974	*The literature explodes and optimization grows up*
SETL	Cocke & Schwartz, Allen-Cocke Catalog, 1971
Smalltalk	Theory of analysis (Kildall, 1971, Allen & Cocke, 1972)
Lisp	Interprocedural analysis (Spillman,1972)
APL	Strength reduction, dead code elimination, Live (SETL)
	Expression tree algorithms (Sethi, Aho & Ullman)
1975-1979	*Global optimization comes of age*
Pascal	Full literature of data-flow analysis
CLU	Strength reduction (Cocke & Kennedy, 1977)
Alphard	Partial redundancy elimination (Morel & Renvoise, 1979)
Com. Lisp	Inline substitution studies (Scheiffler, 1977, Ball, 1979)
	Tail recursion elimination (Steele, 1978)
	Data dependence analysis (Bannerjee, 1979)

A Sense of History

1980-1984	*Programming environments and new architectures*
Smalltalk80	Incremental analysis (Reps, 1982; Ryder, Zadeck, 1983)
ADA	Incremental compilation (Schwartz *et al.*, 1984)
Scheme	Interprocedural analysis (Myers, 1981; Cooper, 1984)
	RISC compilers (PL.8, 1980; MIPS, 1983)
	Graph coloring allocation (Chaitin, 1981; Chow, 1983)
	Vectorization (Wolfe, 1982; R. Allen, 1983)
1985–1989	*Resurgence of interest in classical optimization*
C++	Constant propagation (Wegman & Zadeck,Torczon,1985)
ML	Code motion of control structures (Cytron *et al.*, 1986)
Modula-3	Value numbering (Alpern *et al.*, Rosen *et al.*, 1988)
	Software pipelining (Lam, Aiken & Nicolau, 1988)
	Pointer analysis (Ruggeri, 1988)
	SSA-form (Cytron *et al.*, 1989)

A Sense of History

1990-1994	*Architects (and memory speed) drive the process*
Fortran 90	Hierarchical allocation (Koblenz & Callahan,1991)
	Scalar replacement (Carr 1991)
	Cache blocking (Wolf, 1991)
	Prefetch placement (Mowry, 1992)
	Commercial interprocedural compilers (Convex, 1992)
1995–1999	*The internet & SSA both come of age*
Java	JIT compilers (Everyone, from 1996 to present)
Perl (?)	Code compression (Franz, 1995; Frasier et al., 1997; ...)
	SSA-based formulations of old methods (lots of papers)
	Compile to VM (Java, 1995; Bernstein, 1998; …)
	Memory layout optimizations (Smith, 19??; others …)
	Widespread use of analysis (pointers, omega test, …)

The state of compilers today: My subjective view

- 2000 to 2005: Expansion beyond classical optimization
  - Dawson Engler's work on finding bugs in the Linux Kernel (2001)
  - Work by Manuvir Das on protocol violation errors (2002)
  - SLAM at MSR for model checking of C code (2001)
  - MOPS by David Wagner for security property checking (2001)
  - Findbugs by William Pugh for bug pattern detection (2004)

- 2005 to 2010: Program synthesis, energy/power, Web 2.0
  - Program sketching by Solar Lezama et al (2005, 2006)
  - Program synthesis by Gulwani (2007-2014)
  - Failure oblivious computing by Rinard (2005- 2014)
  - Fault-tolerance by Berger, Pattabiraman, Zorn (2005-2014)
  - Energy savings through relaxed correctness – Sankaralingam, Chillimbi, Pattabiraman, Grossman, Ceze (2009 to 2014)
  - Fast compilation of JavaScript programs (2010 onwards)
  - Analysis of programs for continuity (Gulwani and Chaudhri)