CS3300 - Compiler Design

Introduction

Kartik Nagar

IIT Madras

Compilers: What?

- What is a compiler?

Compilers: What?

- What is a compiler?
  - a program that translates an executable program in one language into an executable program in another language.

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

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?

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

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

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.

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.

Examples

- "Low level" languages are typically compiled.
  - C, C++, Go, Rust

Examples

- “Low level” languages are typically compiled.
    - C, C++, Go, Rust
- “High level” languages are typically interpreted.

Examples

- “Low level” languages are typically compiled.
  - C, C++, Go, Rust
- “High level” languages are typically interpreted.
  - Python, Ruby

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

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.

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

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

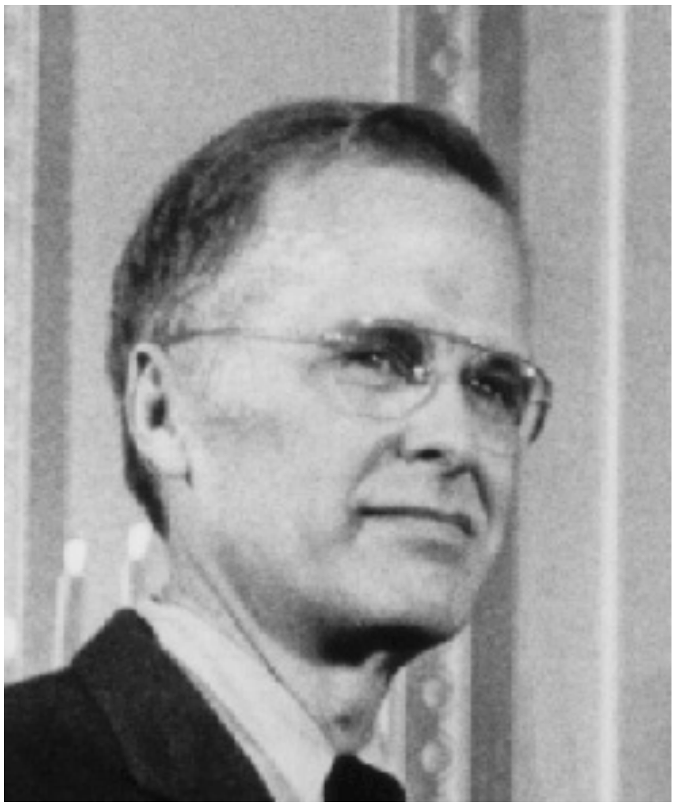


Figure: Turing Award Winners, Grace Hopper and John Backus

Compilers: Why?

Isn’t it a solved problem?

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

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

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 ⇒ changes in compilers
Major unsolved problem: Design of a programming language and its compiler which optimises the use of modern multi core and many core machines.

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 problem: Design of a programming language and its compiler which optimises the use of modern multi core and many core machines.

- new features pose new problems

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 problem: Design of a programming language and its compiler which optimises the use of modern multi core and many core machines.

- new features pose new problems
- changing concerns lead to new challenges: Security, correctness

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 ⇒ changes in compilers
Major unsolved problem: Design of a programming language and its compiler which optimises the use of modern multi core and many core machines.

- new features pose new problems
- changing concerns lead to new challenges: Security, correctness
- old solutions need re-engineering

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 problem: Design of a programming language and its compiler which optimises the use of modern multi core and many core machines.

- new features pose new problems
- changing concerns lead to new challenges: Security, correctness
- old solutions need re-engineering

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 ⇒ changes in compilers
Major unsolved problem: Design of a programming language and its compiler which optimises the use of modern multi core and many core machines.

- new features pose new problems
- changing concerns lead to new challenges: Security, correctness
- old solutions need re-engineering

Changes in compilers should prompt changes in architecture

- New languages and features

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 ⇒ new challenges
- *real* results
- extremely complex interactions

Compilers have a major impact on how computers are used

Requirements

What qualities are important in a compiler?

Requirements

What qualities are important in a compiler?

1. Correct code

Requirements

What qualities are important in a compiler?

1. Correct code
2. Output runs fast

Requirements

What qualities are important in a compiler?

1. Correct code
2. Output runs fast
3. Compiler runs fast

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

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

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

Requirements

What qualities are important in a compiler?

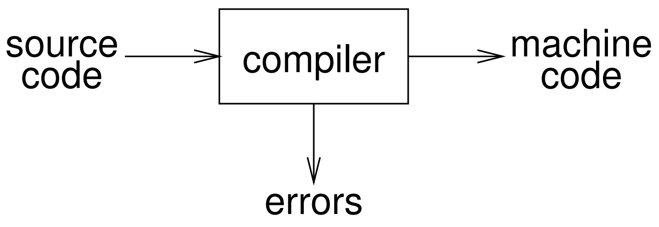
1. Correct code
2. Output runs fast
3. Compiler runs fast
4. Compile time proportional to program size
5. Good diagnostics for syntax errors
6. Works well with the debugger
7. Good diagnostics for flow anomalies

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. Good diagnostics for flow anomalies
8. Consistent, predictable optimization

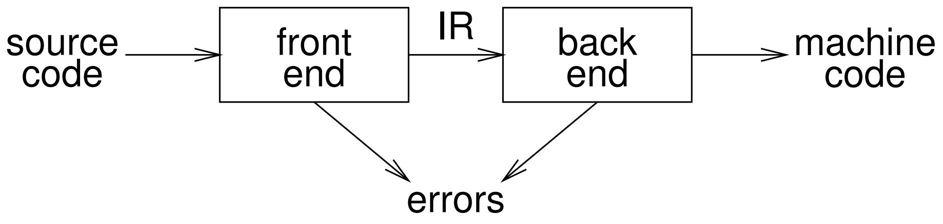
Abstract view



Implications:

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

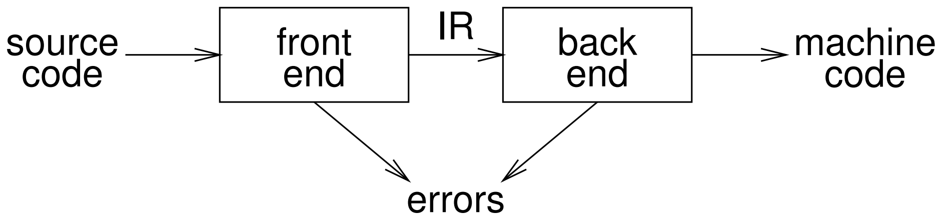
Traditional two pass compiler



Implications:

- intermediate representation (IR).

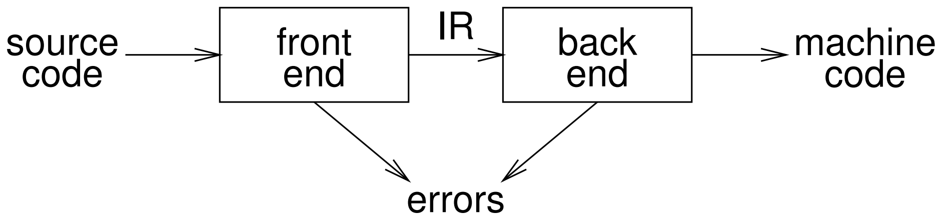
Traditional two pass compiler



Implications:

- intermediate representation (IR).
- front end maps legal code into IR

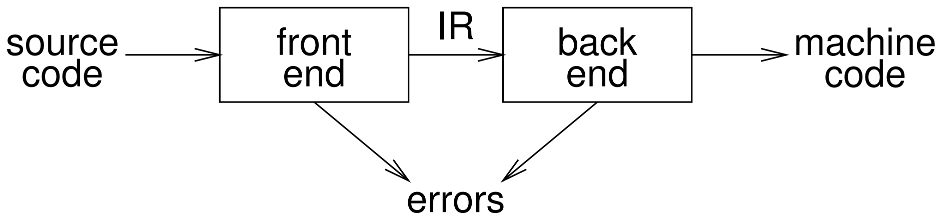
Traditional two pass compiler



Implications:

- intermediate representation (IR).
- front end maps legal code into IR
- back end maps IR onto target machine

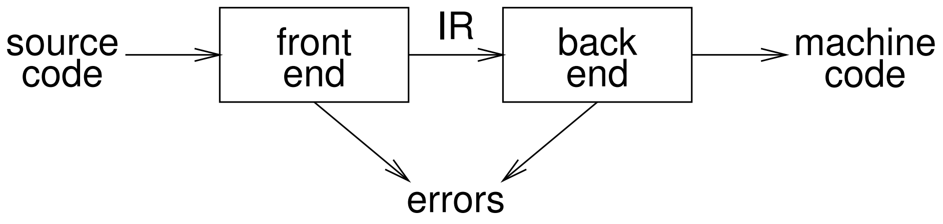
Traditional two pass compiler



Implications:

- intermediate representation (IR).
- front end maps legal code into IR
- back end maps IR onto target machine
- simplify retargeting

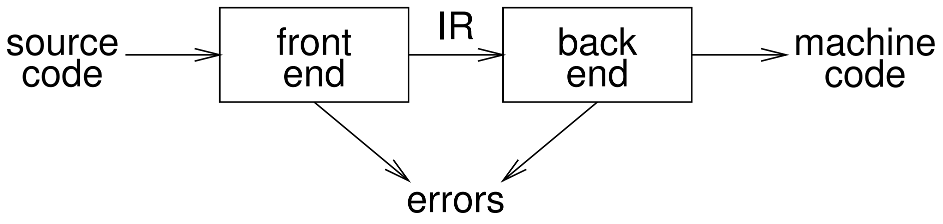
Traditional two pass compiler



Implications:

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

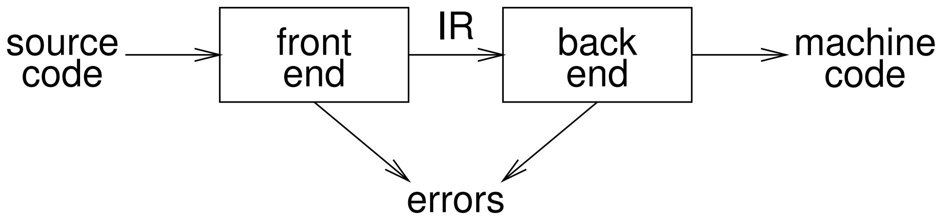
Traditional two pass compiler



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

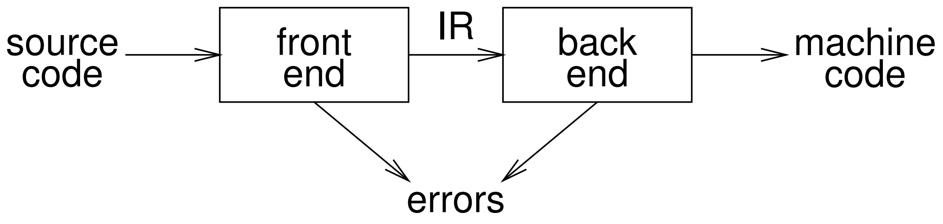
Traditional two pass compiler


$$\begin{tikzcd}
\text{source code} \arrow[r] & \boxed{\text{front end}} \arrow[r, "\text{IR}"] \arrow[dr] & \boxed{\text{back end}} \arrow[r] \arrow[d] & \text{machine code} \\
& & \text{errors} &
\end{tikzcd}$$

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 ⇒ better code

Traditional two pass compiler

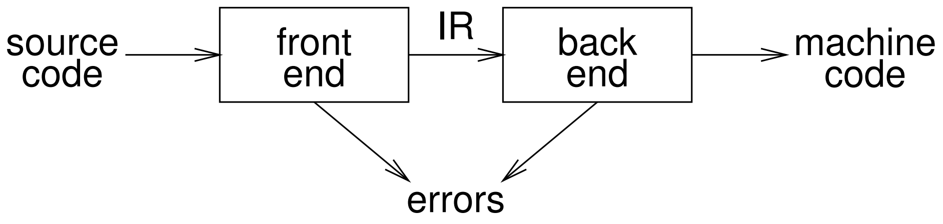


Implications:

- 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

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

Traditional two pass compiler

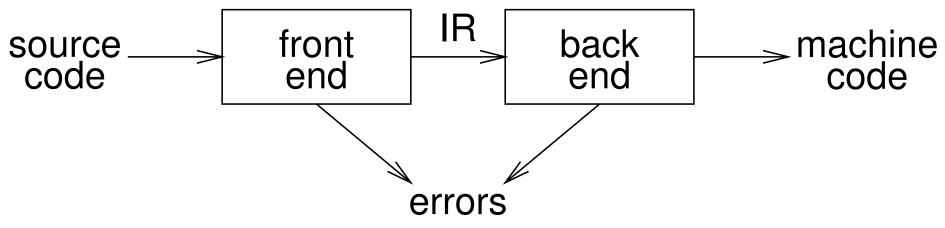


Implications:

- 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

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).

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

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.

Administrivia

- Lecture Timings
  - Slot B: Monday 9 AM, Wednesday 1 PM, Friday 11 AM
  - Online on Google Meet
- Course Webpage: https://kartiknagar.github.io/courses/compiler/
- Course Moodle page: TBD
  - Lecture slides, links to video lectures, etc. will be uploaded here.
- Course Google group: CS3300-Aug-Nov-2021
- Instructor e-mail address: nagark@cse.iitm.ac.in
  - Instructor Office Hours: None.
  - Feel free to e-mail me if you want to meet. TA Office hours will be announced soon.

Grading Policy (tentative)

- Theory: 60%, Lab: 40%
- Theory
  - Quiz 1: 14%, Quiz 2: 14%, Endsem: 30%
  - Class Participation: 2%.
  - Class Participation will be monitored throughout the semester. You can participate by asking/answering questions during the lectures and/or in the Google group forum.
- Lab: 5 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.

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**].
- 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 SDE like Eclipse is recommended.
- Flex, Bison, JavaCC, JTB – tools you will learn to use.
- Make / Ant / Scripts – recommended toolkit.

Acknowledgement

These slides are heavily adapted from the slides prepared by Prof. V Krishna Nandivada @ 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.