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

Critical Facts

Welcome to CS415 — *Compilers*

Topics in the design of programming language translators, including parsing, run-time storage management, error recovery, code generation, and optimization

- Instructor: Zheng Zhang (eddy.zhengzhang@cs.rutgers.edu)
- Teaching Assistants: Shaohua Duan, Guangyuan Hu (sd904, gh279@cs.rutgers.edu)
- My Office Hours: Thursday, 3:30-4:30pm, CoRE310
- Required Text : Engineering a Compiler, Morgan-Kaufmann
- Recommended Text: The Dragon Book
- Web Site: www.cs.rutgers.edu/courses/415/classes/spring_2016_zhang
  - → Lecture notes, projects, slides …
- News group: sakai system sakai.rutgers.edu
  - → Homework and project questions
- Get account on ilab

Announcements

No recitation this week (today!)

Special permission numbers

please come and see me after class

Basis for Grading

- Exams
  - → Midterm 20%
  - → Final 30%
- **Homework** **15%** (important)
- Recitation Attendance 5%
- Projects (tentative!)
  - → Back End 10%
  - → Front End 10%
  - → Code generator 10%
  - → Vectorizer or optimizer 10% (not required)

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

Basis for Grading

- Exams
  - → Midterm
  - → Final
  - ✦ Closed-notes, closed-book
  - ✦ final is cumulative
- Homework
  - ✦ Reinforce concepts, provide practice
  - ✦ Number of assignments *t.b.d.*
- Projects
  - → Back End
  - → Front End
  - → Code generator
  - ✦ High ratio of thought to programming
  - ✦ single student labs (note academic integrity information)

Rough Syllabus

• Overview	§ 1
• Local Register Allocation	§ 13
• Instruction Scheduling	§ 12
• Scanning	§ 2
• Parsing	§ 3
• Context Sensitive Analysis	§ 4
• Inner Workings of Compiled Code	§ 6, 7
• Introduction to Optimization	§ 8
• Data Flow Analysis	§ 9

- More Optimization (*time permitting*)
- Advanced topics in language design/compilation:
  automatic parallelization / vectorization / locality optimizer

Class-taking technique for CS 415

- I will use projected material extensively
  - → I Feel free to interrupt me if you have any questions
- 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 tests will cover both lecture and reading
  - → I will probably hint at good test questions in class
- 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 Sakai forum

About Textbooks

- "Engineering a Compiler" by Cooper and Torczon
  - → First or second edition
- Book presents modern material
  - → Addresses every aspect of modern compiler construction
- Other recommended textbook is the "Dragon Book"
  - → Aho, Lam, Sethi, Ullman: Compilers - Principles, Techniques, and Tools (2nd edition)
  - → Older version (Dragon book) also fine

Programming Languages

- What is a (formal) language?
  - → Words are finite sequences of symbols taken from a finite alphabet; A language is a set of words over a finite alphabet
  - → Languages can be finite or infinite
- How to specify a language?
  - → The more structure the language has, the more complex its specification
  - → One option: use a "rewrite rule" based system to generate words in the language
- How to recognize whether a word is in a language or not?
  - → The more structure the language has, the more complex its recognition
  - → One option: use an abstract, "mathematical" machine to "parse" a word

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 (python) is compiled to bytecode
  - → which is then interpreted
  - → Or a hybrid strategy is used
    - Just-in-time compilation (to native host code)
    - 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

Intrinsic interest

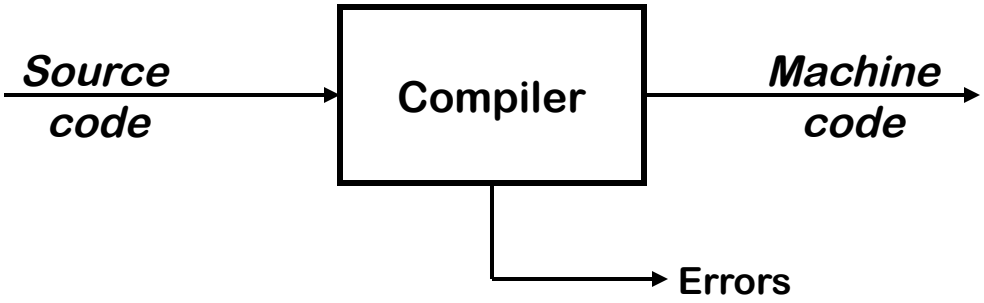
- Compiler construction involves ideas from many different parts of computer science

Artificial intelligence	Greedy algorithms Heuristic search techniques
Algorithms	Graph algorithms, scheduling, Dynamic programming
Theory	DFAs & PDAs, pattern matching Fixed-point algorithms
Systems	Allocation & naming, synchronization, parallelism, memory layout
Architecture	Pipeline & hierarchy management Instruction set use

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)
  - Compilers must hide that complexity from the programmer
  - Success requires mastery of complex interactions

High-level View of a Compiler

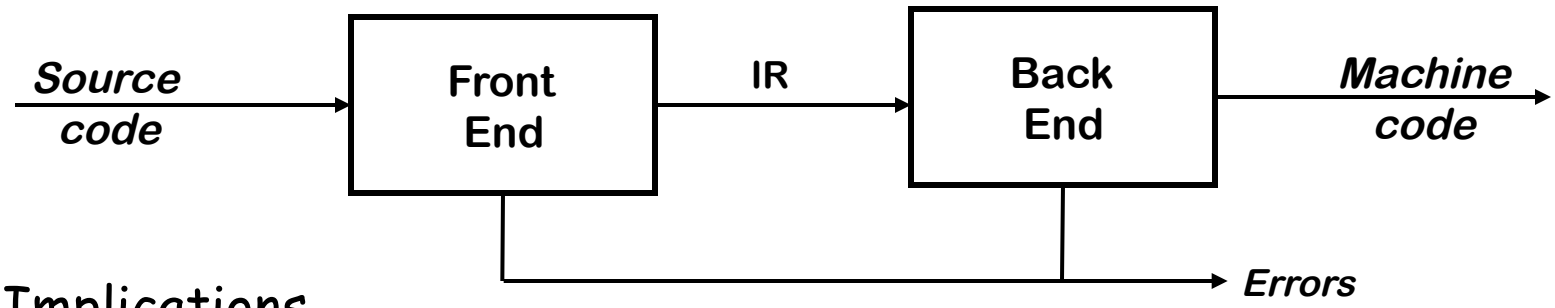


Implications

- 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

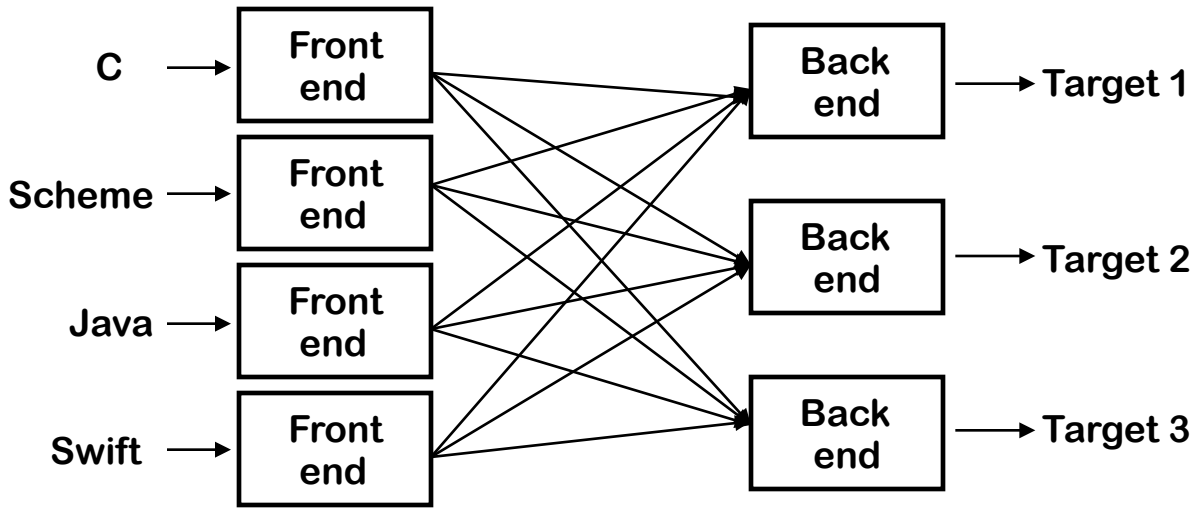


Implications

- 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 NPC

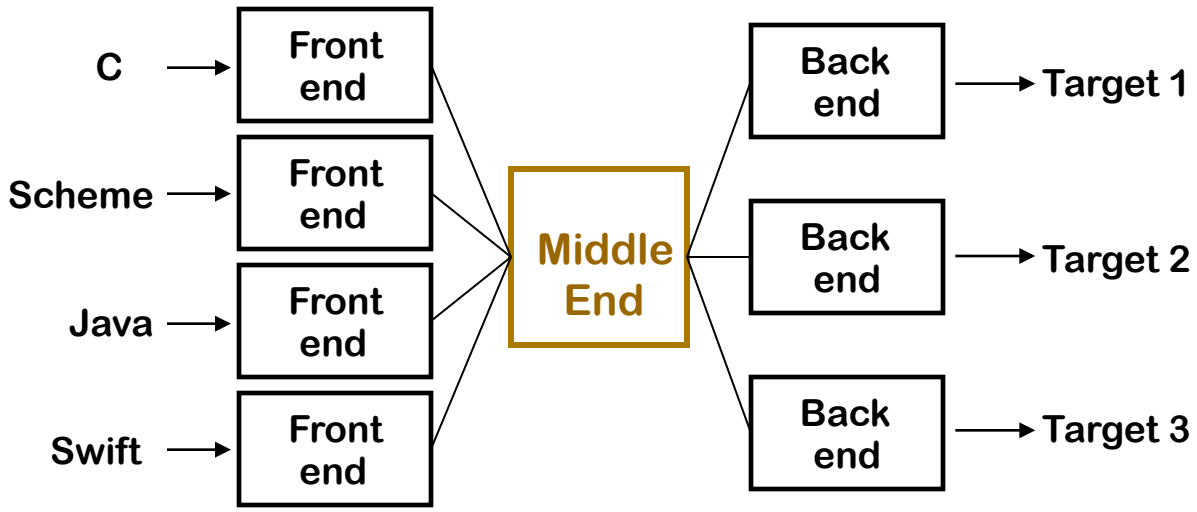
A Common Fallacy



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

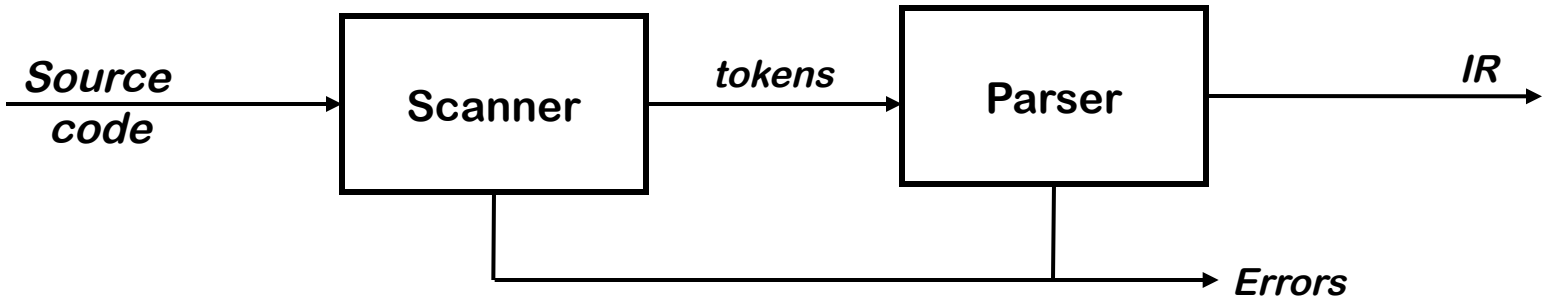
A Common Fallacy



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

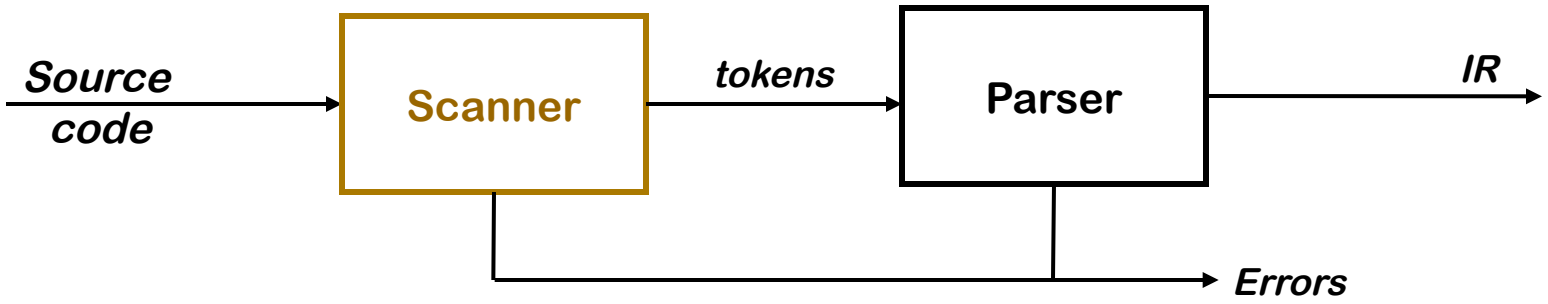
The Front End



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

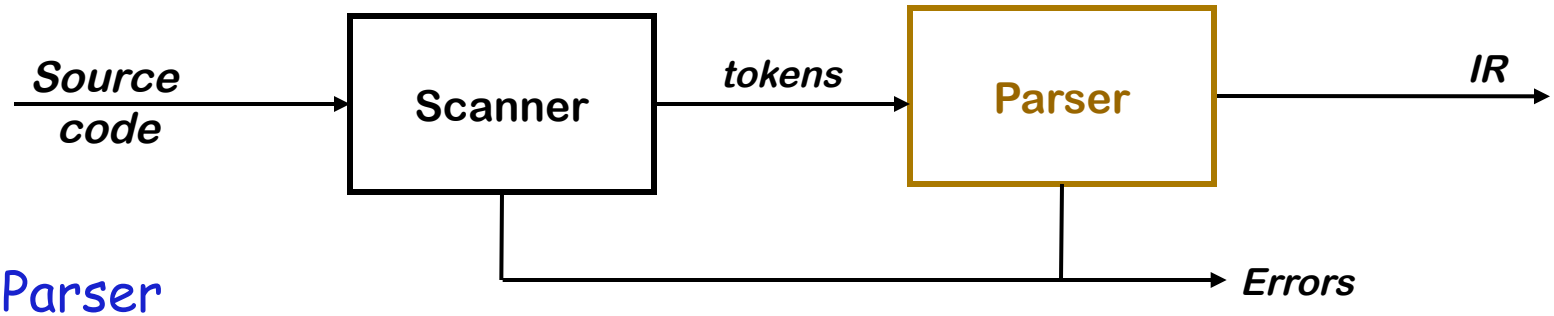
The Front End



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

Today we mostly use automatic parser generators

The Front End

Context-free syntax is specified with a grammar

$$CatNoise \rightarrow CatNoise\ \underline{mew}$$
$$\mid\ \underline{mew}$$

This grammar defines the set of noises that a cat 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" (CFG)

The Front End

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

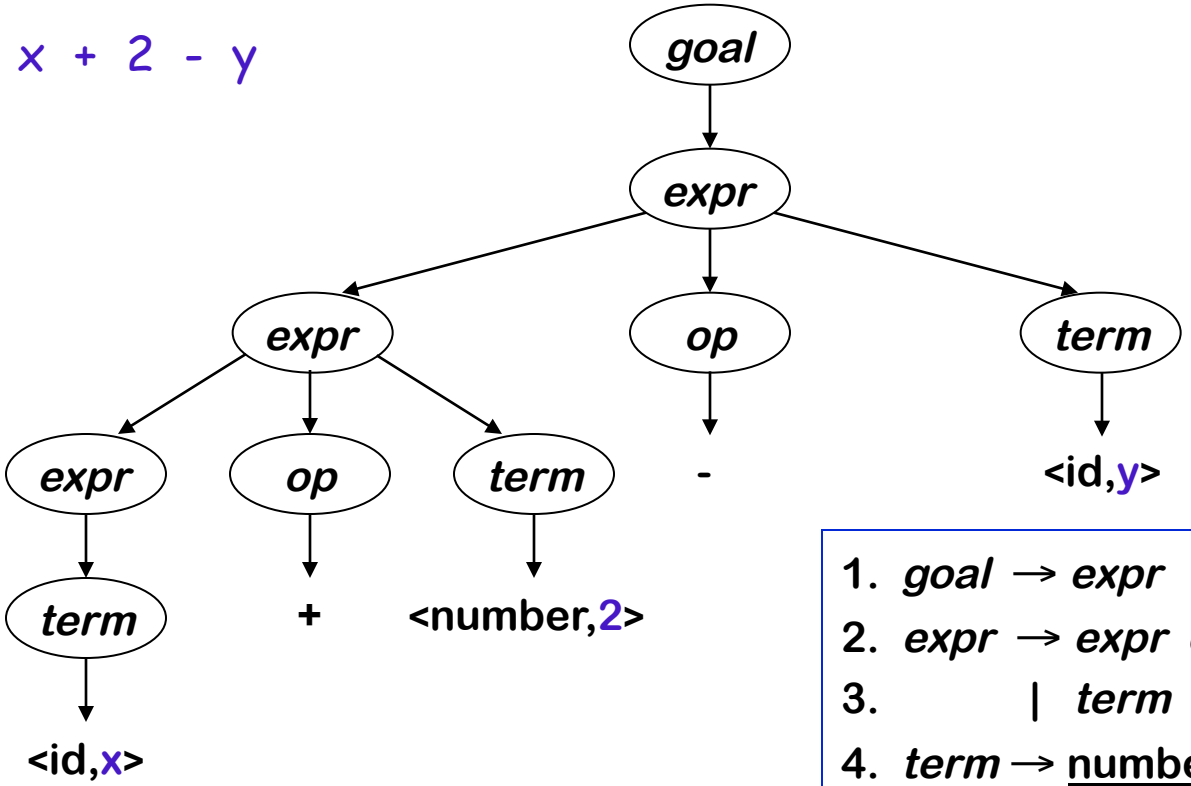
Production	Result
	goal
1	*expr*
2	*expr op term*
5	*expr op* y
7	*expr* - y
2	*expr op term* - y
4	*expr op* 2 - y
6	*expr* + 2 - y
3	*term* + 2 - y
5	x + 2 - y

To recognize a valid sentence in some CFG, we 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

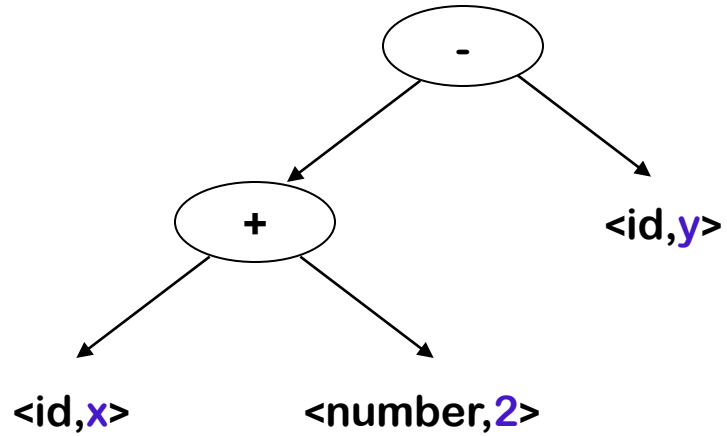


This contains a lot of unneeded information.

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*

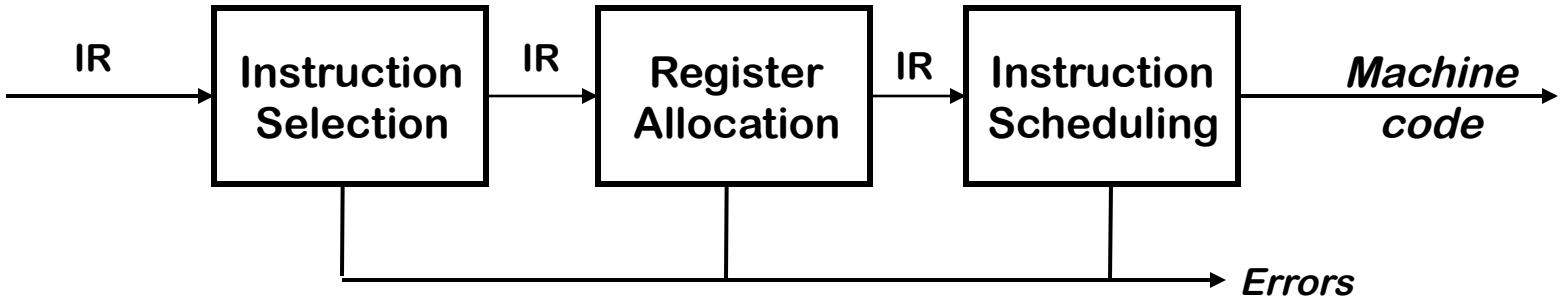


The AST summarizes grammatical structure, without including detail about the derivation

This is much more concise

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

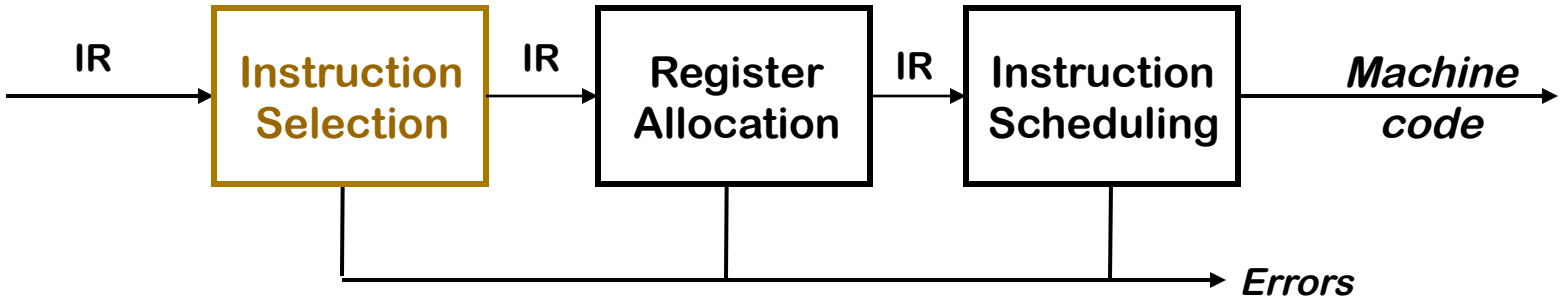
The Back End



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

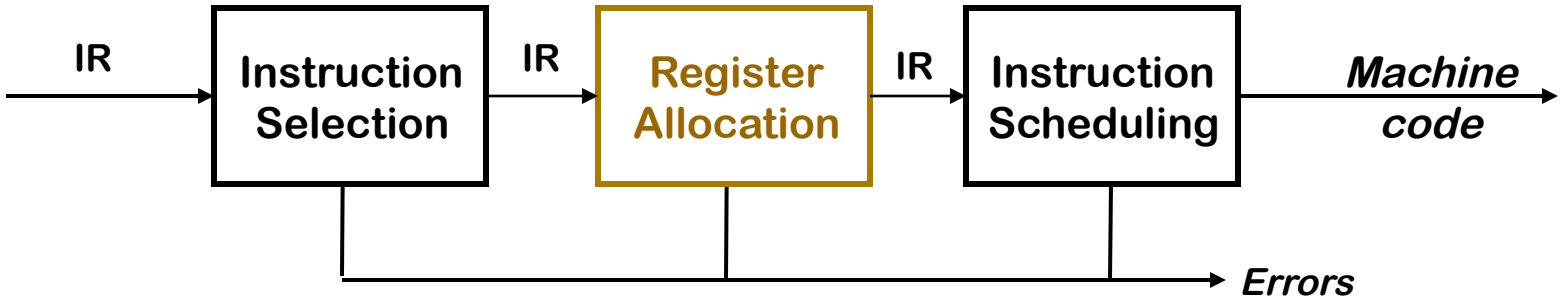
The Back End



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

The Back End

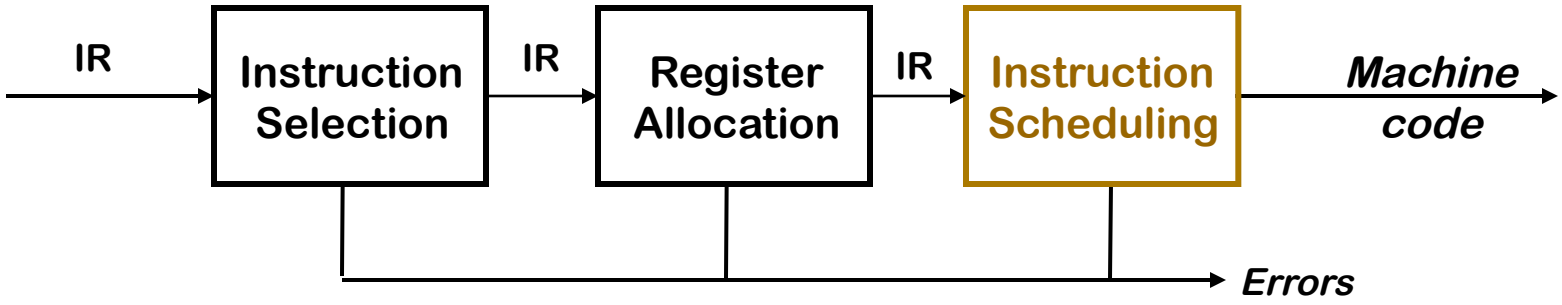


Register Allocation

- 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



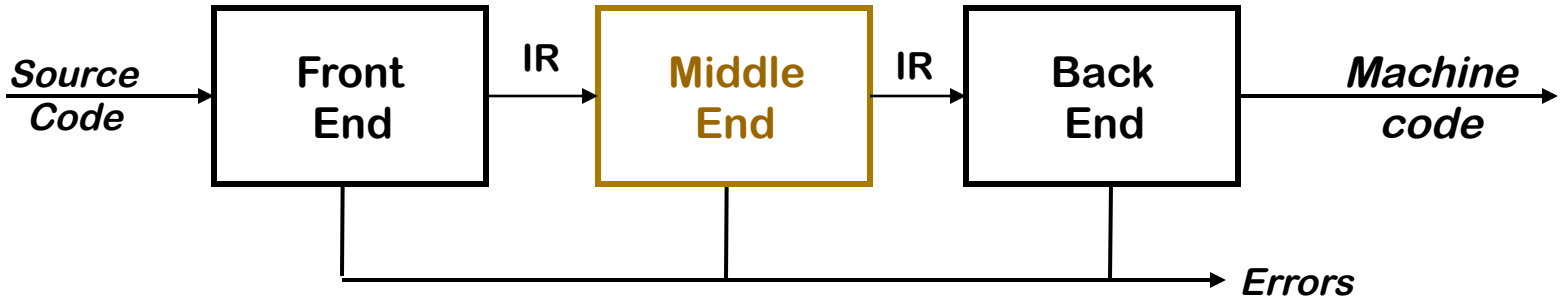
Instruction Scheduling

- 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, energy consumption, …
- Must preserve "meaning" of the code
  - → Dependence preserved, values of named variables unchanged

Next class

Registers Allocation

Read EaC: Chapters 13.1 – 13.3

No recitation this week (today!)