RUTGERS
THE STATE UNIVERSITY
OF NEW JERSEY

CS415 Compilers
Overview of the Course

These slides are based on slides copyrighted by Keith Cooper, Ken Kennedy & Linda Torczon at Rice University

RUTGERS Welcome to CS415 - Compilers

- Instructor: Ulrich Kremer (uli@cs.rutgers.edu)
  - → My Office Hours via webex/zoom: TBD
  - → Lectures 1, 2, and 3: online
- Teaching Assistant
  - → TA: Jonathan Garcia-Mullen
  - → **Recitations & office hours will start next week**
- Textbooks:
  - → Required: Engineering a Compiler by Cooper/Torcson
  - → Recommended: New (or old) Dragon Book
- Web Site:

  http://www.cs.rutgers.edu/courses/415/classes/spring_2022_kremer/
  - → Project descriptions, handouts, homeworks, lecture slides, …
  - → lec01.pdf vs. lec01mod.pdf
- canvas.rutgers.edu and piazza
  - → Homework and project questions, grades, homework solutions
- SPN or prerequisits overrides: Please send me an email

Basis for Grading (tentative)

- Exams
  - → Midterm 25%
  - → Final 35%
- Homeworks 10%
- Projects (tentative!)
  - → Back End 10%
  - → Front End 10%
  - → Code generator/optimizer 10%

Cheating and Honor Code.

Notice: *This grading scheme and projects are tentative and subject to change.*

Basis for Grading

- Exams
  - → Midterm
  - → Final
  - ✦ Midterm in class
  - ✦ Final is cumulative (scheduled final exam)
- Homework
  - ✦ Reinforce concepts, provide practice
  - ✦ Number of assignments t.b.d.
- Projects
  - → Back End
  - → Front End
  - → Code generator/opt.
  - ✦ High ratio of thought to programming
  - ✦ Single student labs (note academic integrity information)

Rough Syllabus

- Overview § 1
- Instruction Scheduling § 12
- Local Register Allocation § 13
- Scanning § 2
- Parsing § 3
- Context Sensitive Analysis § 4
- Inner Workings of Compiled Code § 6, 7
- Introduction to Optimization § 8
- Code Selection § 11
- More Optimization (*time permitting*)
  - → Advanced topics in language design/compilation:
    automatic parallelization, GPUs, power & energy, quantum computing

Class-taking technique for CS 415

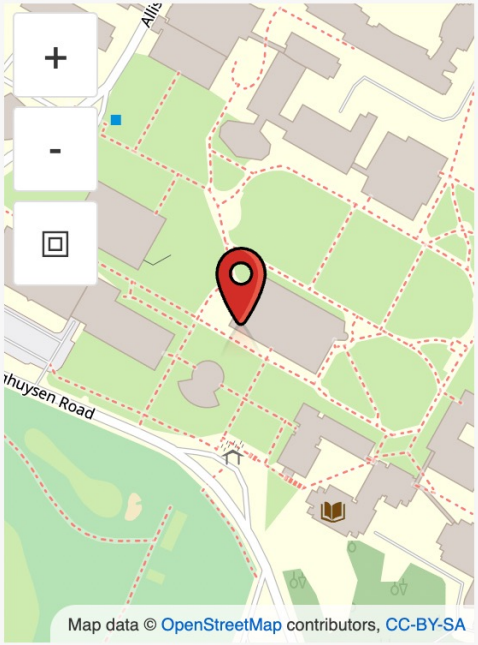
- Lectures in person in SEC 118
- Get an ilab account NOW, if you not have one already.
- I will use slides
  - → I will moderate my speed, *you* sometimes need to say "STOP"
- You should read the book
  - → Not all material will be covered in class
  - → Book complements the lectures
- You are responsible for material from class
  - → The midterm and final will cover both, lectures and readings
- CS 415 is not a programming course
  - → Projects are graded on functionality, documentation, and project reports more than style. However, things should be reasonable
- Use the resources provided to you
  - → See me or the TA in office hours if you have questions
  - → Post questions regarding homework and projects on piazza
- Email personal issues to me with subject line starting "cs415:"

Our Lecture Hall

Science & Engineering Resource Center - Room 118

Classroom Features

Classroom Map



Campus: Busch
Address: 118 Frelinghuysen Road, Piscataway, NJ 08854-8019
Seating Chart: PDF

About Textbooks

- Our required text book is available for free online
  - → Available through our course website
  - → Available on sakai/Resources
- Other recommended textbook is the "Dragon Book"
  - → Aho, Lam, Sethi, Ullman: Compilers - Principles, Techniques, and Tools (2nd edition)
  - → Older version (Dragon book) also fine

Compilers

- What is a compiler?
  - → *A program that translates an executable program in one language into an executable program in another language*
  - → *A good compiler should improve the program, in some way*
- What is an interpreter?
  - → *A program that reads an executable program and produces the results of executing that program*

- *C is typically compiled, Scheme is typically interpreted*
- Java is compiled to bytecode (code for the Java VM)
  - → which is then interpreted
  - → Or a hybrid strategy is used
    - Just-in-time compilation
    - Dynamic optimization (hot paths)

Why Study Compilation?

- Compilers are important system software components
  - → They are intimately interconnected with architecture, systems, programming methodology, and language design
- Compilers include many applications of theory to practice
  - → Scanning, parsing, static analysis, instruction selection
- Many practical applications have embedded languages
  - → Commands, macros, …
- Many applications have input formats that look like languages,
  - → Matlab, Mathematica
- Writing a compiler exposes practical algorithmic & engineering issues
  - → Approximating hard problems; efficiency & scalability
  - → No free lunch, i.e., there are multi-dimensional tradeoffs

Intrinsic interest

- Compiler construction involves ideas from many different parts of computer science

Artificial intelligence	Greedy algorithms Heuristic search techniques Machine learning
Algorithms	Graph algorithms, union-find, dynamic programming, approximations
Theory	DFAs & PDAs, pattern matching, fixed-point algorithms
Systems	Allocation & naming, synchronization, data locality
Architecture	Pipeline & hierarchy management, instruction set use, parallelism, quantum computing

Intrinsic merit

- Compiler construction poses challenging and interesting problems:
  - → Compilers must do a lot but also run fast
  - → Compilers have primary responsibility for run-time performance
  - → Compilers are responsible for making it acceptable to use the full power of the programming language
  - → Computer architects perpetually create new challenges for the compiler by building more complex machines (e.g.: multi-core, GPUs, FPGAs, quantum computers, neuromorphic processors)
  - → Compilers must/should hide that complexity from the programmer
  - → Success requires mastery of complex interactions

Making Languages Usable

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

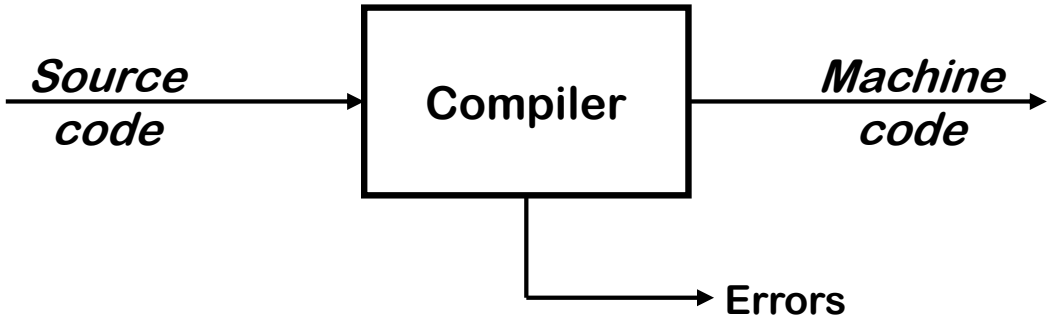
— John Backus

About the instructor

- My own research
  - → Compiling for homogeneous parallel architectures
  - → Compiler-directed power and energy management
  - → Programming models and languages for dynamic networks of mobile devices (SpatialViews, Sarana)
  - → Resource-aware security enforcement (establishing trust)
  - → Programming models and languages for Autonomous Underwater Vehicles (AUVs)
  - → Programming models for program quality/resources tradeoffs
  - → Programming models for non-von Neumann machines (e.g. quantum computing, reversible computing, neuromorphic computing)

- Thus, my interests lie in
  - → Interplay between compiler, OS, and architecture
  - → Static analysis to discern program behavior
  - → Run-time performance analysis
  - → Physical measurements (power/energy, performance, memory)
  - → New ways of thinking about computation

High-level View of a Compiler

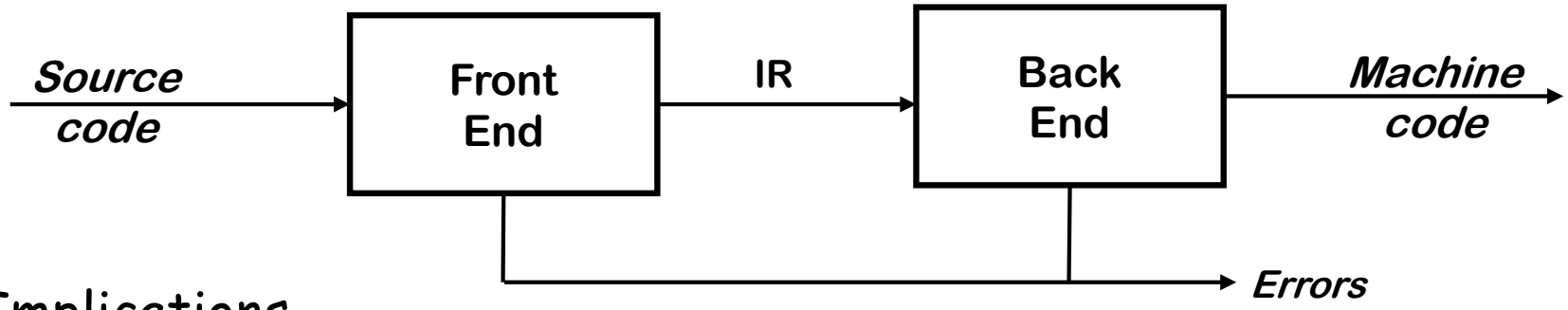


Implications

- 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

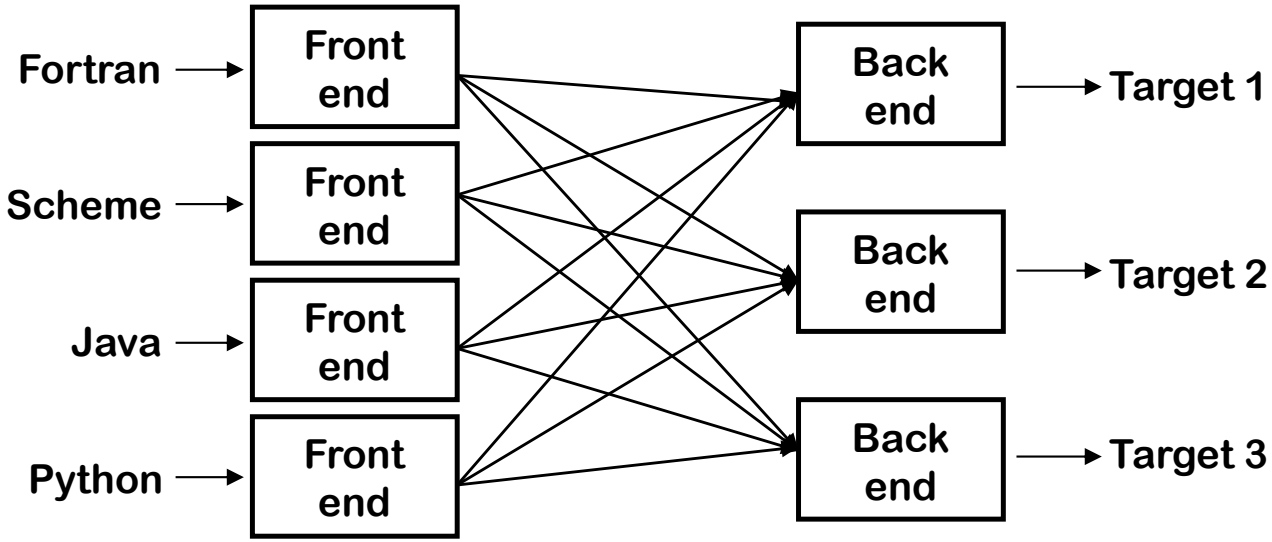


Implications

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

Typically, front end is O(n) or O(n log n), while back end is NP-complete

A Common Fallacy

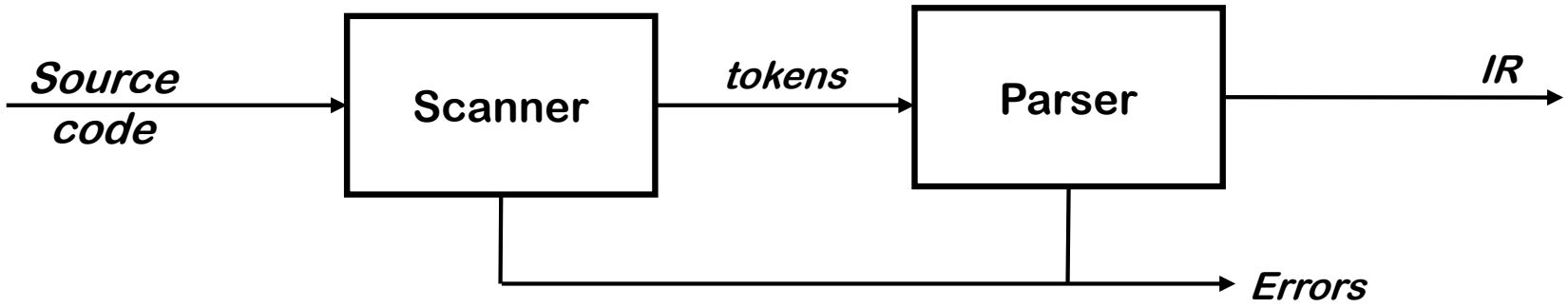


Can we build *n x m* compilers with *n+m* components?

- 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

Limited success in systems with very low-level IRs

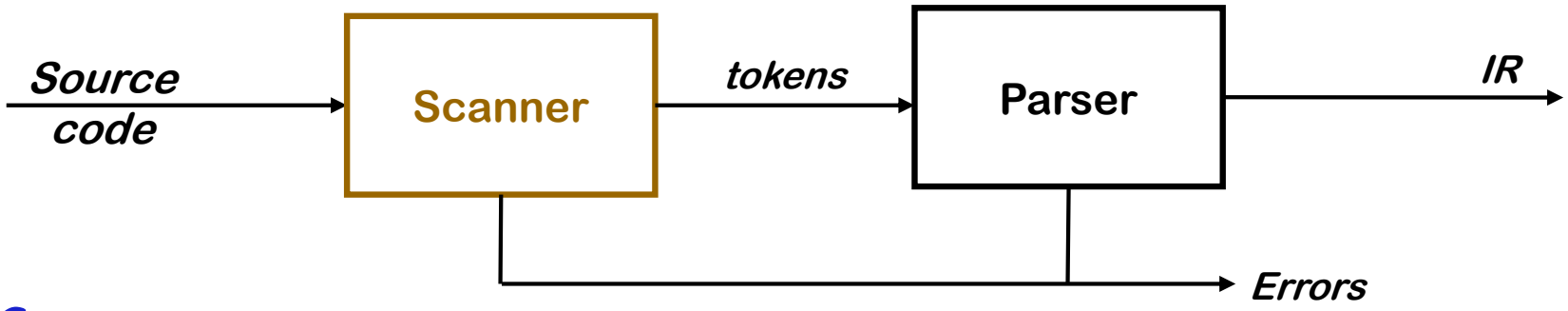
The Front End



Responsibilities

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

The Front End



Scanner

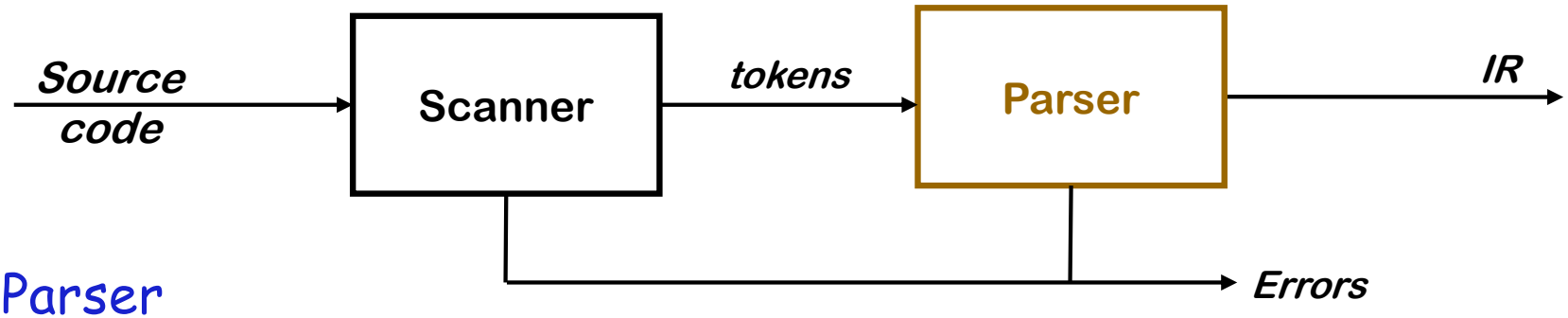
- Maps character stream into words—the basic unit of syntax
- Produces pairs — a word & its part of speech

  x = x + y ; *becomes* <id,x> = <id,x> + <id,y> ;

  → *word ≅ lexeme, part of speech ≅ token type*

  → In casual speech, we call the pair a *token*
- Typical tokens include *number*, *identifier*, +, -, new, *while*, *if*
- Scanner eliminates white space (*including comments*)
- Speed is important

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

The Front End

Context-free syntax is specified with a grammar

$$\textit{SheepNoise} \rightarrow \textit{SheepNoise}\ \underline{\text{baa}}$$
$$\mid\ \underline{\text{baa}}$$

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* $(P : N \rightarrow N \cup T)$

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 Front End

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

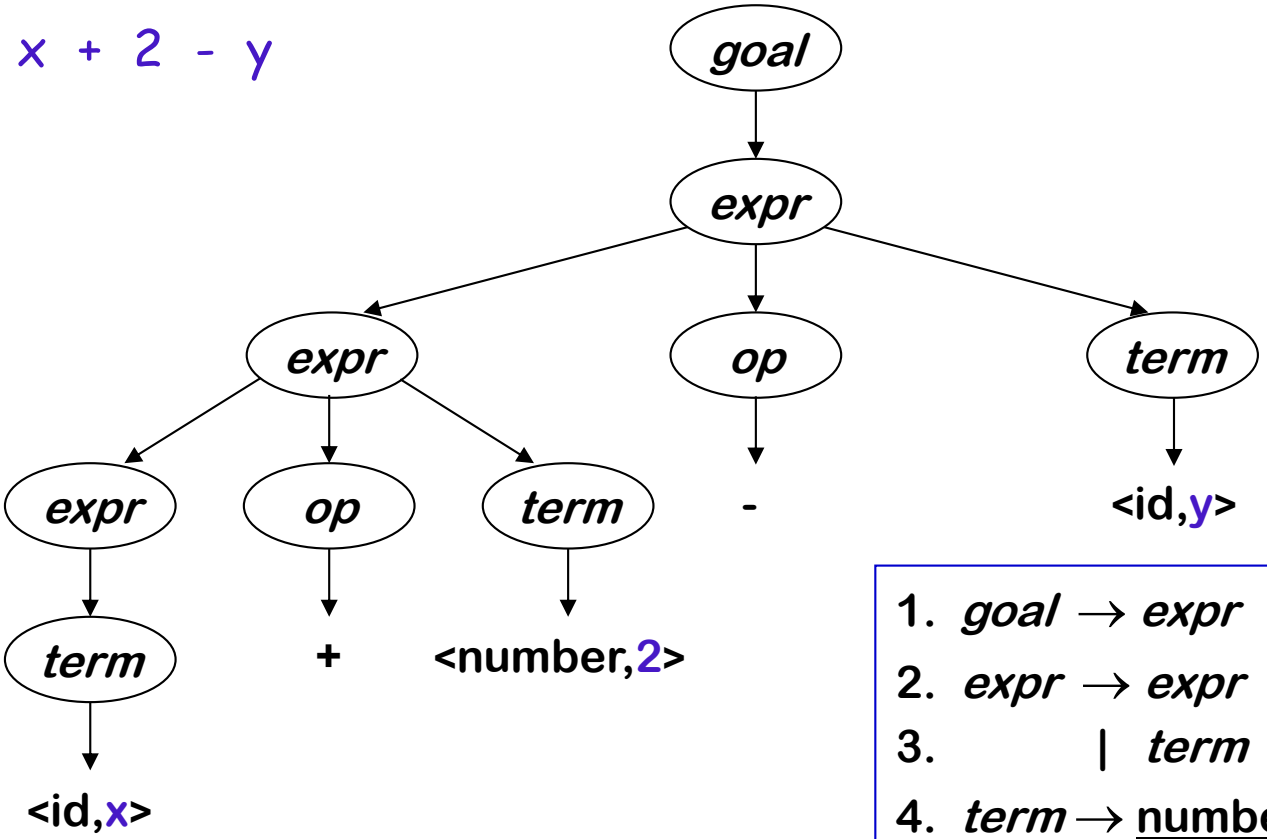
Production	Result	
	goal	=>
1	*expr*	=>
2	*expr op term*	=>
5	*expr op* y	=>
7	*expr* - y	=>
2	*expr op term* - y	=>
4	*expr op* 2 - y	=>
6	*expr* + 2 - y	=>
3	*term* + 2 - y	=>
5	x + 2 - y	

To recognize a valid sentence in some CFG, we will need to construct a derivation automatically (forwards or backwards)

The Front End

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

x + 2 - y

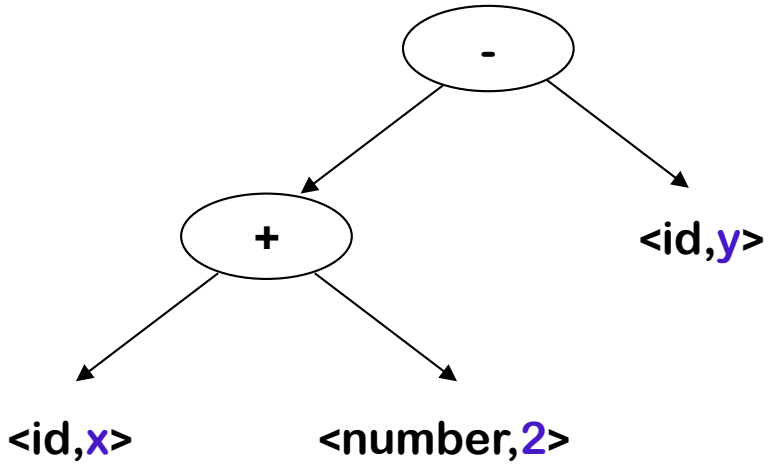


This contains a lot of unneeded information.

1. *goal* → *expr*
2. *expr* → *expr op term*
3. | *term*
4. *term* → number
5. | id
6. *op* → +
7. | -

The Front End

Compilers often use an *abstract syntax tree (AST)*

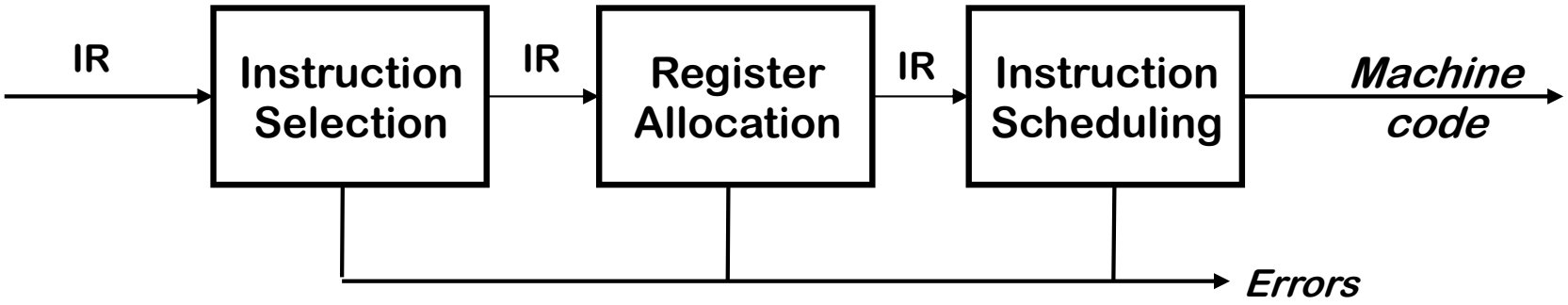


The AST summarizes grammatical structure, without including detail about the derivation

This is much more concise

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

The Back End

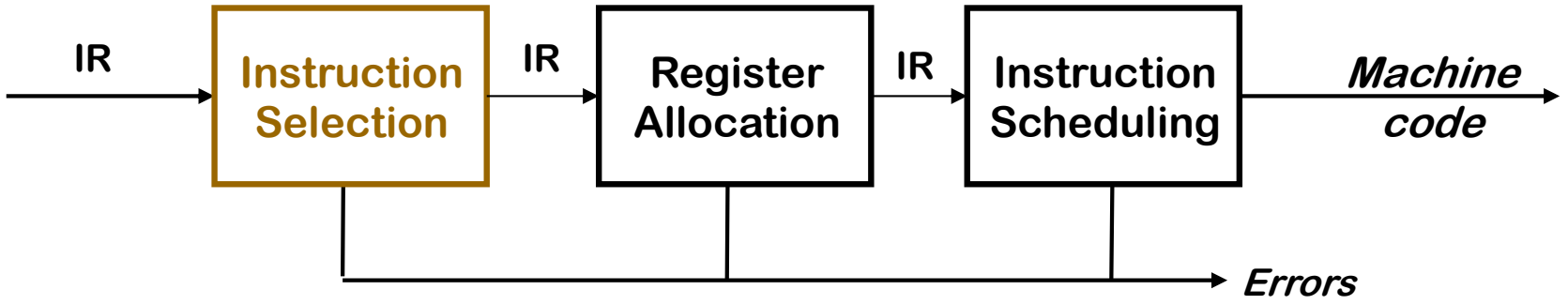


Responsibilities

- 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



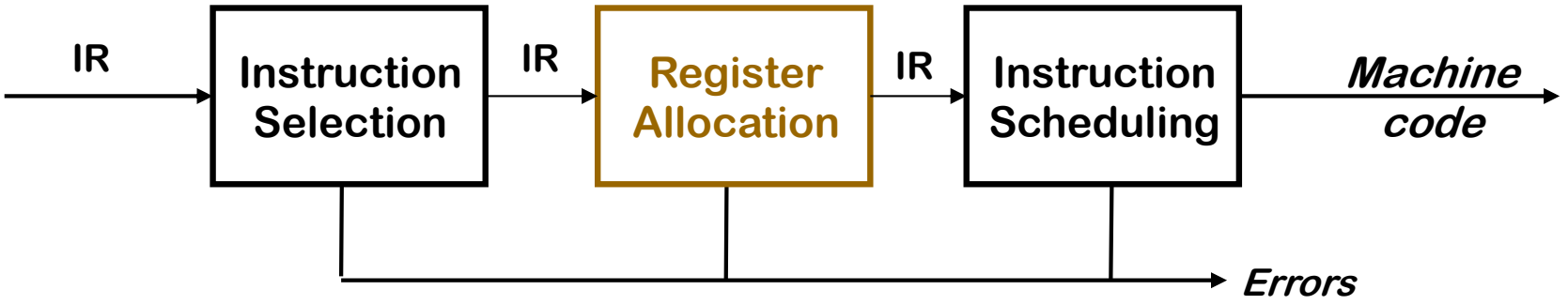
Instruction Selection

- 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

This was the problem of the future in 1978

- → Spurred by transition from PDP-11 to VAX-11
- → Orthogonality of RISC simplified this problem

The Back End

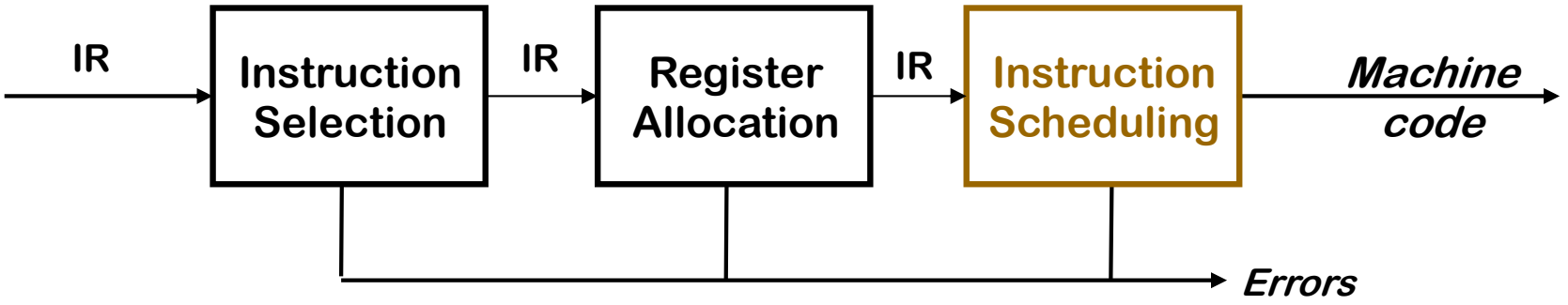


Register Allocation

- Have each value in a register when it is used
- Manage a limited set of resources
- Select appropriate LOADs & STOREs
- Optimal allocation is NP-Complete (1 or *k* registers)

Typically, compilers approximate solutions to NP-Complete problems

The Back End



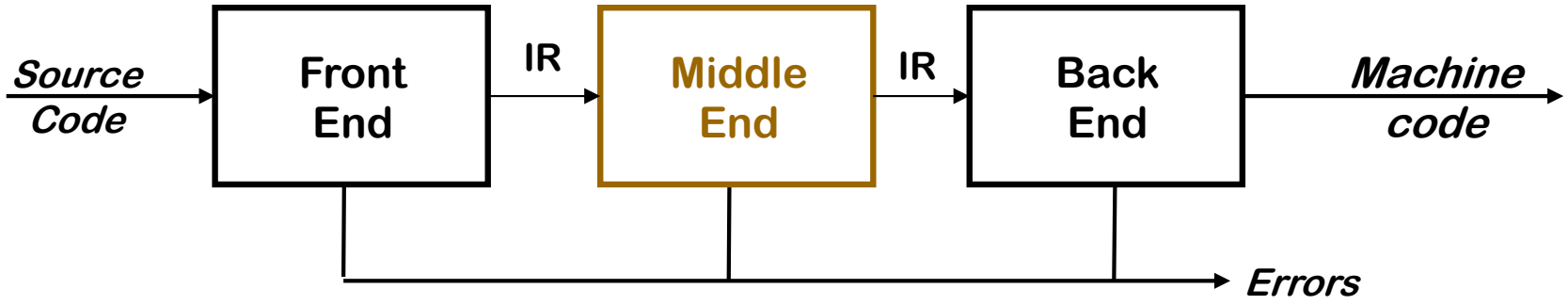
Instruction Scheduling

- 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-pass Compiler

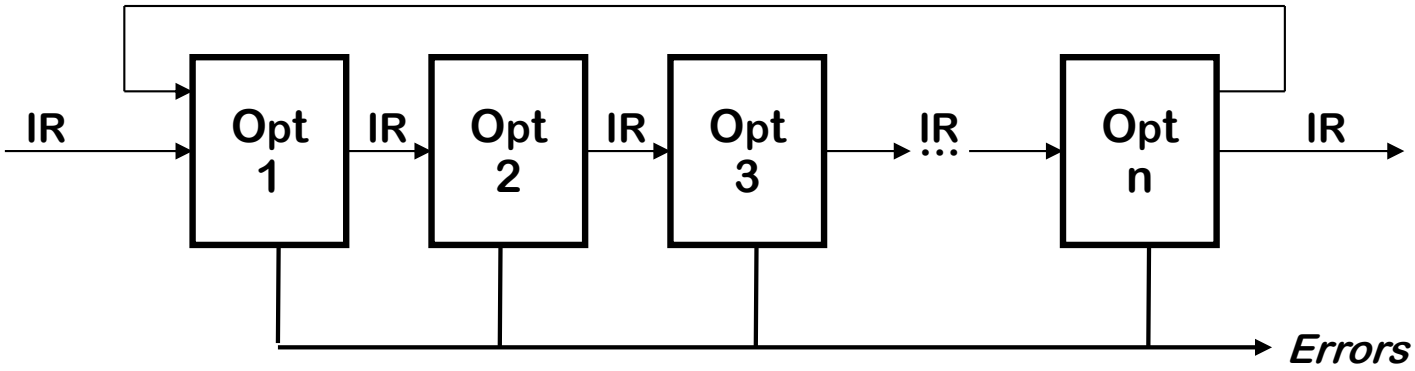


Code Improvement (or *Optimization*)

- Analyzes IR and rewrites (or *transforms*) IR
- Primary goal is to reduce running time of the compiled code
  - → May also improve space, power dissipation, energy consumption, …
- Must preserve "meaning" of the code (may include approximations, i.e., quality of outcomes tradeoffs)
  - → Measured by values of named variables or produced output

Subject of cs515, and cs516, maybe final weeks of cs415

The Optimizer (or Middle End)

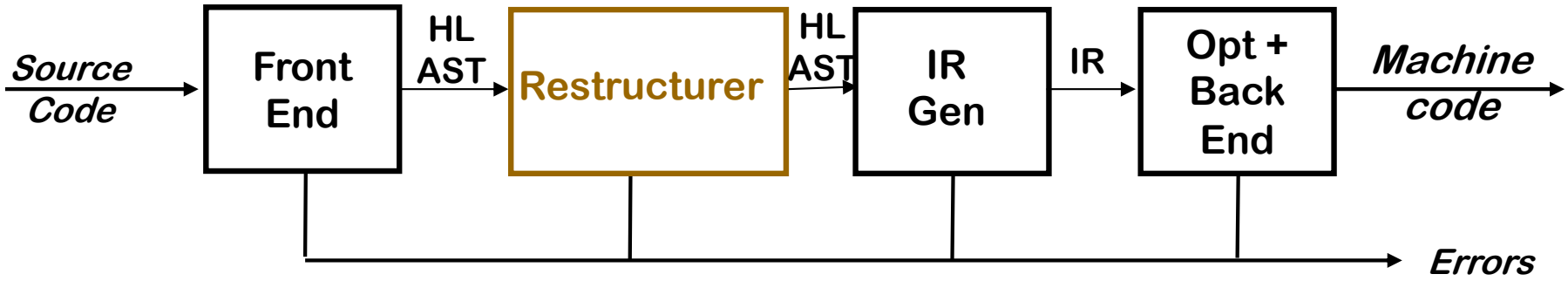


Modern optimizers are structured as a series of passes

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

Modern Restructuring Compiler



Typical Restructuring (source-to-source) Transformations:

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

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 (exception handling)
- Interface to the operating system
  - → Input and output
- Support of parallelism
  - → Parallel thread initiation
  - → Communication and synchronization

Next class

Instruction Scheduling

Read EaC: Chapters 12.1 – 12.3

No recitations or office hours this week