CS3300 - Compiler Design

Introduction

KC Sivaramakrishnan

IIT Madras

Administrivia

- Lecture Timings
  - Slot B: Monday 9 AM, Tuesday 8 AM, Wednesday 1 PM, Friday 11 AM
- Lab Timings
  - Slot Q: Tuesday 2 PM to 4:45 PM
- Course Webpage: `https://kcsrk.info/cs3300_m22/`
  - Lecture slides will be uploaded here.
- Course Moodle page: TBD
- Slack for communication: TBD
- Instructor e-mail address: kcsrk@cse.iitm.ac.in
  - Instructor Office Hours: None.
  - Feel free to ping me on Slack if you want to meet. TA Office hours will be announced soon.

Grading Policy

- Theory: 60%, Lab: 40%
- Theory
  - Quiz 1: 15%, Quiz 2: 15%, Endsem: 30%
  - **Extra:** Class Participation: upto 5%.
    - Class Participation will be monitored throughout the semester. You can participate by asking/answering questions during the lectures and/or in the Slack.
- Lab: 6 Assignments. More details will be announced by the end of the week.

Course outline

- Overview of Compilers
- Lexical Analysis and Parsing
- Type checking
- Intermediate Code Generation
- Register Allocation
- Code Generation
- Overview of advanced topics.

Goal of the course: At the end of the course,

- Students will have a fair understanding of some standard passes in a general purpose compiler.
- Students will have hands on experience on implementing a compiler for a subset of Java.

Course Textbooks

- Compilers: Principles, Techniques, and Tools, Alfred Aho, Monica Lam, Ravi Sethi, Jeffrey D. Ullman. Addison-Wesley, 2007 [**The Dragon Book**].
  - **2020 ACM Turing Award!** – . . . Who Shaped the Foundations of Programming Language Compilers and Algorithms
  - *"Columbia's Aho and Stanford's Ullman Developed Tools and Fundamental Textbooks Used by Millions of Software Programmers around the World"*
- Modern compiler implementation in Java, Second Edition, Andrew W. Appel, Jens Palsberg. Cambridge University Press, 2002.

Your friends: Languages and Tools

Start exploring

- C and Java - familiarity a must - Use of a IDE like Eclipse is recommended.
- Flex, Bison, JavaCC, JTB – tools you will learn to use.
- Make / Ant / Scripts – recommended toolkit.

Mutual expectations

For the class to be a mutually learning experience:

- What will be required from the students?
  - An open mind to learn.
  - Curiosity to know the basics.
  - Explore their own thought process.
  - Help each other to learn and appreciate the concepts.
  - Honesty and hard work.
  - Leave the fear of marks/grades.
- What are the students expectations?

Acknowledgement

These slides are heavily adapted from the slides prepared by V Krishna Nandivada and Kartik Nagar @ IIT Madras. Liberal portions of text are also taken verbatim from Antony L. Hosking @ Purdue, Jens Palsberg @ UCLA, Alex Aiken @ MIT and the Dragon book.

What, When and Why of Compilers

Compilers: What?

- What is a compiler?
  - a program that translates an executable program in one language into an executable program in another language.
  - Usually from a high-level language to machine language
- What is an interpreter?
  - a program that reads an executable program and produces the results of running that program
  - usually, this involves executing the source program in some fashion
- This course deals mainly with compilers. Many of the same issues also arise in interpreters.
- A common statement – XYZ is an interpreted (or compiled) language.

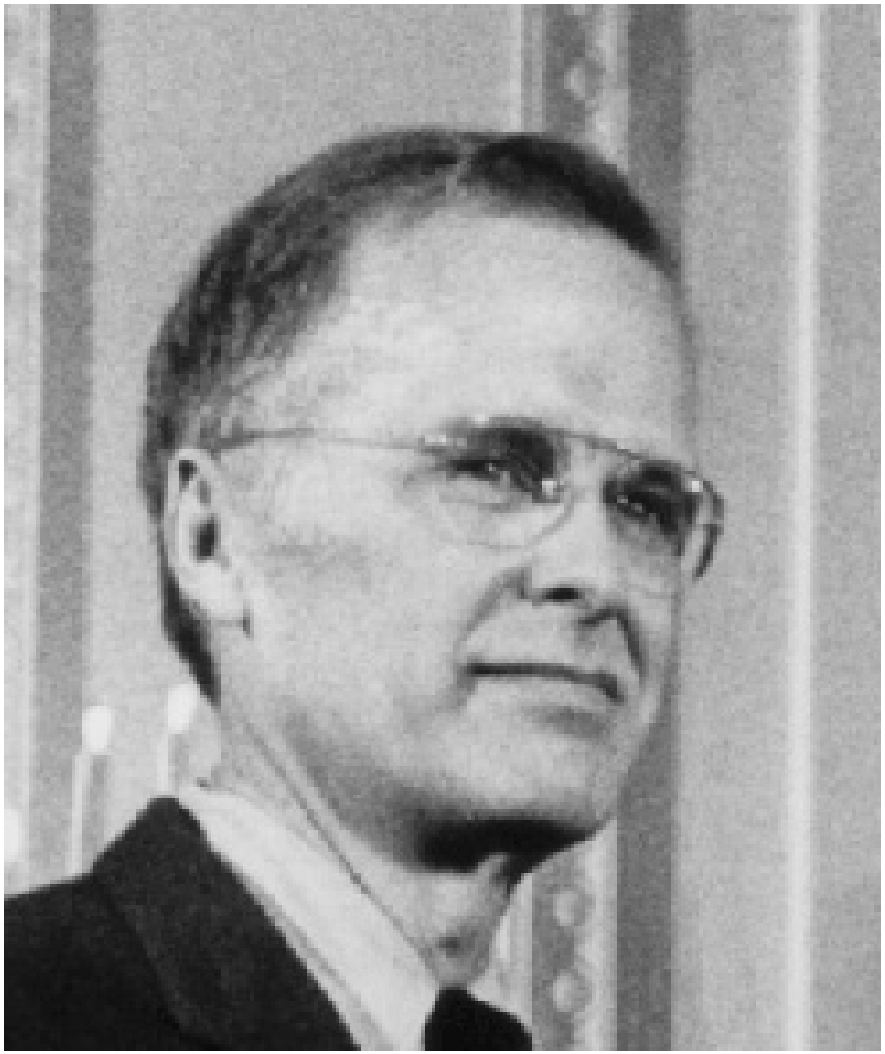
Examples

- “Low level” languages are typically compiled.
  - C, C++, Go, Rust
- “High level” languages are typically interpreted.
  - Python, Ruby
- Some languages are both compiled and interpreted
  - Java, Javascript - Interpreter + Just in Time (JIT) Compiler

Compilers: When?

- In 1954, IBM developed the 704, “the first mass-produced computer with floating-point arithmetic hardware” [Wikipedia].
  - Unfortunately, software costs would exceed hardware costs, since all programming was done in assembly.
- **John Backus** developed the FORTRAN I language (1957) for writing high-level code, and also a compiler for translating it to assembly.
  - Development time halved, with performance being close to the hand-written assembly!
  - Modern compilers preserve the outline of the FORTRAN I compiler
- Independently, in the 1950s, **Grace Hopper** developed the COBOL language and a compiler for it.

Images of the day



Turing Award Winners, Grace Hopper and John Backus

Compilers: Why?

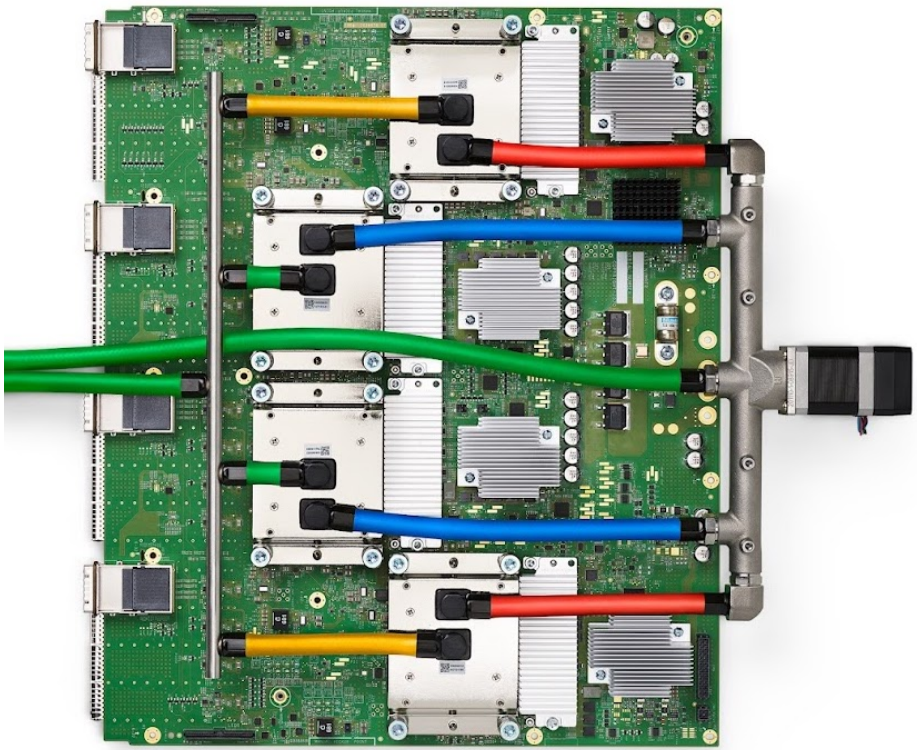
Isn't it a solved problem? "Optimization for scalar machines was solved years ago"

Machines have changed drastically in the last 20 years
Changes in architecture $\Rightarrow$ changes in compilers

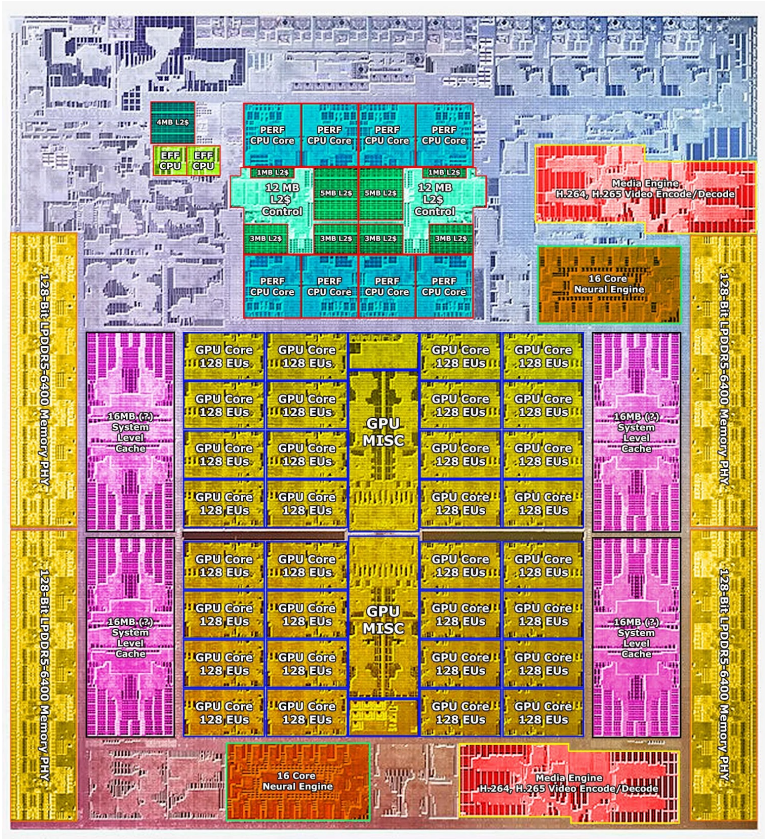
Major unsolved problems: Design of a programming language and its compiler for

- AI/ML
- many- and multi-core architectures
- intermittent computing
- FPGAs
- probabilisitic programming languages

AI/ML Hardware Innovation requires . . .



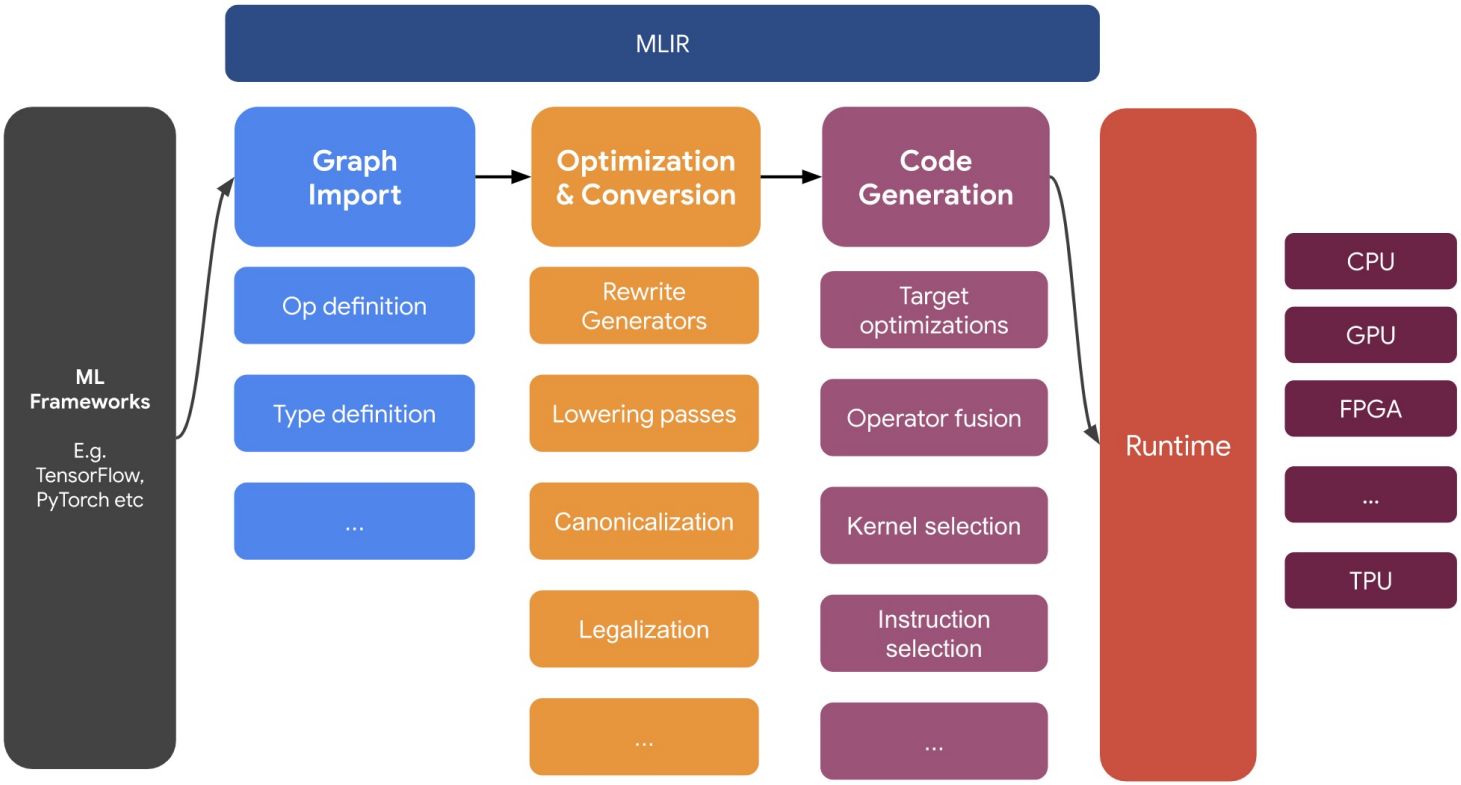
Google's Tensor Processing Unit (TPU)



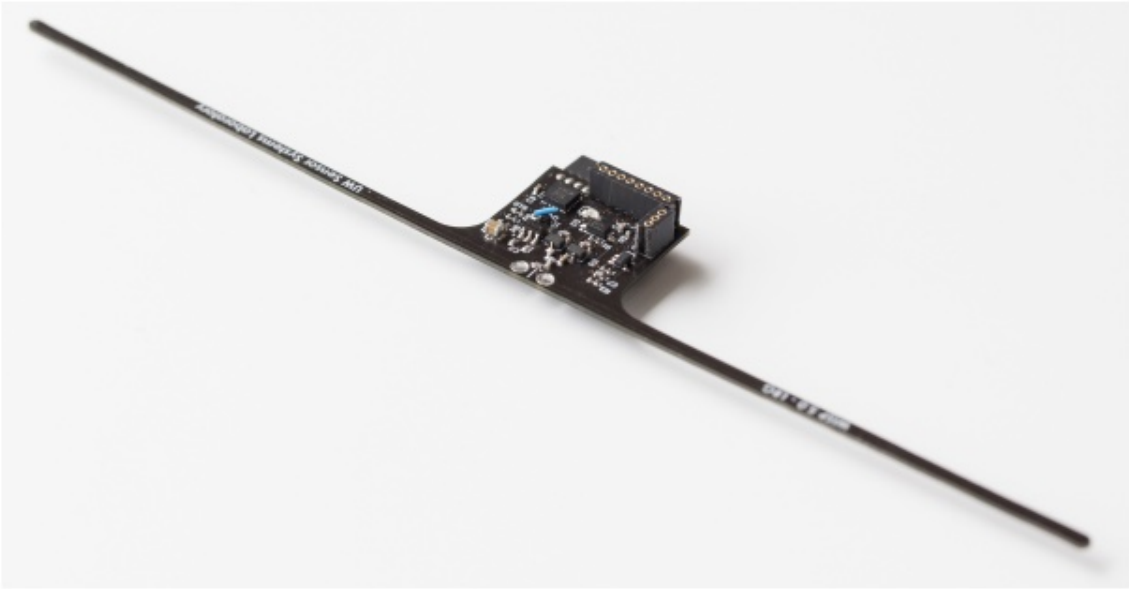
Apple's M1 Max Die Shot

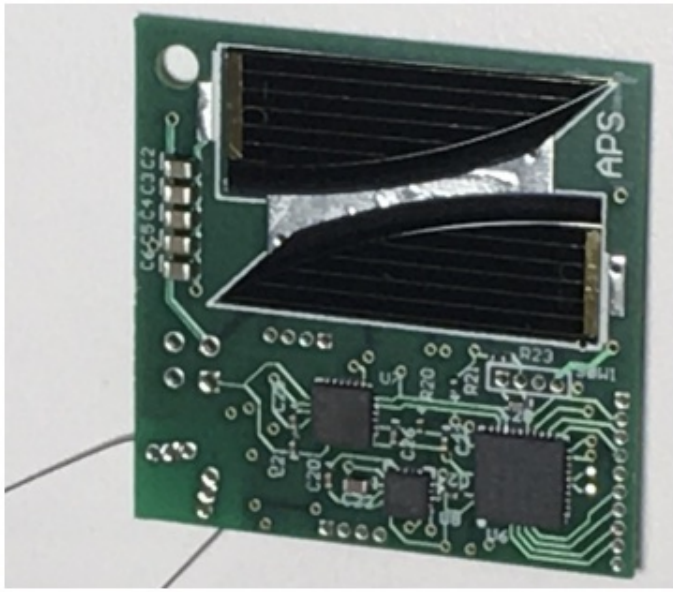
Compiler Innovation

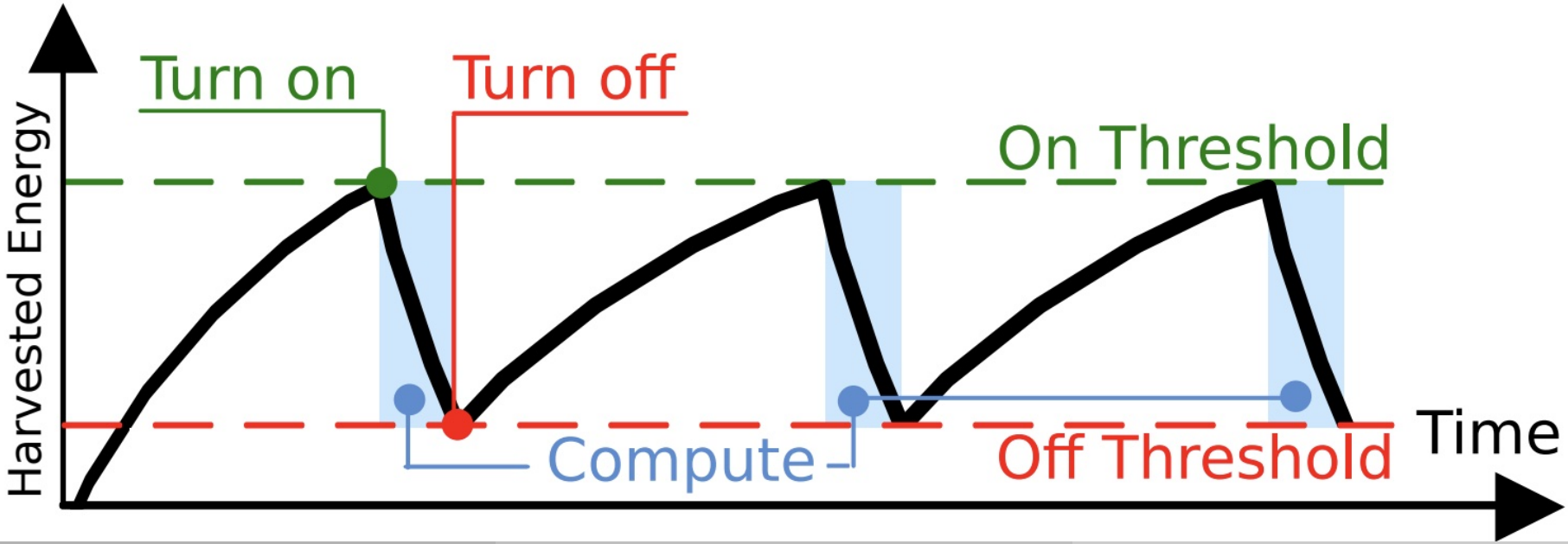
MLIR, or Multi-Level Intermediate Representation, sits between the model representation and low-level compilers/executors that generate hardware-specific code.



Intermittent computing







Interest

Compiler construction is a microcosm of computer science

- **Algo** graph algorithms, union-find, dynamic programming, ...
- **theory** DFAs for scanning, parser generators, lattice theory, ...
- **systems** allocation, locality, layout, synchronization, ...
- **architecture** pipeline management, hierarchy management, instruction set use, ...
- **optimizations** Operational research, load balancing, scheduling, ...

Inside a compiler, all these and many more come together. Has probably the healthiest mix of theory and practise.

Intrinsic Merit

Compiler Design is challenging and fun

- interesting problems
- primary responsibility (read:*blame*) for performance
- new architectures $\Rightarrow$ new challenges
- *real* results
- extremely complex interactions

Compilers have a major impact on how computers are used

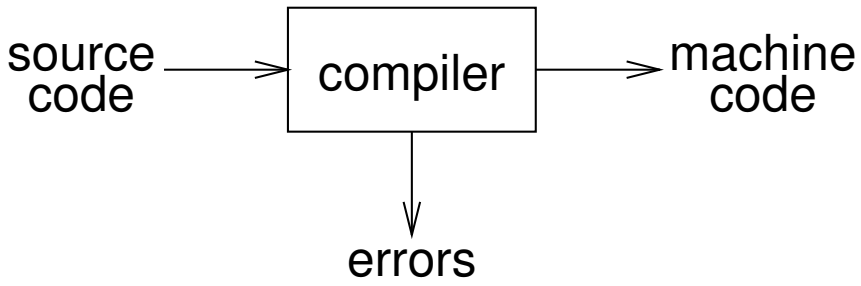
Overview of Compilers

Requirements

What qualities are important in a compiler?

1. Correct code
2. Output runs fast
3. Compiler runs fast
4. Compile time proportional to program size
5. Good diagnostics for syntax errors
6. Works well with the debugger
7. Consistent, predictable optimization

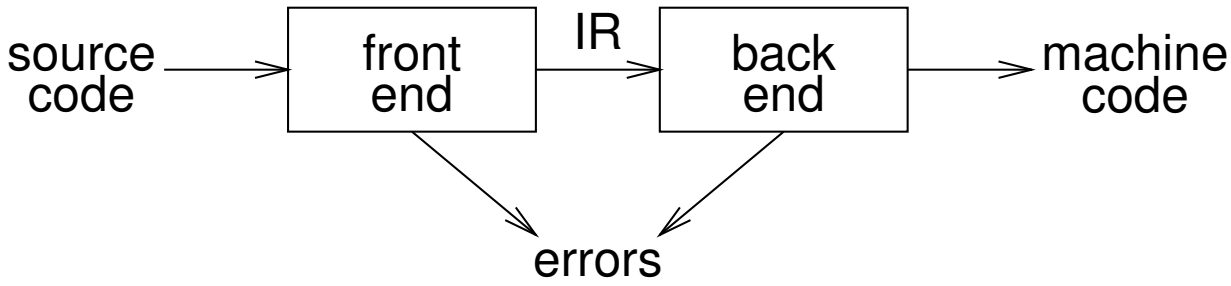
Abstract view



Implications:

- recognize legal (and illegal) programs
- generate correct code
- manage storage of all variables and code
- agreement on format for object (or assembly) code

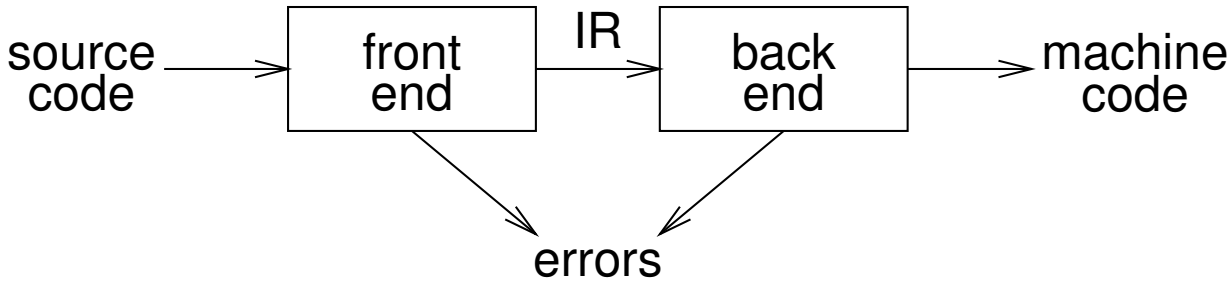
Traditional two pass compiler



Implications:

- intermediate representation (IR).
- front end maps legal code into IR
- back end maps IR onto target machine
- simplify retargeting
- allows multiple front ends
- multiple passes $\Rightarrow$ better code

Traditional two pass compiler

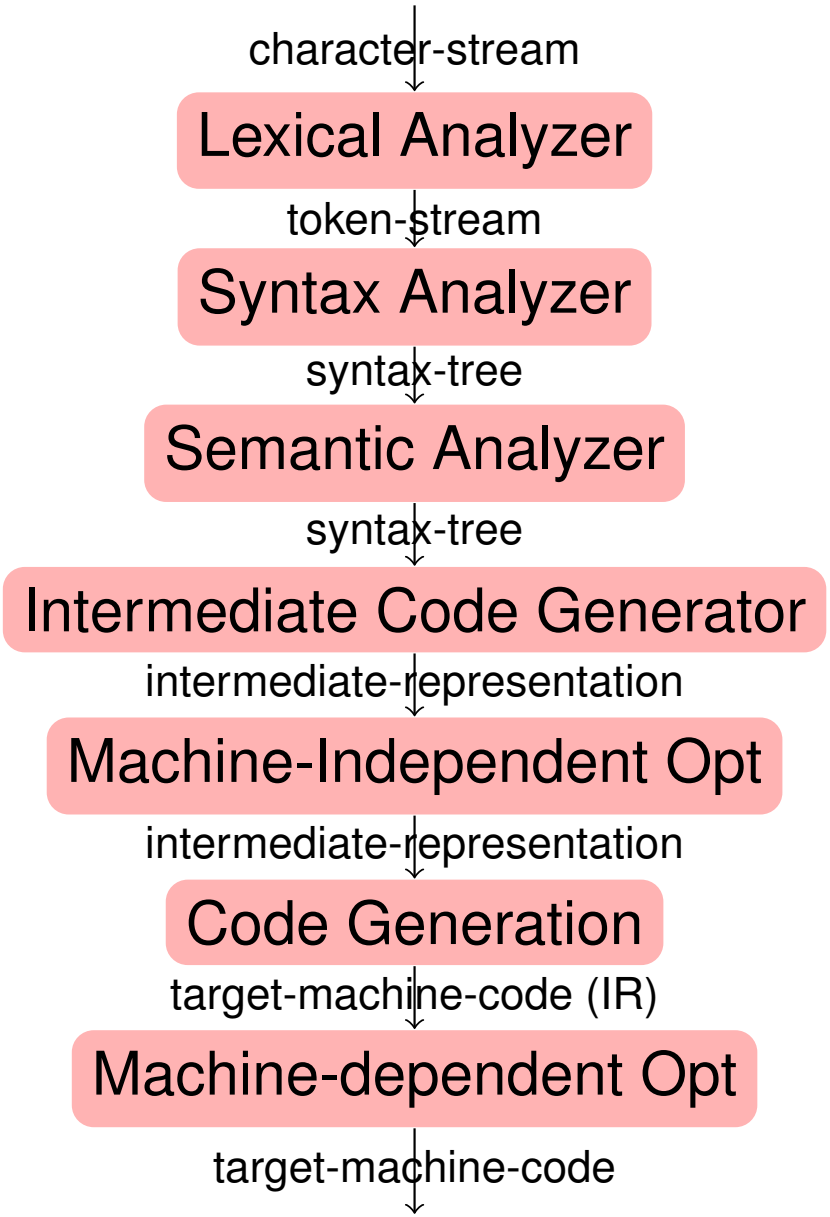


A rough statement: Most of the problems in the Front-end are simpler (polynomial time solution exists).

Most of the problems in the Back-end are harder (many problems are NP-complete in nature).

Our focus: Mainly front end and little bit of back end.

Phases inside the compiler



Front end responsibilities:

- Recognize syntactically legal code; report errors.
- Recognize semantically legal code; report errors.
- Produce IR.

Back end responsibilities:

- Optimizations, code generation.

Our target

- five out of seven phases.
- glance over optimizations

Major Phases in a Compiler

- Lexical Analysis
- Parsing
- Semantic Analysis
- Optimization
- Code Generation

Lexical Analysis

- To understand the different phases of a compiler, we will use as an analogy, how humans comprehend the English language.
- First step: recognize words in a sentence
  - Smallest unit above letters.
- All words in a sentence must be valid. Examples:
  - This is a sentence
  - Thsi si otn

Lexical Analysis for Programs

- Lexical Analyzer divides program text into ‘words’ or ‘tokens’ (in compiler terminology)
- For example, consider the program statement:
  `if x==y then z=1; else z=2;`
- Tokens
  - Keywords: `if`, `then`, `else`
  - Identifiers: `x`, `y`, `z`
  - Operators: `==`, `=`, `;`
  - Constants: `1`, `2`
  - Whitespace
- Lexical Analyzer also rejects a program if it has any invalid word.

Parsing

- Once words are understood, the next step is to understand the structure of the sentence.
- Parsing = Diagramming Sentences
  - The diagram will be a tree.

This is a sentence

Parsing Programs

- Parsing program statements is the same.

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

Semantic Analysis

- Once sentence structure is understood, we can try to understand its “meaning”.
  - Understanding the entire meaning of a program is too hard for compilers–in fact, undecidable.
- Compilers perform limited semantic analysis to catch inconsistencies.

Semantic Analysis in English

- Example: Ram said Shyam left his assignment at home.
  - What does “his” refer to? Ram or Shyam?
- Even worse: Ram said Ram left his assignment at home
  - How many Ram’s are there? Which one left his assignment?

Semantic Analysis in Programming

- Programming languages define strict rules to avoid such ambiguities.
- The C code prints “4”; the inner definition is used

```
{
  int X = 3;
  {
    int X = 4;
    printf("\%d", X);
  }
}
```

More Semantic Analysis

- Compilers perform many semantic checks besides variable bindings.
- Example: Shyam left her homework at home
- Possible type mismatch between her and Shyam

Optimization

- No strong counterpart in terms of our analogy, but we can think of it as editing to make sentences simpler.
- Automatically modify programs so that they
  - Run faster
  - Use less memory
  - In general, use of conserve some resource
- Example: `X = Y*0` is the same as `X=0`, but does not involve an additional multiplication, and is thus faster.
- In this course, we won't study optimization in detail. But there are advanced courses such as Modern Compilers: Theory and Practice for that purpose.

Code Generation

- Typically produces Assembly code.
- In terms of our analogy, translating the sentence in English into another language.

Intermediate Representations

- Compilers typically perform translations between successive intermediate languages.
  - All but the first and last are intermediate representations (IR) internal to the compiler.
- IRs are ordered in descending level of abstraction
  - Highest is source.
  - Lowest is assembly.
  - Other IRs that we will during the course: Token Stream, Syntax Tree, Three-Address Code.
- IRs are useful because lower levels expose features hidden by higher levels
  - Registers, Memory layout, raw pointers, etc.
- But lower levels obscure high-level meaning
  - Classes, Higher-order functions, Even loops...