CS143: Intro

David L. Dill

Stanford University

Introduction

- Still getting organized – more info on web page tonight and tomorrow.
- Web page: cs143.stanford.edu
- Exams
  - Midterm – in class,
  - Final 6/9 3:30 PM (If it disagrees with University schedule, University wins).
- Piazza – first place to go for questions and answers.

Graded work

- Project 50%
  - Programming problems: Implement a compiler for “COOL” language
  - Parts 1 & 2 – 10% each
  - Parts 3 & 4 – 15% each
- Written assignments 10%
  - More theoretical questions
  - Submitted on Scoryst (more details later)
- Midterm 15% (in class).
- Final 25%

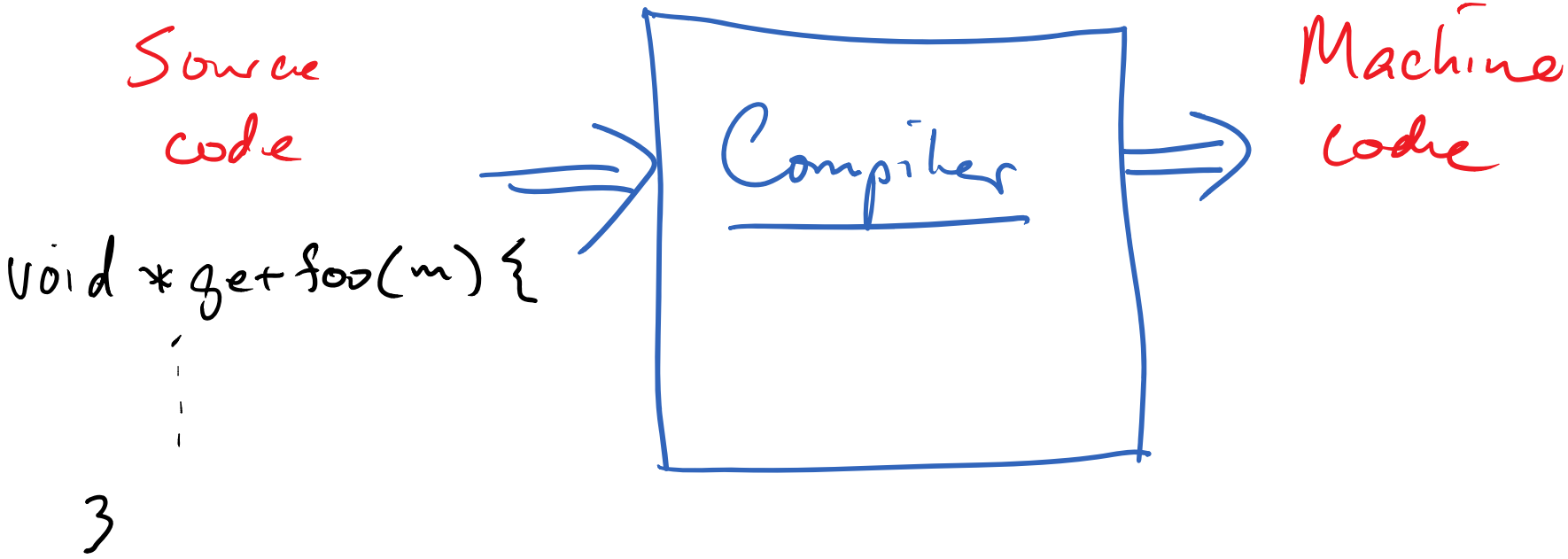
Outline

- What are compilers and why are they worth studying?
- Compiler organization
- Front end
  - Lexical analysis
  - Syntactic analysis (parsing)
  - Semantic analysis
- Back end
  - Optimization
  - Code generation
- Interpreters
- Engineering

What are compilers and why are they worth studying?

What is a compiler?

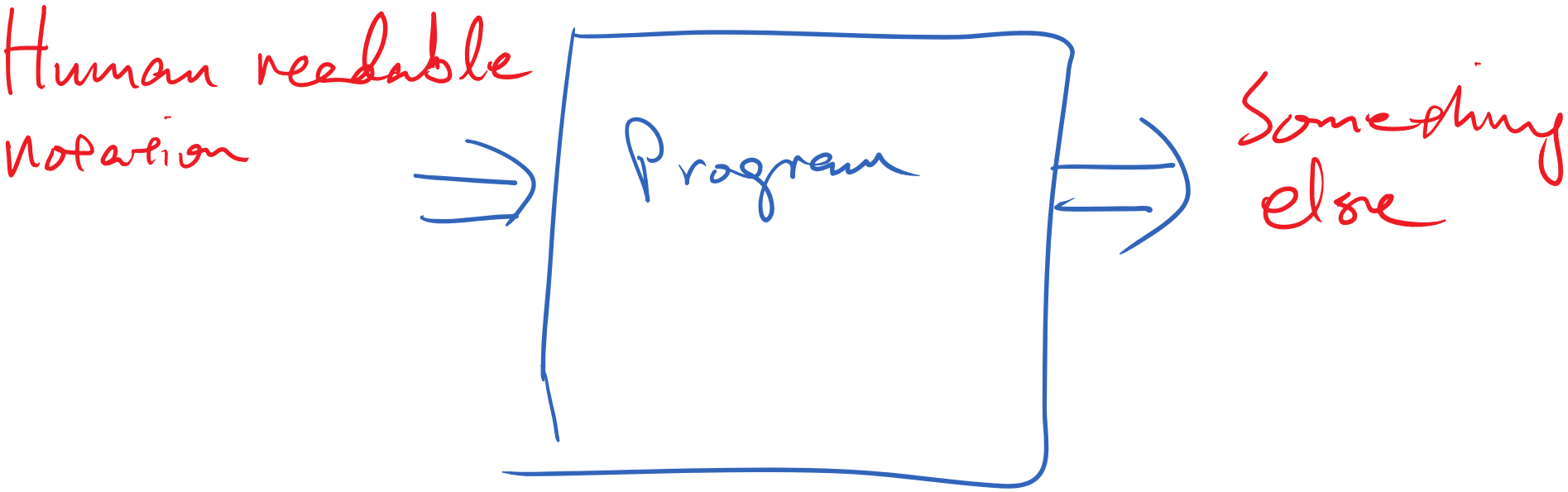
Programming language translator.



Other programs

Human readable notation

Other programs



Often uses similar architecture / technology

Examples

- Document formatters
- Description languages
  - Graphics
  - Hardware
  - Data (many kinds)
- Query languages
- Compiler compilers

Knowing about compilers can help you with lots of other problems

Why study compilers?

Practical

Standard architecture, ideas, tools for many applications

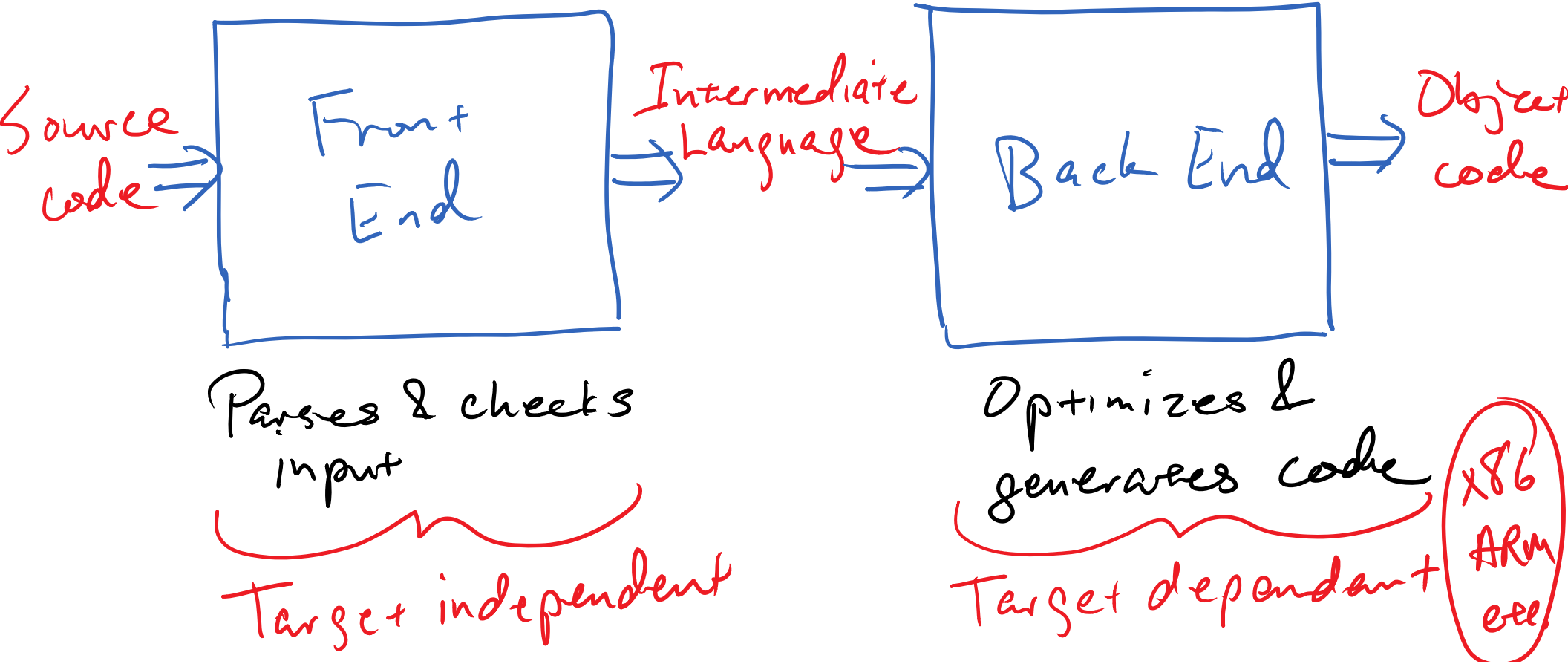
Implementing a programming language is fun.

Theory: Applied formal language theory, algorithms, logic.

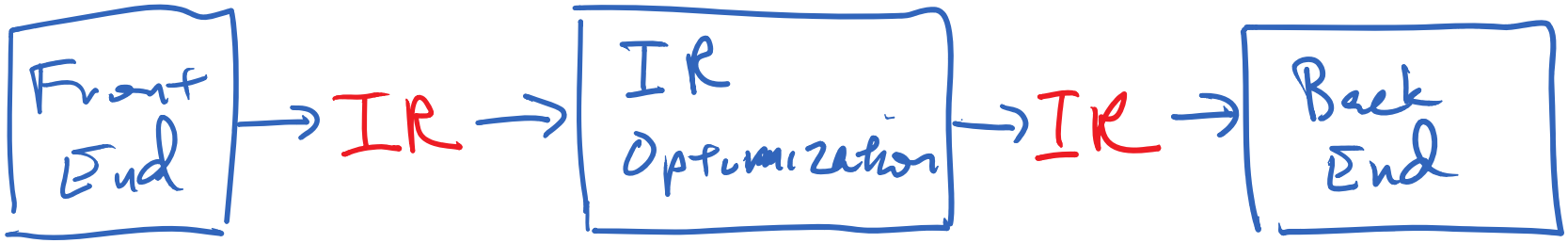
Compiler organization

Structure of a compiler

Approximate



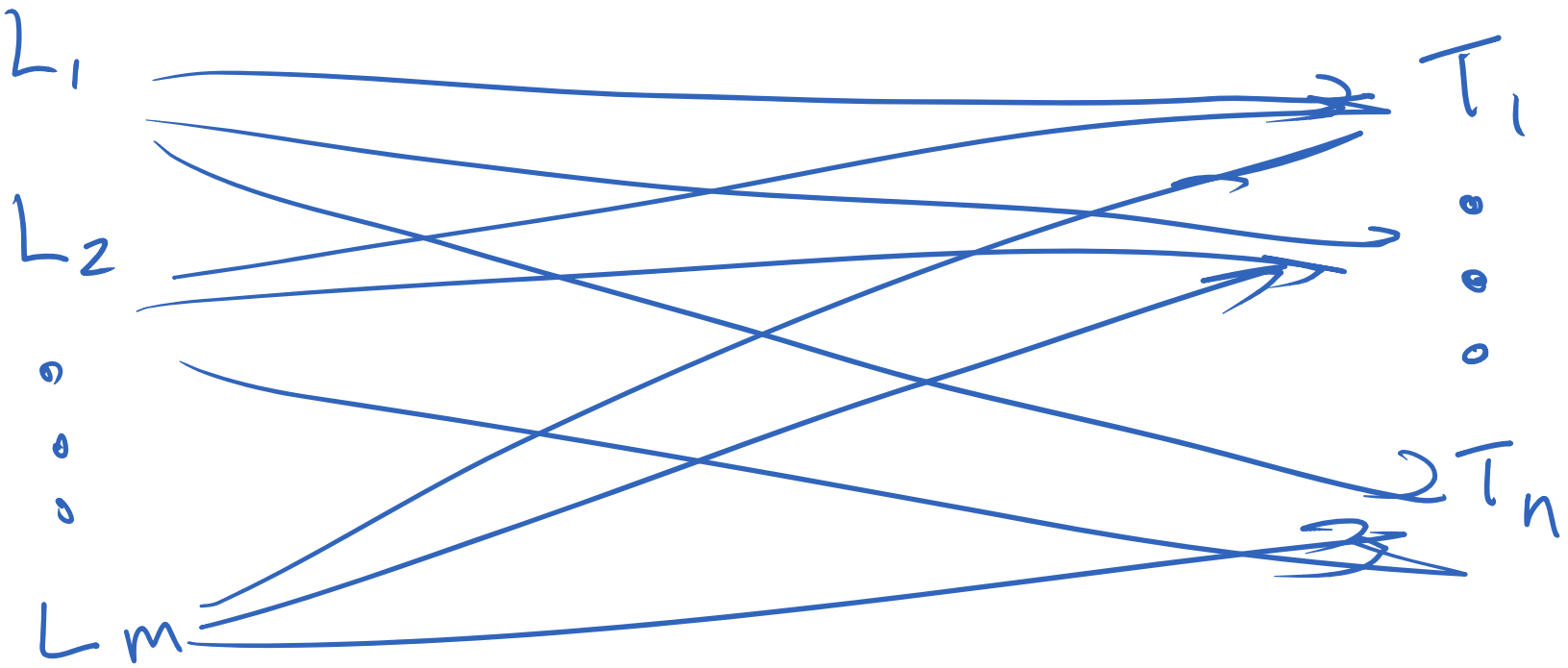
Better diagram



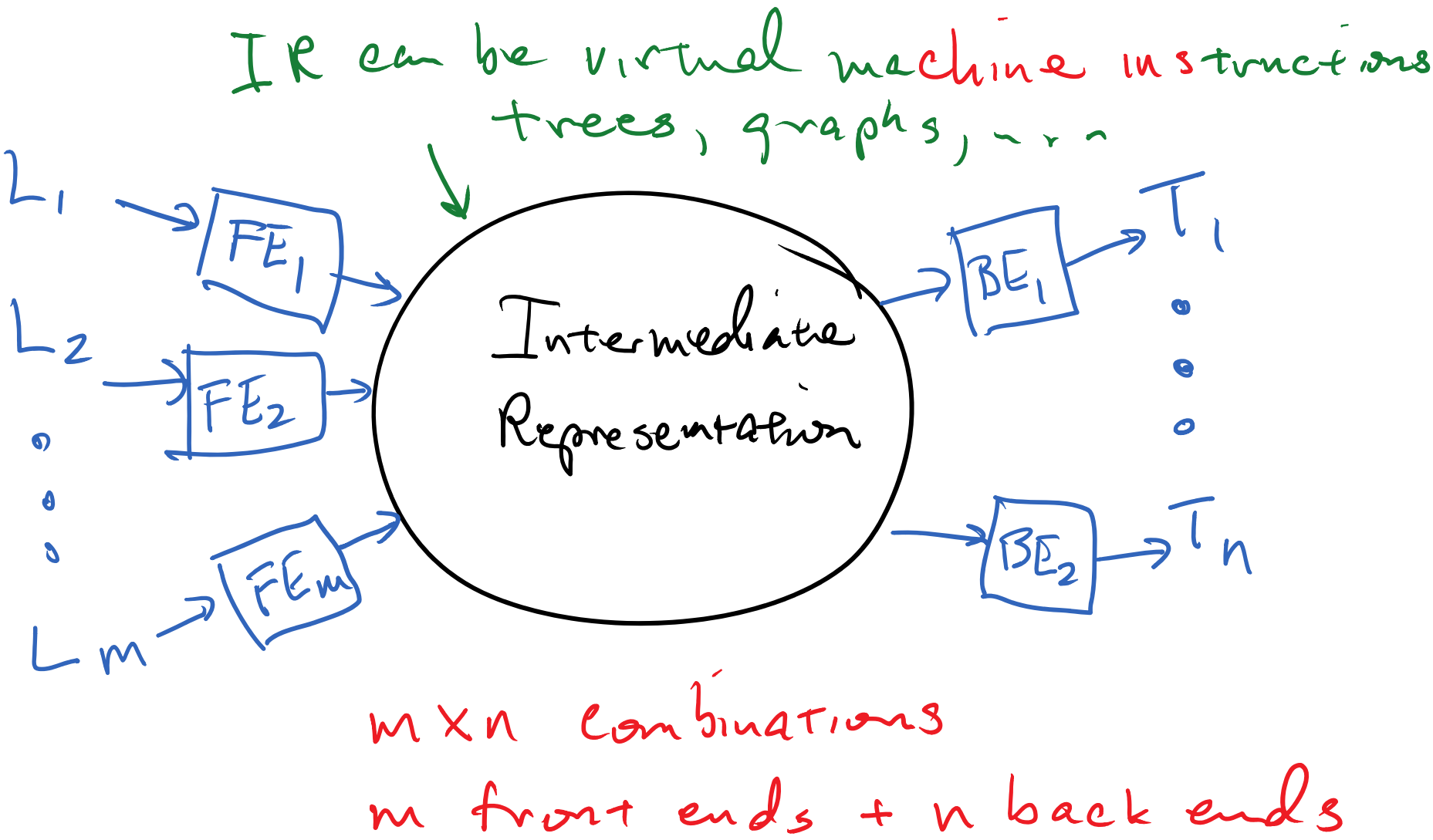
target independent optimizations

target dependent optimizations

Oversimplified!

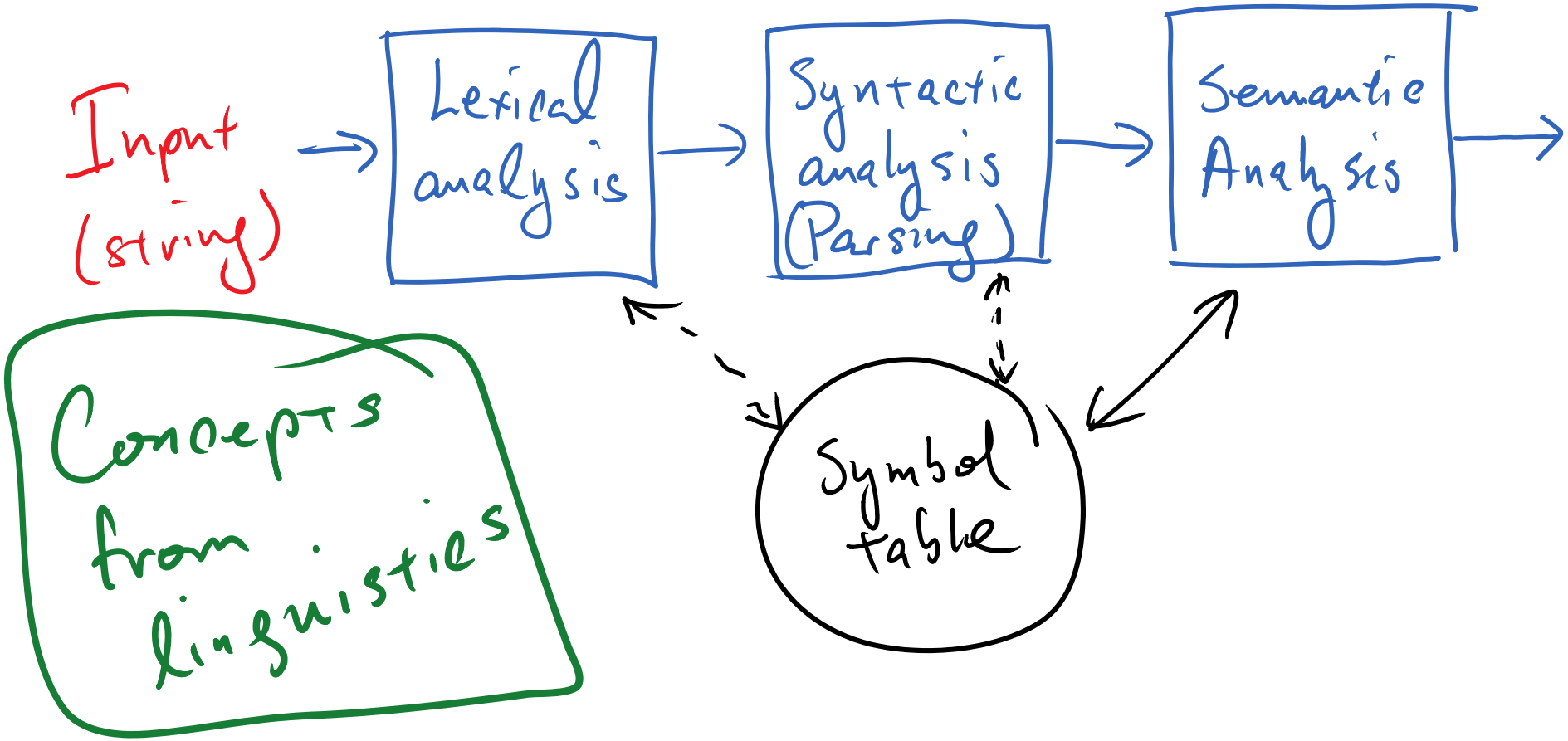


m languages X n targets


IR can be virtual machine instructions
trees, graphs, ...
L_1
L_2
L_m
FE_1
FE_2
FE_m
Intermediate Representation
BE_1
BE_2
T_1
T_n
m x n combinations
m front ends + n back ends

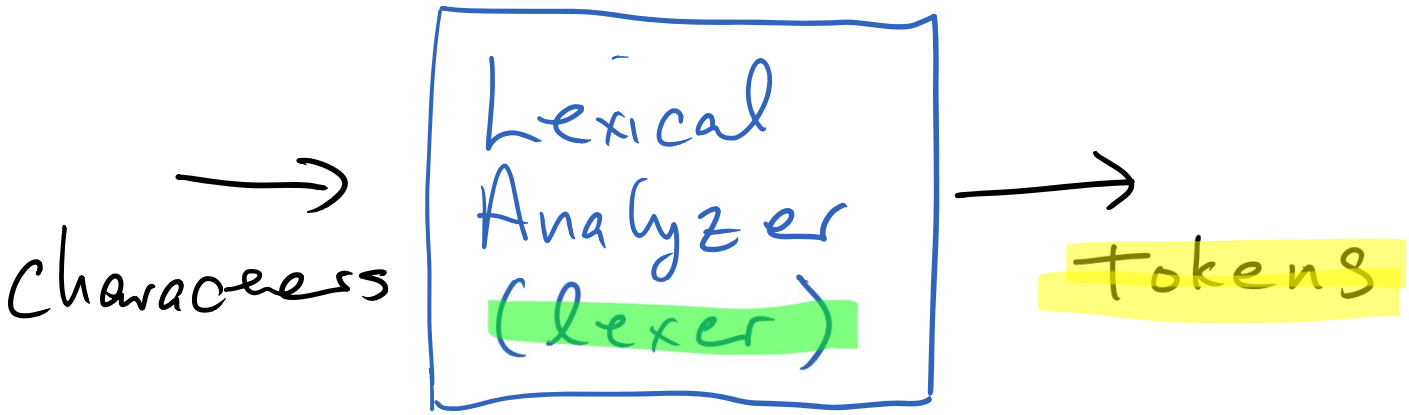
Front end

Front End


Input
(string)
Lexical analysis
Syntactic analysis (Parsing)
Semantic Analysis
Concepts from linguistics
Symbol table

Lexical analysis

l e t t e r s → words



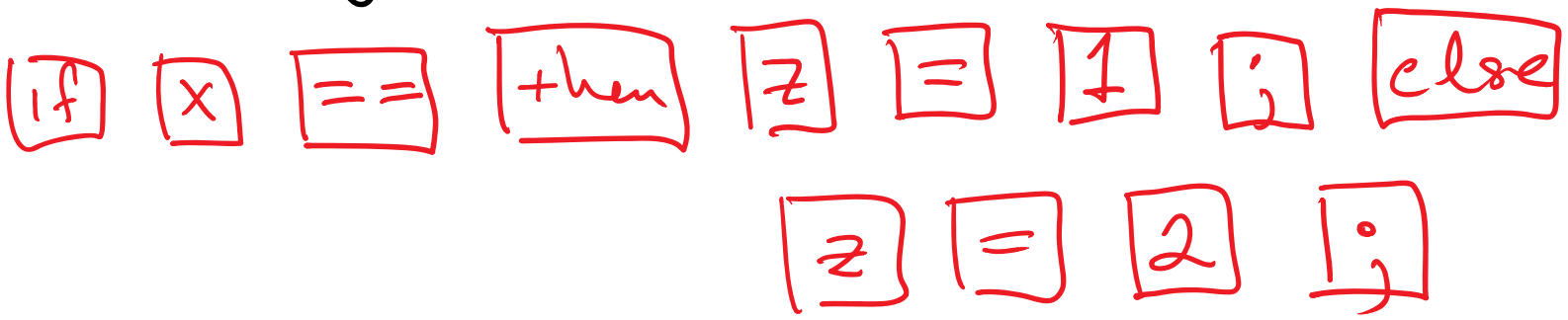
Lexical analysis

```
if x == y then z = 1, else z = 2;
```

tokens?

Lexical analysis

if x == y then z = 1 , else z = 2;



Theory in compilers

Precise definitions

Automated generation

Regular languages → automatic lexer generation

Context-free languages → automatic parser generation

Lexical Analysis Theory

Precise descriptions – Regular expressions

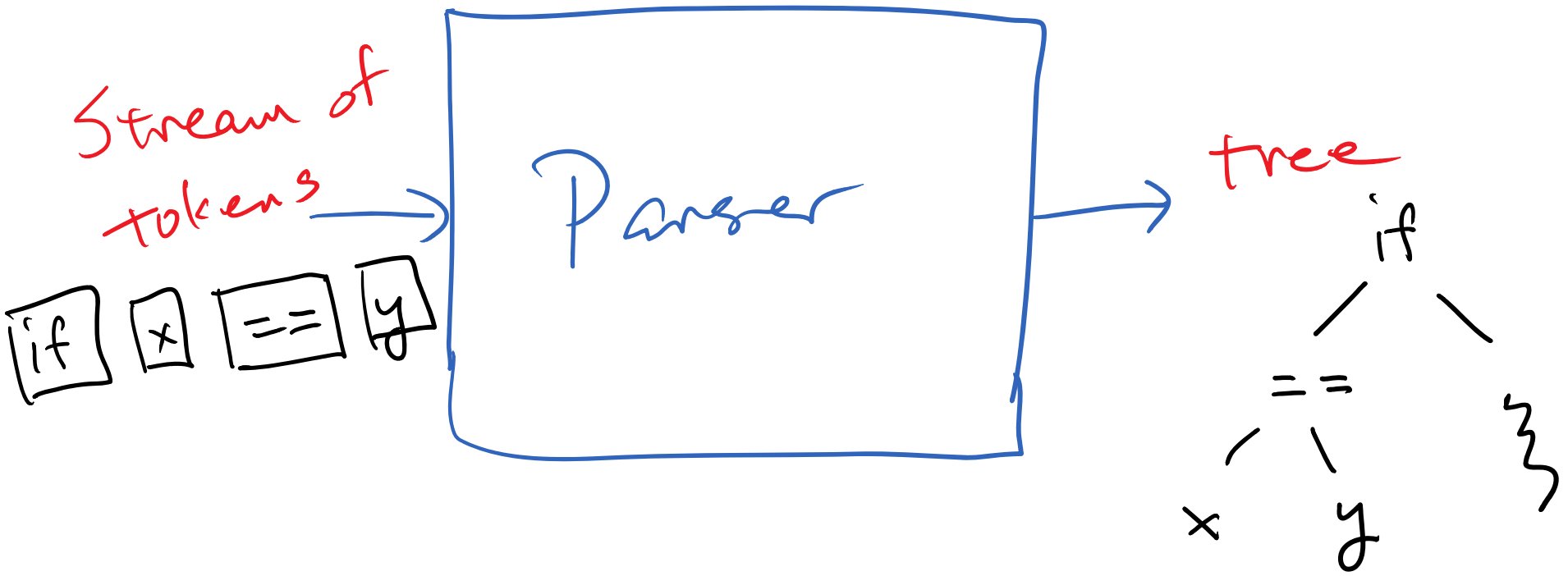
[a-z A-Z] [a-z A-Z 0-9 _]*

Automatic generation

Regular expressions → NFA → DFA

→ lexical analyzer program

Parsing

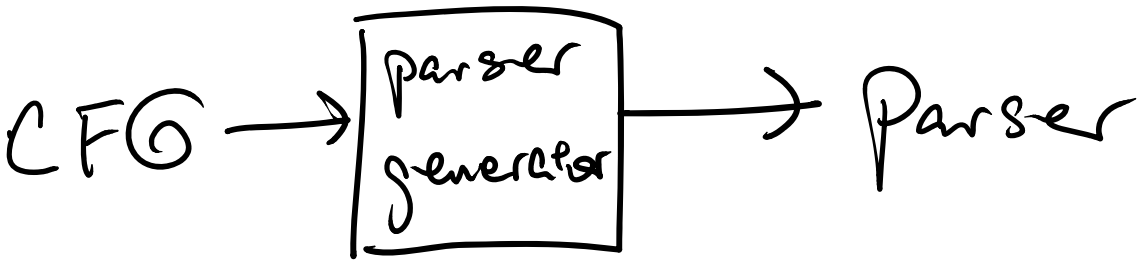


Parsing Theory

Precised descriptions - context-free grammars

("Backus Naur Form")

Automatic generation - parser generators



Semantic Analysis

"Static semantics" - checked at "compile time" - catch errors before program is run.

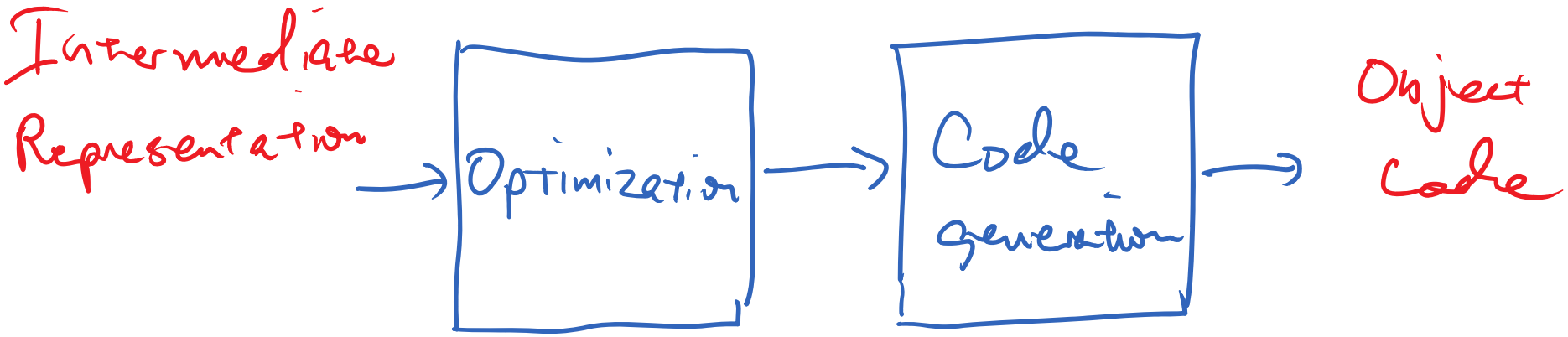
Type checking

Other (1 = 2 ?)

Theory is programming language specific.

Back end

Back end


Intermediate Representation
Optimization
Code generation
Object code

Optimization

Transform to "equivalent", better performing code.

"better performing"
- faster (may be complicated!)
- resource usage (code size, registers, memory)

"equivalent" - often compromised a bit.

Optimization

Sophisticated analysis often required.

Analyze code to understand properties

Results say which transformations are valid/useful.

We will not cover optimization in depth (See CS243)

Code Generation

Usually generates intermediate code

Map intermediate representation to instructions

$x + y \Rightarrow$ put x value in right place (e.g. register)
put y in right place
add instruction
put result in right place.

Code Generation

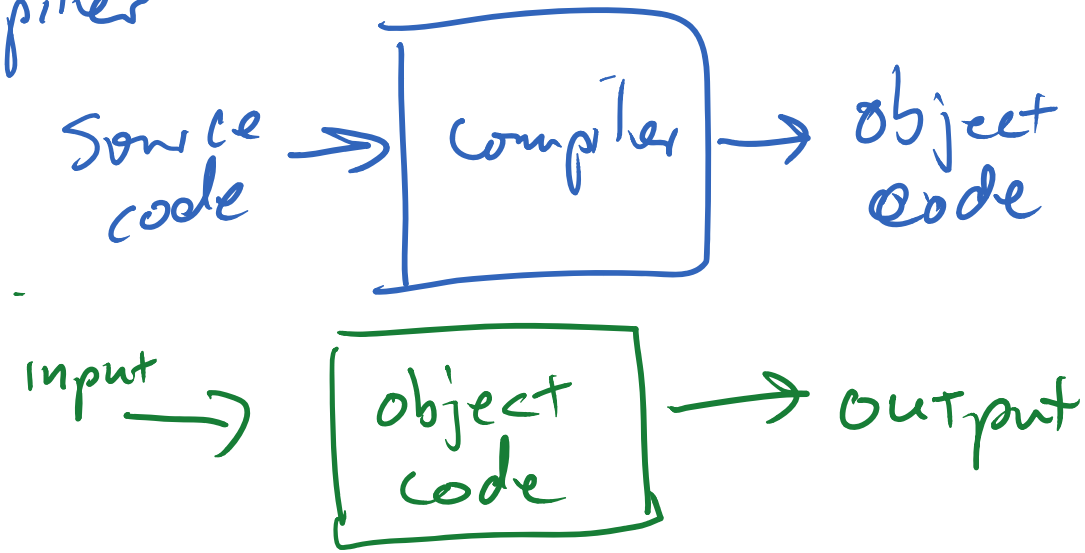
Issues

- Managing scarce resources (e.g. registers)
- Minimizing redundant moves
- Complex machine instructions may be able to do several intermediate operations

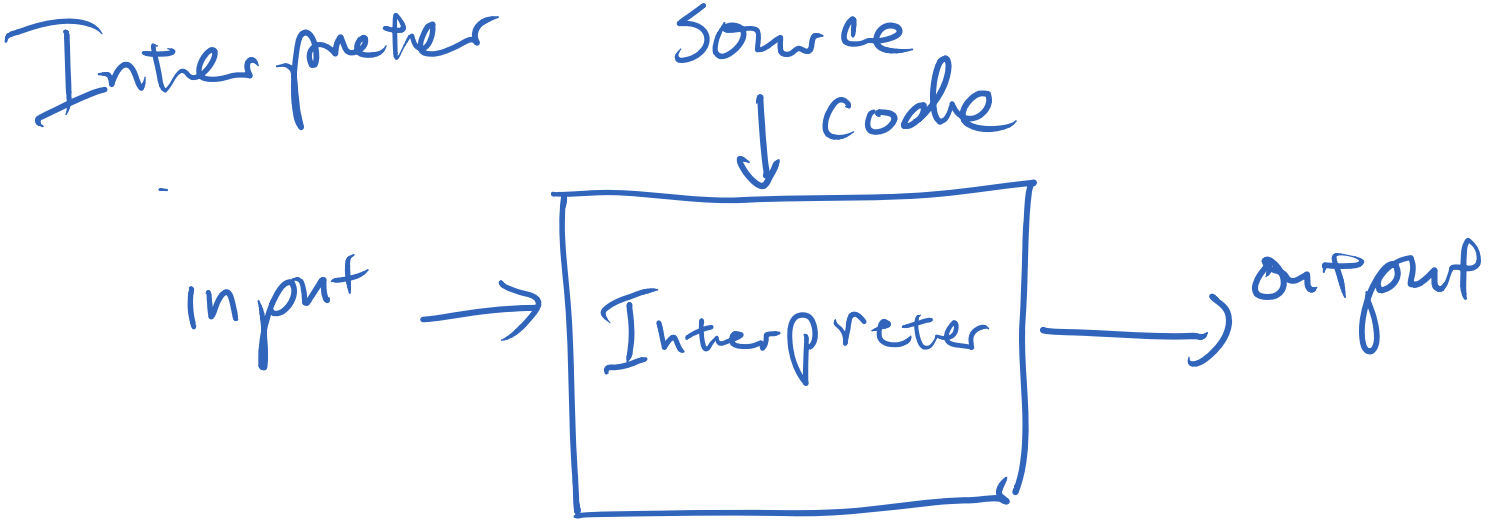
Interpreters

Interpreters vs Compilers

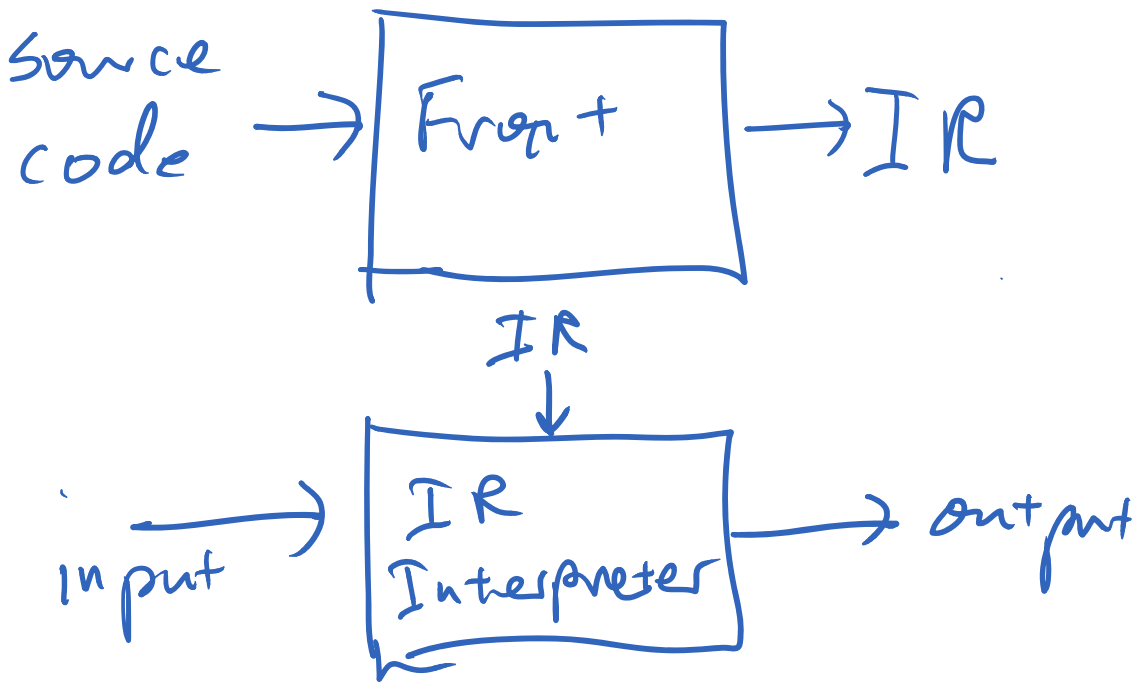
Compiler



Interpreters vs Compilers



Early phases — esp. lexical & syntax — are similar



E.g.

Java

IR = JVM code

Interpreter = JVM

Engineering

Engineering issues

Modern compilers are frustratingly complex

- Programming language inconsistency
- Target language complexity
- Complex optimization
- Code evolution