Lecture 1

CS 516: COMPILERS

Administrivia

Prof. Koskinen

Vidya Lakshmi Rajagopalan
Course Assistant

Benjamin Lirio
Course Assistant

Administrivia

Prof. Koskinen

Office Hours

Tuesdays 11:00am-12:00pm
& by appointment

https://stevens.zoom.us/j/96905914431

Vidya Lakshmi Rajagopalan

Course Assistant

https://stevens.zoom.us/j/96118015278

Benjamin Lirio

Course Assistant

Vidya Hours

Mondays 1-3pm

Ben Hours

Wednesdays 3-5pm

Announcements

- HW1: Hellocaml
  - available on the course web site soon
  - due ***next Thursday***, February 2nd at 11:59pm

- Course project infrastructure
  - OCaml and dune build system
  - Clone code from GitHub Classroom
  - Upload to Gradescope for project autograding

Announcements

Web docs: http://www.erickoskinen.com/compilers/23sp/

Assignment notifications, grades, etc.

Clarifications, questions, discussion.
Action item: Email Vidya or Ben for access.

I will try to record lectures.
https://stevens.zoom.us/j/96858931801

Code projects.

Why CS 516?

Why CS 516?

- You will learn:
  - Practical applications of theory
  - Lexing/Parsing/Interpreters
  - How high-level languages are implemented in machine language
  - (A subset of) Intel x86 architecture
  - More about common compilation tools like GCC and LLVM
  - A deeper understanding of code
  - A little about programming language semantics & types
  - Functional programming in OCaml
  - How to manipulate complex data structures
  - How to be a better programmer

Why CS 516?

- You will learn:
  - Practical applications of theory
  - Lexing/Parsing/Interpreters
  - How high-level languages are implemented in machine language
  - (A subset of) Intel x86 architecture
  - More about common compilation tools like GCC and LLVM
  - A deeper understanding of code
  - A little about programming language semantics & types
  - Functional programming in OCaml
  - How to manipulate complex data structures
  - How to be a better programmer

- Expect this to be a *very challenging*, implementation-oriented course.
  - Programming projects can take *tens* of hours per week…

The CS516 Compiler

- Course projects
  - HW1 75pts - HellOcaml
  - HW2 50pts - X86 Simulator
  - HW3 50pts - X86 Assembler
  - HW4 100pts - LLVM Lite (Large project)
  - HW5 50pts - Lexing
  - HW6 50pts - Frontend Compilation
  - HW7 50pts - Typechecking
  - HW8 50pts - Compiling structs and function pointers
  - HW9 50pts - Dataflow Analysis, Alias Analysis, Dead Code
  - HW10 50pts - Register Allocation and Experiments

- Goal: build a complete compiler from a high-level, type-safe language to x86 assembly.

Challenging & Rewarding

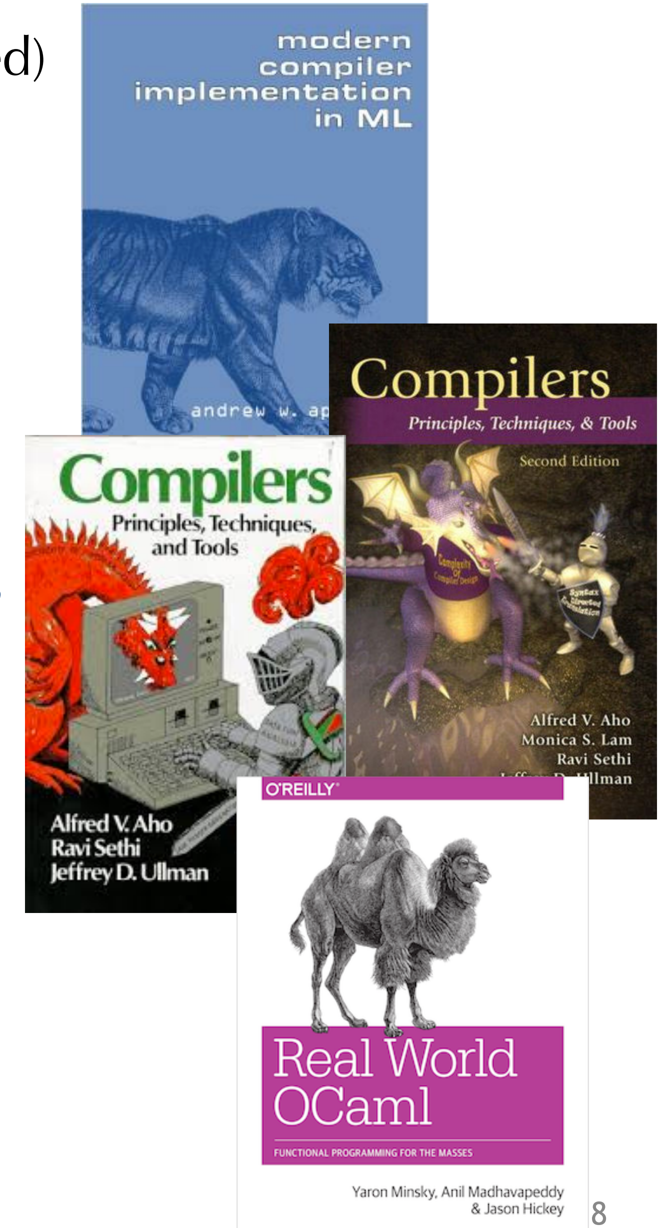
- What (anonymous) students said about this course:
  - "The course covers some extremely fundamental concepts to computer science."
  - "The projects were high quality."
  - "This was probably the most interesting and informative class I have taken so far; it is very fast paced but it is well worth it. The workload is extremely high, but it is engaging and rewarding."

- This course will be difficult.

- It is a 500-level course, so there is ***no such thing as a "D" grade***.

Resources

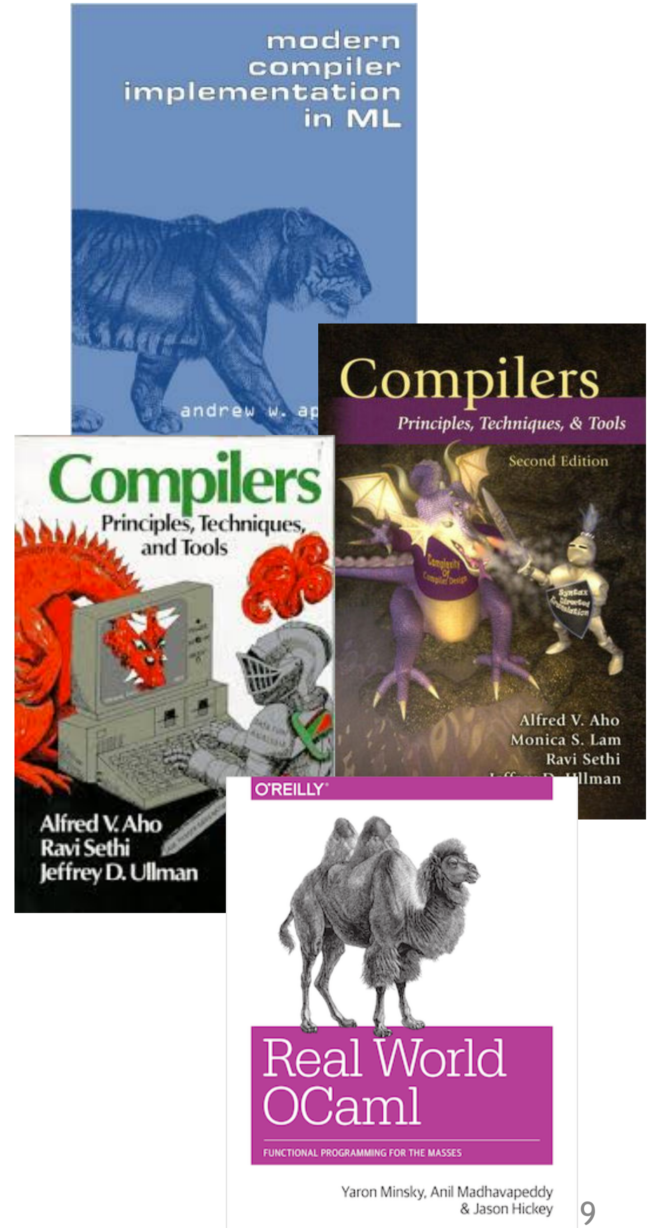
- Course textbook: (recommended, not required)
  - *Modern compiler implementation in ML* (Appel)
- Additional compilers books:
  - *Compilers – Principles, Techniques & Tools* (Aho, Lam, Sethi, Ullman)
    - *a.k.a. "The Dragon Book"*
  - *Advanced Compiler Design & Implementation* (Muchnick)
- About Ocaml:
  - *Real World Ocaml* (Minsky, Madhavapeddy, Hickey)
    - realworldocaml.org
  - *Introduction to Objective Caml* (Hickey)



Resources

Course Materials thanks to:

- Steve Zdancewic, UPenn (CIS341)
- Ilya Sergey, Yale-NUS
- Zachary Kinkaid, Princeton
- Andrew Myers, Cornell
- Greg Morrisett, Cornell
- Ahmal Ahmed, Northeastern
- Decades of work thanks to the PL community…



Why OCaml?

Why OCaml?

- OCaml is a dialect of ML – “Meta Language”
  - It was designed to enable easy manipulation *abstract syntax trees*
  - Type-safe, mostly pure, functional language with support for polymorphic (generic) algebraic datatypes, modules, and mutable state
  - The OCaml compiler itself is well engineered
    - you can study its source!
  - It is the right tool for this job

Why OCaml?

- OCaml is a dialect of ML – "Meta Language"
  - It was designed to enable easy manipulation *abstract syntax trees*
  - Type-safe, mostly pure, functional language with support for polymorphic (generic) algebraic datatypes, modules, and mutable state
  - The OCaml compiler itself is well engineered
    - you can study its source!
  - It is the right tool for this job

- New to OCaml? (Or forgot since CS496/CS510?)
  - Next couple lectures will (re)introduce it
  - First two projects will help you get up to speed programming
  - See "Introduction to Objective Caml" by Jason Hickey
    - book available on the course web pages, referred to in HW1

HW1: Hellocaml

HW1: Hellocaml

- Homework 1 is available on the course web site.
  - Individual project – no groups
  - Due: *next Thursday, 2 February 2023 at 11:59pm*
  - Topic: OCaml programming, an introduction to interpreters

HW1: Hellocaml

- Homework 1 is available on the course web site.
  - Individual project – no groups
  - Due: *next Thursday, 2 February 2023 at 11:59pm*
  - Topic: OCaml programming, an introduction to interpreters

HW1: Hellocaml

- Homework 1 is available on the course web site.
  - Individual project – no groups
  - Due: *next Thursday, 2 February 2023 at 11:59pm*
  - Topic: OCaml programming, an introduction to interpreters
- Logistics for getting access to Github Classroom will be posted soon.

HW1: Hellocaml

- Homework 1 is available on the course web site.
  - Individual project – no groups
  - Due: *next Thursday, 2 February 2023 at 11:59pm*
  - Topic: OCaml programming, an introduction to interpreters

- Logistics for getting access to Github Classroom will be posted soon.

HW1: Hellocaml

- Homework 1 is available on the course web site.
  - Individual project – no groups
  - Due: *next Thursday, 2 February 2023 at 11:59pm*
  - Topic: OCaml programming, an introduction to interpreters

- Logistics for getting access to Github Classroom will be posted soon.

- We recommend using VSCode + OCaml Platform
  - See the course web pages about the CS516 tool chain to get started

HW1: Hellocaml

- Homework 1 is available on the course web site.
  - Individual project – no groups
  - Due: *next Thursday, 2 February 2023 at 11:59pm*
  - Topic: OCaml programming, an introduction to interpreters

- Logistics for getting access to Github Classroom will be posted soon.

- We recommend using VSCode + OCaml Platform
  - See the course web pages about the CS516 tool chain to get started

HW1: Hellocaml

- Homework 1 is available on the course web site.
  - Individual project – no groups
  - Due: *next Thursday, 2 February 2023 at 11:59pm*
  - Topic: OCaml programming, an introduction to interpreters

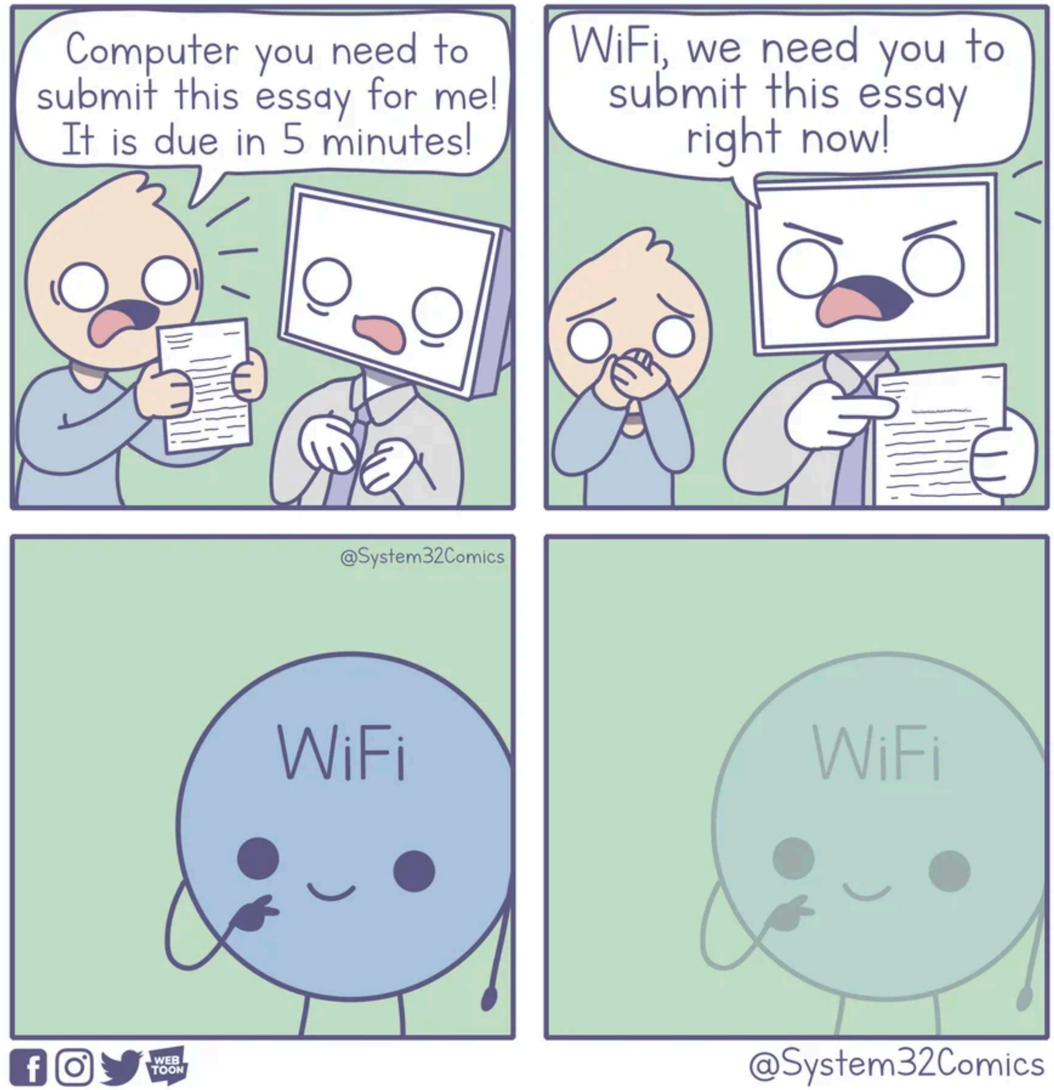
- Logistics for getting access to Github Classroom will be posted soon.

- We recommend using VSCode + OCaml Platform
  - See the course web pages about the CS516 tool chain to get started

- Quickstart guide:
  - open up the project in VSCode
  - start a "sandbox terminal" via OCaml Platform plugin
  - type **`make test`** at the command prompt
  - Please: Use Slack to report any troubles with the toolchain!

Homework Policies

- Homework (except HW1) may be done individually or in pairs
- Late projects:
  - up to 24 hours late:
    10 point penalty
  - up to 48 hours late:
    20 point penalty
  - after 48 hours:
    not accepted



Homework Policies

Homework Policies

- Homework (except HW1) may be done individually or in pairs

Homework Policies

- Homework (except HW1) may be done individually or in pairs
- Budget of 7 "extra days":
  - You are given a budget of seven (7) extra days that you can use during the semester to extend the deadlines of projects (except for Homework 1).
  - e.g. you might decide to extend HW3 by 2 days, HW5 by 1 day and HW8 by 4 days.
  - After that, you will not be able to extend deadlines any more. At that point submissions will be considered **late** and subject to the following penalties.

Homework Policies

- Homework (except HW1) may be done individually or in pairs
- Budget of 7 "extra days":
  - You are given a budget of seven (7) extra days that you can use during the semester to extend the deadlines of projects (except for Homework 1).
  - e.g. you might decide to extend HW3 by 2 days, HW5 by 1 day and HW8 by 4 days.
  - After that, you will not be able to extend deadlines any more. At that point submissions will be considered **late** and subject to the following penalties.
- Late projects:
  - up to 24 hours late: 10 point penalty
  - up to 48 hours late: 20 point penalty
  - after 48 hours: not accepted

Homework Policies

- Homework (except HW1) may be done individually or in pairs
- Budget of 7 “extra days”:
  - You are given a budget of seven (7) extra days that you can use during the semester to extend the deadlines of projects (except for Homework 1).
  - e.g. you might decide to extend HW3 by 2 days, HW5 by 1 day and HW8 by 4 days.
  - After that, you will not be able to extend deadlines any more. At that point submissions will be considered **late** and subject to the following penalties.
- Late projects:
  - up to 24 hours late: 10 point penalty
  - up to 48 hours late: 20 point penalty
  - after 48 hours: not accepted
- Submission policy:
  - Projects that don’t compile will get no credit
  - Partial credit will be awarded according to the guidelines in the project description

Homework Policies

- Homework (except HW1) may be done individually or in pairs
- Budget of 7 "extra days":
  - You are given a budget of seven (7) extra days that you can use during the semester to extend the deadlines of projects (except for Homework 1).
  - e.g. you might decide to extend HW3 by 2 days, HW5 by 1 day and HW8 by 4 days.
  - After that, you will not be able to extend deadlines any more. At that point submissions will be considered **late** and subject to the following penalties.
- Late projects:
  - up to 24 hours late: 10 point penalty
  - up to 48 hours late: 20 point penalty
  - after 48 hours: not accepted
- Submission policy:
  - Projects that don't compile will get no credit
  - Partial credit will be awarded according to the guidelines in the project description
- Academic integrity: don't cheat
  - This course will abide by the Honor Code
  - "low level" and "high level" discussions across groups are fine
  - "mid level" discussions / code sharing are not permitted
  - General principle: *When in doubt, ask!*

Homework Due Dates

Points	Description	Due
75pts	HW1: Hellocaml	2/2
50pts	HW2: X86 Simulator	2/9
50pts	HW3: X86 Assembler	2/16
100pts	HW4: LLVM Lite	3/2
50pts	HW5: Lexing	3/9
50pts	HW6: Frontend Compilation	3/30
50pts	HW7: Typechecking	4/6
50pts	HW8: Compiling structs and function pointers	4/13
50pts	HW9: Dataflow Analysis, Alias Analysis, Dead Code	4/20
50pts	HW10: Register Allocation and Experiments	4/27

Homework Due Dates

Points	Description	Due
75pts	HW1: Hellocaml	2/2
50pts	HW2: X86 Simulator	2/9
50pts	HW3: X86 Assembler	2/16
100pts	HW4: LLVM Lite	3/2
50pts	HW5: Lexing	3/9
50pts	HW6: Frontend Compilation	3/30
50pts	HW7: Typechecking	4/6
50pts	HW8: Compiling structs and function pointers	4/13
50pts	HW9: Dataflow Analysis, Alias A	
50pts	HW10: Register Allocation and	

On most Thursdays
there will be a homework due

Course Policies

Prerequisites: Automata (CS 334) and Algorithms (CS 385 or CS570 or CS590)

- Significant programming experience
- If HW1 is a struggle, this class might not be a good fit for you (HW1 is significantly simpler than the rest…)

Grading:

70%	Projects: Compiler	Groups of 1 or 2 students Implemented in OCaml
2%	Attendance/Participation (randomized)	
28%	Quizzes	Frequent low-stakes assessment

Lecture attendance is crucial.

- Active participation (asking questions, etc.) is encouraged
- When in person, *no laptops (or other devices)*!
- It's too distracting for me and for others in the class.

Frequent Low-Stakes Assessment

- Exams are just snapshots of knowledge.
- Quizzes more reflective of your “learning journey and skill mastery”
- Less stressful for you.
- Good feedback for me.
  - I use results to shape next lectures/assignments.

- These are ***learning opportunities***!
  - Study regularly; not cram. Far less stressful.
  - You will sometimes feedback and be asked to reflect on it.

Syllabus

- PDF in Canvas
- Full details in the course documents:

http://www.erickoskinen.com/compilers/23sp/

CS 516: COMPILERS

Lecture 1

Topics

- What is a compiler?
- Introduction to OCaml programming

Materials

- lec01.zip (factorial.c, etc.)
- intro.ml

What is a compiler?

COMPILERS

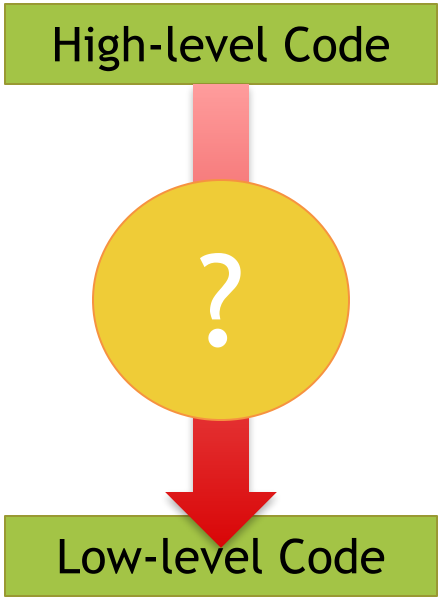
What is a Compiler?

What is a Compiler?

- A compiler is a program that translates from one programming language to another.

What is a Compiler?

- A compiler is a program that translates from one programming language to another.
- Typically: *high-level source code* to *low-level machine code* (object code)
  - Not always: Source-to-source translators, Java bytecode compiler, GWT Java ⇒ Javascript



Historical Aside

- This is an old problem!
- Until the 1950's: computers were programmed in assembly.
- 1951—1952: Grace Hopper developed the A-0 system for the UNIVAC I
  - She later contributed significantly to the design of COBOL
- 1957: the FORTRAN compiler was built at IBM
  - Team led by John Backus
- 1960's: development of the first bootstrapping compiler for LISP
- 1970's: language/compiler design blossomed

- Today: *thousands* of languages (most little used)
  - Some better designed than others...

Historical Aside

- This is an old problem!
- Until the 1950's: computers were programmed in assembly.
- 1951—1952: Grace Hopper developed the A-0 system for the UNIVAC I
  - She later contributed significantly to the design of COBOL
- 1957: the FORTRAN compiler was built at IBM
  - Team led by John Backus
- 1960's: development of the first bootstrapping compiler for LISP
- 1970's: language/compiler design blossomed
- Today: *thousands* of languages (most little used)
  - Some better designed than others...

1980s: ML / LCF
1984: Standard ML
1987: Caml
1991: Caml Light
1995: Caml Special Light
1996: Objective Caml
2005: F# (Microsoft)
2020: OCaml Platform

Source Code

- Optimized for human readability

```
#include <stdio.h>

int factorial(int n) {
  int acc = 1;
  while (n > 0) {
    acc = acc * n;
    n = n - 1;
  }
  return acc;
}

int main(int argc, char *argv[]) {
  printf("factorial(6) = %d\n", factorial(6));
}
```

Source Code

- Optimized for human readability
  - *Expressive*: matches human ideas of grammar / syntax / meaning

```
#include <stdio.h>

int factorial(int n) {
  int acc = 1;
  while (n > 0) {
    acc = acc * n;
    n = n - 1;
  }
  return acc;
}

int main(int argc, char *argv[]) {
  printf("factorial(6) = %d\n", factorial(6));
}
```

Source Code

- Optimized for human readability
  - *Expressive*: matches human ideas of grammar / syntax / meaning
  - *Redundant*: more information than needed to help catch errors

```
#include <stdio.h>

int factorial(int n) {
  int acc = 1;
  while (n > 0) {
    acc = acc * n;
    n = n - 1;
  }
  return acc;
}

int main(int argc, char *argv[]) {
  printf("factorial(6) = %d\n", factorial(6));
}
```

Source Code

- Optimized for human readability
  - *Expressive*: matches human ideas of grammar / syntax / meaning
  - *Redundant*: more information than needed to help catch errors
  - *Abstract*: exact computation possibly not fully determined by code

```
#include <stdio.h>

int factorial(int n) {
  int acc = 1;
  while (n > 0) {
    acc = acc * n;
    n = n - 1;
  }
  return acc;
}

int main(int argc, char *argv[]) {
  printf("factorial(6) = %d\n", factorial(6));
}
```

Source Code

- Optimized for human readability
  - *Expressive*: matches human ideas of grammar / syntax / meaning
  - *Redundant*: more information than needed to help catch errors
  - *Abstract*: exact computation possibly not fully determined by code
- Example C source:

```c
#include <stdio.h>

int factorial(int n) {
  int acc = 1;
  while (n > 0) {
    acc = acc * n;
    n = n - 1;
  }
  return acc;
}

int main(int argc, char *argv[]) {
  printf("factorial(6) = %d\n", factorial(6));
}
```

Low-level code

- Optimized for Hardware
  - Machine code hard for people to read
  - Redundancy, ambiguity reduced
  - Abstractions & information about intent is lost
- Assembly language
  - then machine language
- Figure at right shows (unoptimized) 32-bit code for the factorial function

```
_factorial:
## BB#0:
    pushl    %ebp
    movl %esp, %ebp
    subl $8, %esp
    movl 8(%ebp), %eax
    movl %eax, -4(%ebp)
    movl $1, -8(%ebp)
LBB0_1:
    cmpl $0, -4(%ebp)
    jle  LBB0_3
## BB#2:
    movl -8(%ebp), %eax
    imull    -4(%ebp), %eax
    movl %eax, -8(%ebp)
    movl -4(%ebp), %eax
    subl $1, %eax
    movl %eax, -4(%ebp)
    jmp  LBB0_1
LBB0_3:
    movl -8(%ebp), %eax
    addl $8, %esp
    popl %ebp
    retl
```

How to translate?

How to translate?

- Source code – Machine code mismatch

How to translate?

- Source code – Machine code mismatch
- Some languages are farther from machine code than others:
  - Consider: C, C++, Java, Lisp, ML, Haskell, Ruby, Python, Javascript

How to translate?

- Source code – Machine code mismatch
- Some languages are farther from machine code than others:
  - Consider: C, C++, Java, Lisp, ML, Haskell, Ruby, Python, Javascript

- Goals of translation:
  - Source level expressiveness for the task
  - Best performance for the concrete computation
  - Reasonable translation efficiency (< $O(n^3)$)
  - Maintainable code
  - Correctness!

Correct Compilation

Correct Compilation

- Programming languages describe computation precisely…
  - therefore, *translation* can be precisely described
  - a compiler can be correct with respect to the source and target language semantics.

Correct Compilation

- Programming languages describe computation precisely…
  - therefore, *translation* can be precisely described
  - a compiler can be correct with respect to the source and target language semantics.

- Correctness is important!
  - Broken compilers generate broken code.
  - Hard to debug source programs if the compiler is incorrect.
  - Failure has dire consequences for development cost, security, etc.

Correct Compilation

- Programming languages describe computation precisely…
  - therefore, *translation* can be precisely described
  - a compiler can be correct with respect to the source and target language semantics.

- Correctness is important!
  - Broken compilers generate broken code.
  - Hard to debug source programs if the compiler is incorrect.
  - Failure has dire consequences for development cost, security, etc.

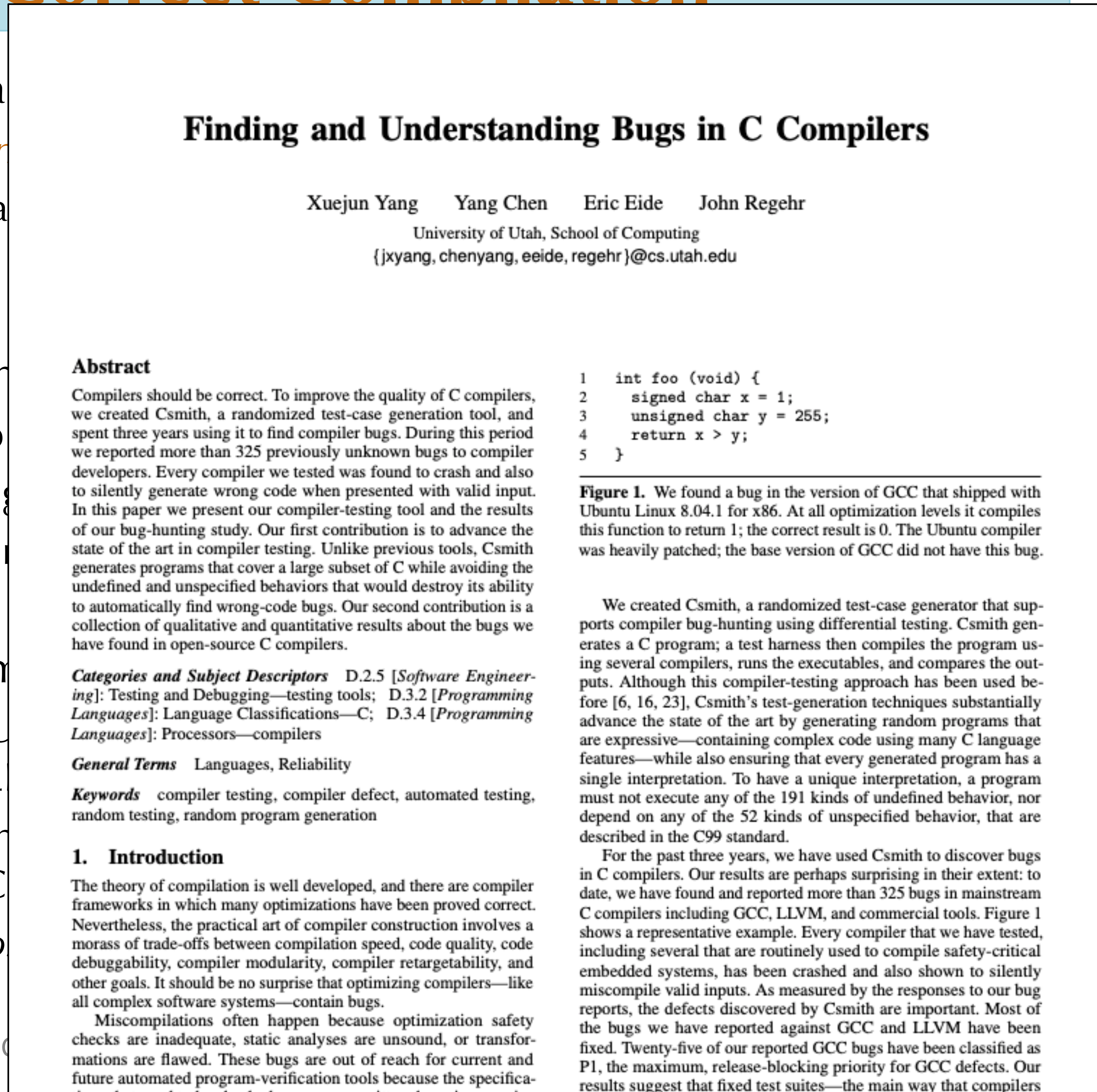
- This course: some techniques for building correct compilers
  - *Finding and Understanding Bugs in C Compilers*,
    Yang et al. PLDI 2011
  - There is much ongoing research about *proving* compilers correct.
    (Google for CompCert, Verified Software Toolchain, or Vellvm)
  - *Decompilation* of binaries

Correct Compilation

- Programming la
  - therefore, *trar*
  - a compiler ca semantics.

- Correctness is ir
  - Broken comp
  - Hard to debu
  - Failure has di

- This course: son
  - *Finding and L* Yang et al. PL
  - There is much (Google for C
  - *Decompilatio*



Finding and Understanding Bugs in C Compilers

Xuejun Yang Yang Chen Eric Eide John Regehr
University of Utah, School of Computing
{jxyang, chenyang, eeide, regehr}@cs.utah.edu

Abstract

Compilers should be correct. To improve the quality of C compilers, we created Csmith, a randomized test-case generation tool, and spent three years using it to find compiler bugs. During this period we reported more than 325 previously unknown bugs to compiler developers. Every compiler we tested was found to crash and also to silently generate wrong code when presented with valid input. In this paper we present our compiler-testing tool and the results of our bug-hunting study. Our first contribution is to advance the state of the art in compiler testing. Unlike previous tools, Csmith generates programs that cover a large subset of C while avoiding the undefined and unspecified behaviors that would destroy its ability to automatically find wrong-code bugs. Our second contribution is a collection of qualitative and quantitative results about the bugs we have found in open-source C compilers.

Categories and Subject Descriptors D.2.5 [*Software Engineering*]: Testing and Debugging—testing tools; D.3.2 [*Programming Languages*]: Language Classifications—C; D.3.4 [*Programming Languages*]: Processors—compilers

General Terms Languages, Reliability

Keywords compiler testing, compiler defect, automated testing, random testing, random program generation

1. Introduction

The theory of compilation is well developed, and there are compiler frameworks in which many optimizations have been proved correct. Nevertheless, the practical art of compiler construction involves a morass of trade-offs between compilation speed, code quality, code debuggability, compiler modularity, compiler retargetability, and other goals. It should be no surprise that optimizing compilers—like all complex software systems—contain bugs.

Miscompilations often happen because optimization safety checks are inadequate, static analyses are unsound, or transformations are flawed. These bugs are out of reach for current and future automated program-verification tools because the specifica-

```
int foo (void) {
  signed char x = 1;
  unsigned char y = 255;
  return x > y;
}
```

Figure 1. We found a bug in the version of GCC that shipped with Ubuntu Linux 8.04.1 for x86. At all optimization levels it compiles this function to return 1; the correct result is 0. The Ubuntu compiler was heavily patched; the base version of GCC did not have this bug.

We created Csmith, a randomized test-case generator that supports compiler bug-hunting using differential testing. Csmith generates a C program; a test harness then compiles the program using several compilers, runs the executables, and compares the outputs. Although this compiler-testing approach has been used before [6, 16, 23], Csmith's test-generation techniques substantially advance the state of the art by generating random programs that are expressive—containing complex code using many C language features—while also ensuring that every generated program has a single interpretation. To have a unique interpretation, a program must not execute any of the 191 kinds of undefined behavior, nor depend on any of the 52 kinds of unspecified behavior, that are described in the C99 standard.

For the past three years, we have used Csmith to discover bugs in C compilers. Our results are perhaps surprising in their extent: to date, we have found and reported more than 325 bugs in mainstream C compilers including GCC, LLVM, and commercial tools. Figure 1 shows a representative example. Every compiler that we have tested, including several that are routinely used to compile safety-critical embedded systems, has been crashed and also shown to silently miscompile valid inputs. As measured by the responses to our bug reports, the defects discovered by Csmith are important. Most of the bugs we have reported against GCC and LLVM have been fixed. Twenty-five of our reported GCC bugs have been classified as P1, the maximum, release-blocking priority for GCC defects. Our results suggest that fixed test suites—the main way that compilers

Idea: Translate in Steps

Idea: Translate in Steps

- Compile via a series of program representations

Idea: Translate in Steps

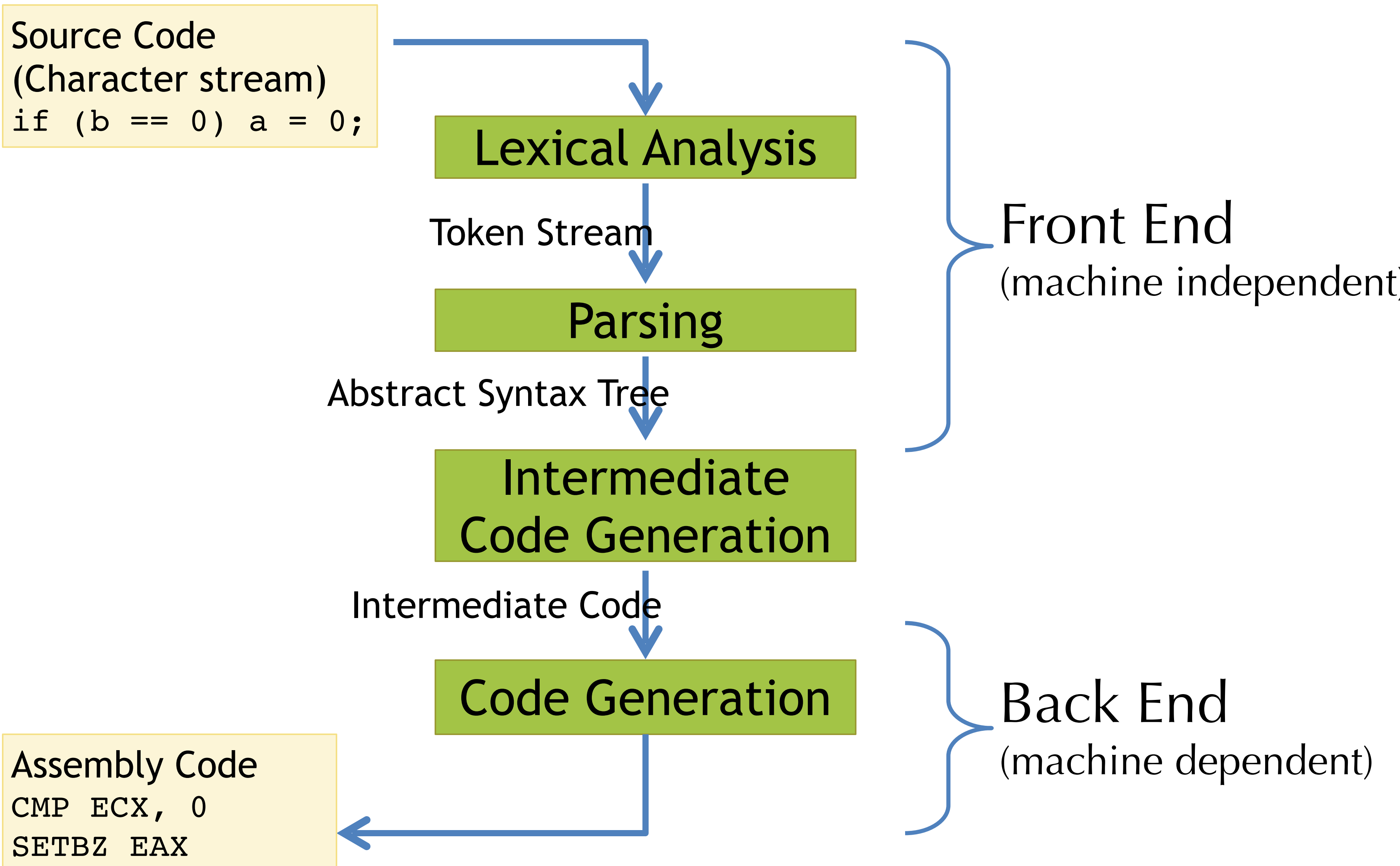
- Compile via a series of program representations

- Intermediate representations are optimized for program manipulation of various kinds:
  - Semantic analysis: type checking, error checking, etc.
  - Optimization: dead-code elimination, common subexpression elimination, function inlining, register allocation, etc.
  - Code generation: instruction selection

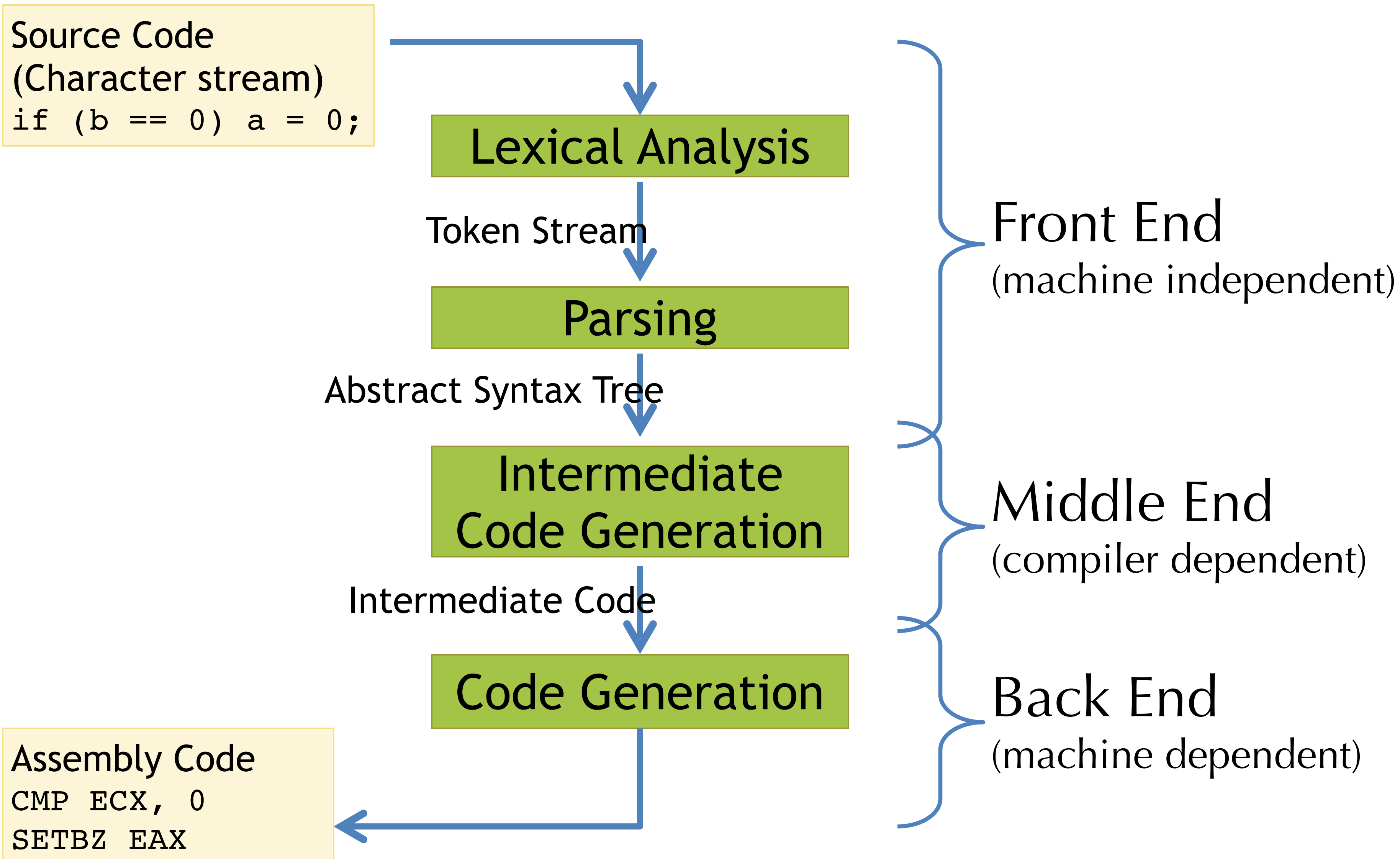
Idea: Translate in Steps

- Compile via a series of program representations
- Intermediate representations are optimized for program manipulation of various kinds:
  - Semantic analysis: type checking, error checking, etc.
  - Optimization: dead-code elimination, common subexpression elimination, function inlining, register allocation, etc.
  - Code generation: instruction selection
- Representations are more machine specific, less language specific as translation proceeds

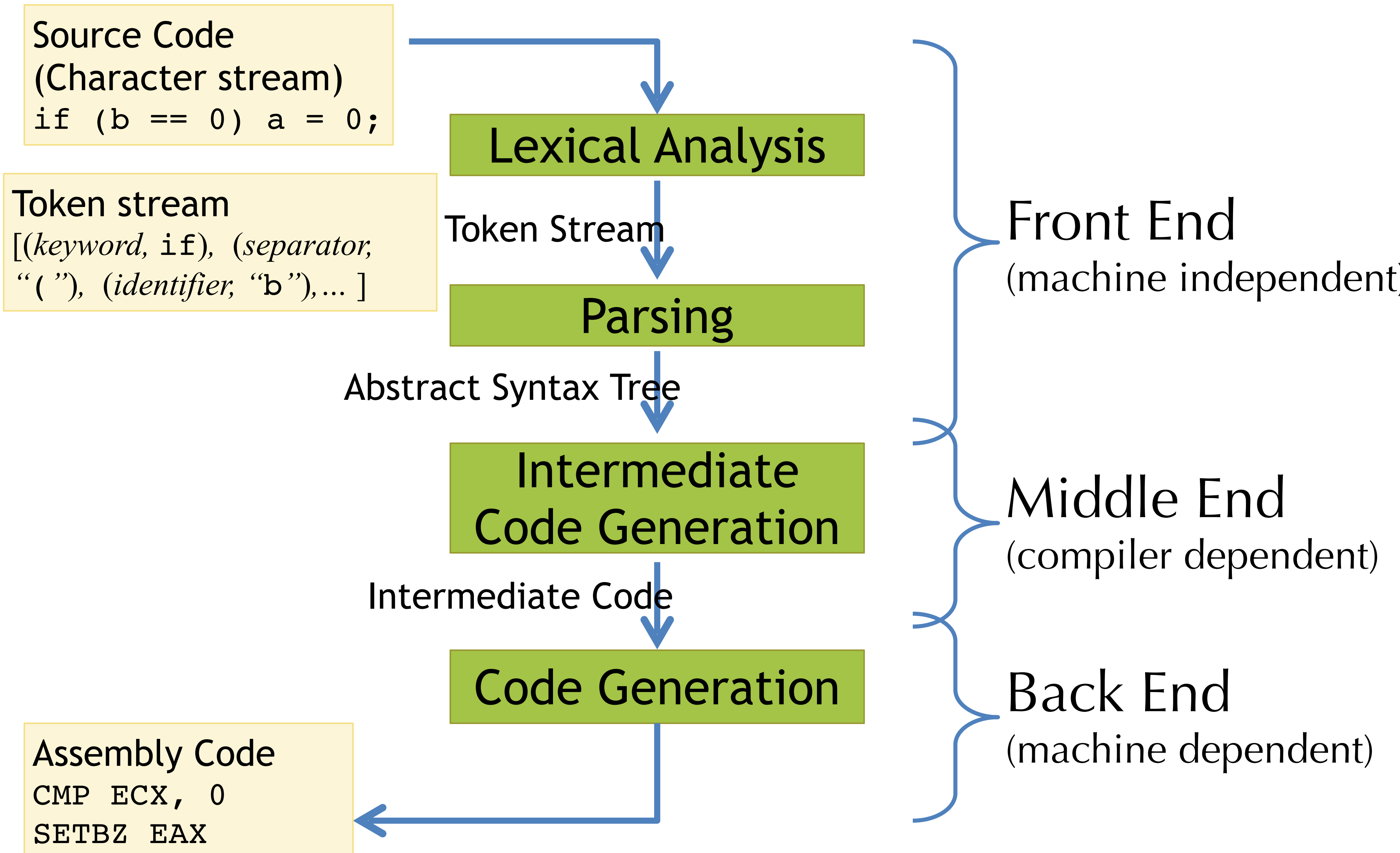
(Simplified) Compiler Structure



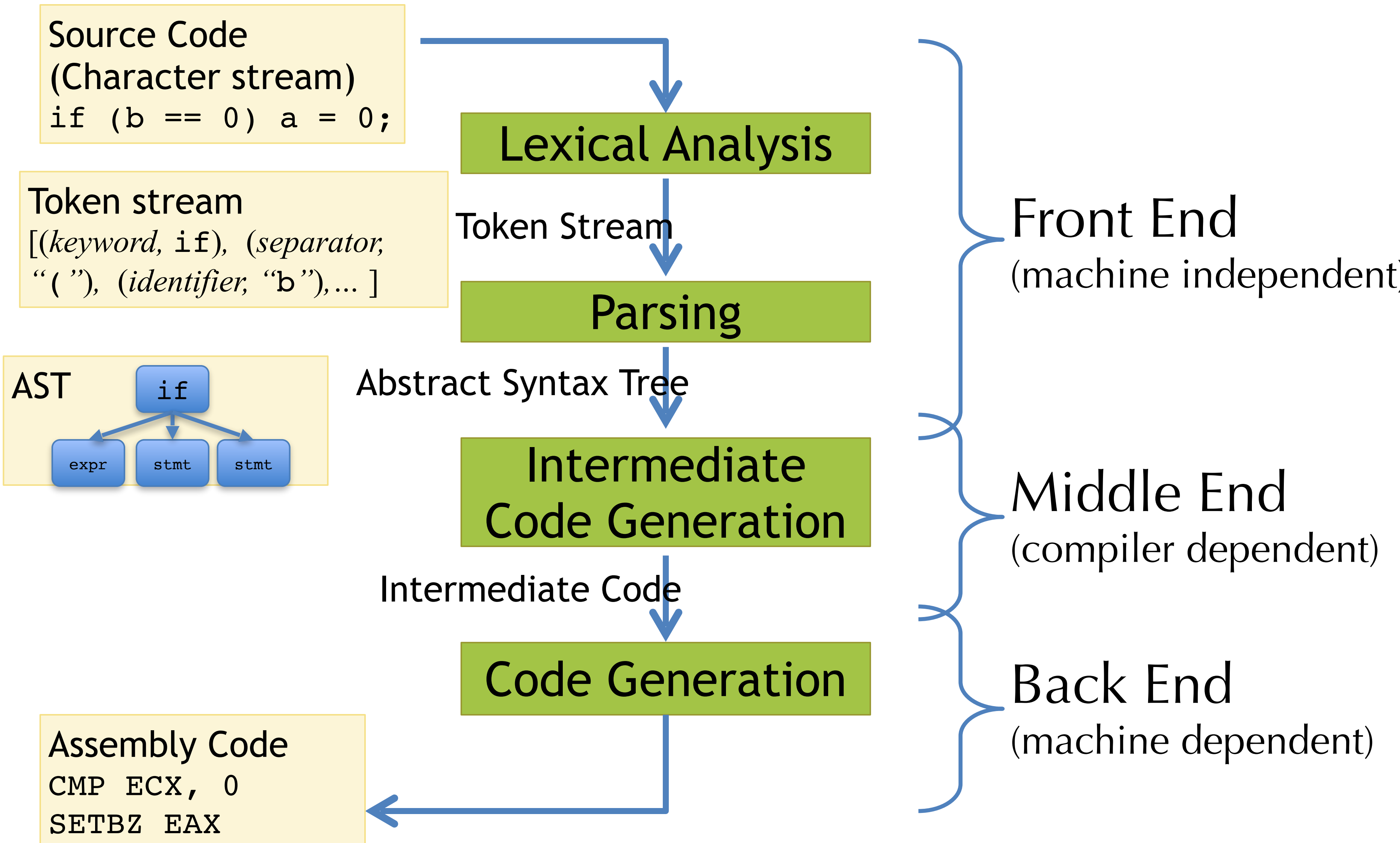
(Simplified) Compiler Structure



(Simplified) Compiler Structure



(Simplified) Compiler Structure



Typical Compiler Stages

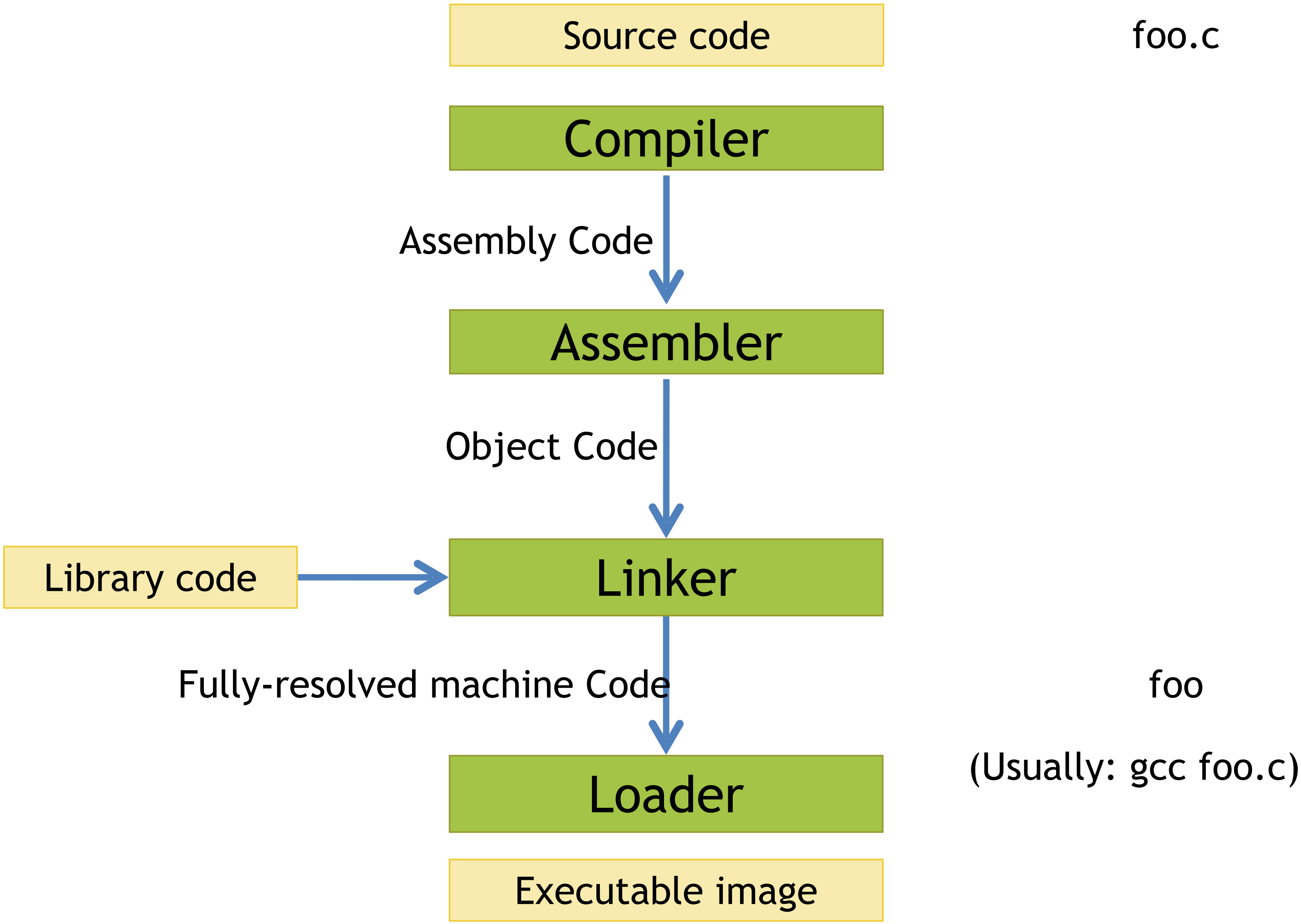
Lexing	→	token stream
Parsing	→	abstract syntax
Disambiguation	→	abstract syntax
Semantic analysis	→	annotated abstract syntax
Translation	→	intermediate code
Control-flow analysis	→	control-flow graph
Data-flow analysis	→	interference graph
Register allocation	→	assembly
Code emission		
Compiler passes		Representations of the program

Typical Compiler Stages

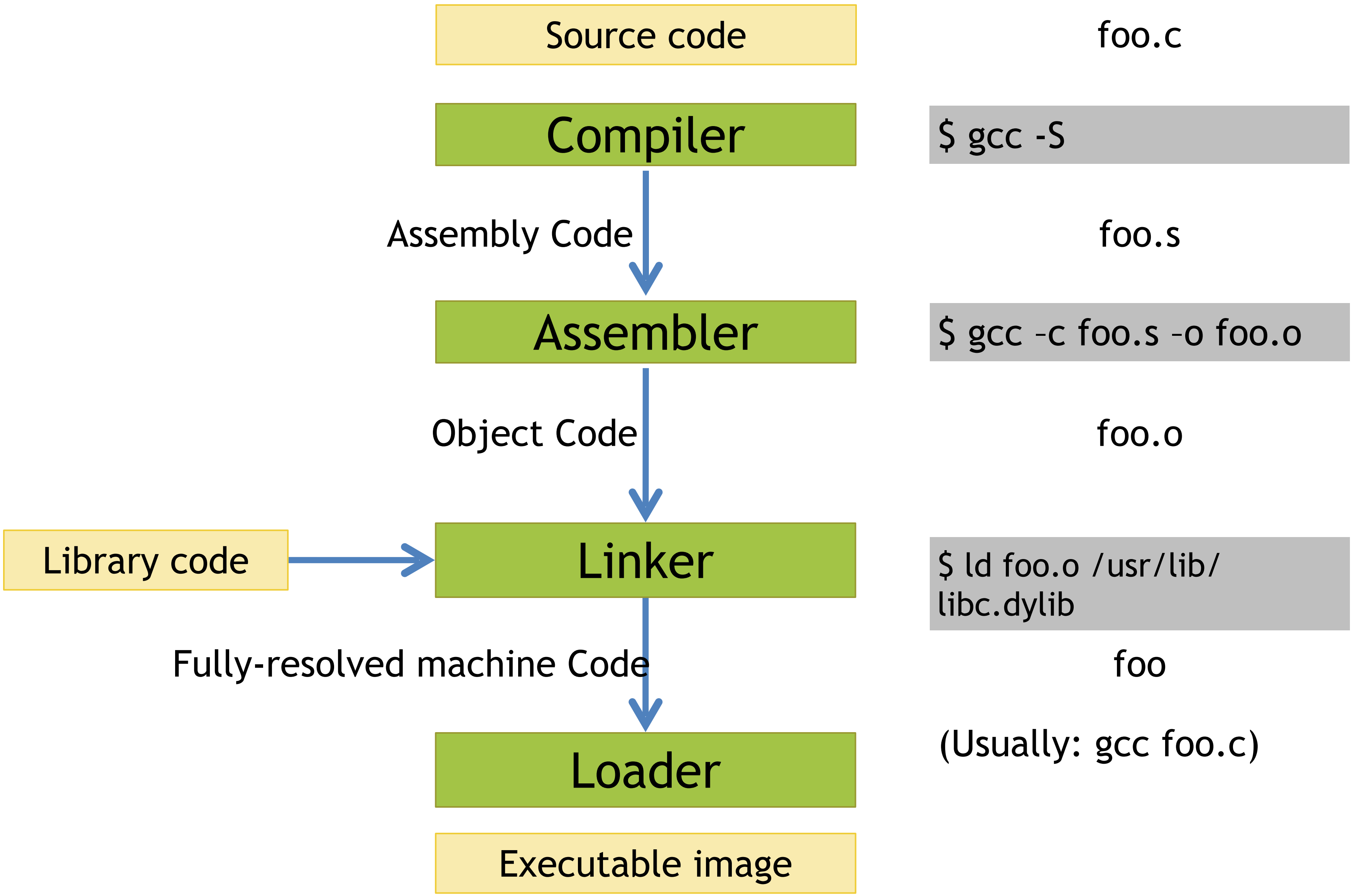
Lexing	→	token stream
Parsing	→	abstract syntax
Disambiguation	→	abstract syntax
Semantic analysis	→	annotated abstract syntax
Translation	→	intermediate code
Control-flow analysis	→	control-flow graph
Data-flow analysis	→	interference graph
Register allocation	→	assembly
Code emission		
Compiler passes		**Representations of the program**

- Optimizations may be done at many of these stages
- Different source language features may require more/different stages
- Assembly code is not the end of the story

Compilation & Execution



Compilation & Execution



GNU/C compiler demo

factorial.c

Short-Term Plan

- **Part I of Today:**
  - Crash course on Ocaml
- **Part II of Today**:
  - "object language" vs. "meta language"
  - Build a simple interpreter
  - Build a simple compiler
- **At home:**
  - Start Homework 1!

Introduction to OCaml programming

A little background about ML

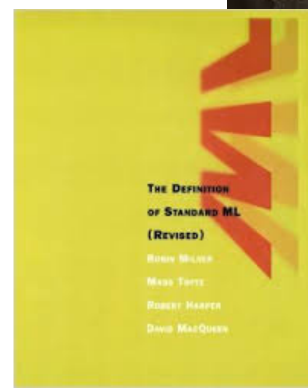
Interactive tour of OCaml via UTop & VSCode

Writing simple interpreters

OCAML

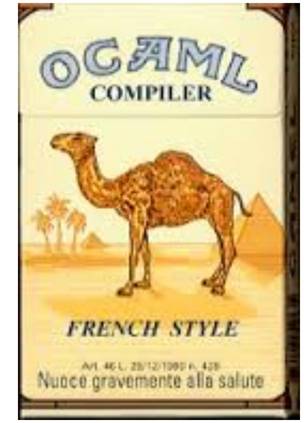
ML's History

- **1971: Robin Milner** starts the LCF Project at Stanford
  - "logic of computable functions"
- **1973:** At Edinburgh, Milner implemented his theorem prover and dubbed it "Meta Language" – ML
- **1984:** ML escaped into the wild and became "Standard ML"
  - SML '97 newest version of the standard
  - There is a whole family of SML compilers:
    - SML/NJ – developed at AT&T Bell Labs
    - MLton – whole program, optimizing compiler
    - Poly/ML
    - Moscow ML
    - ML Kit compiler
    - MLj – SML to Java bytecode compiler
- ML 2000: failed revised standardization
- sML: successor ML – discussed intermittently
- **2014:** sml-family.org + definition on github



OCaml's History

- The Formel project at the Institut National de Rechereche en Informatique et en Automatique (INRIA)
- **1987:** Guy Cousineau re-implemented a variant of ML
  - Implementation targeted the "Categorical Abstract Machine" (CAM)
  - As a pun, "CAM-ML" became "CAML"
- **1991:** Xavier Leroy and Damien Doligez wrote Caml-light
  - Compiled CAML to a virtual machine with simple bytecode (much faster!)
- **1996:** Xavier Leroy, Jérôme Vouillon, and Didier Rémy
  - Add an object system to create OCaml
  - Add native code compilation
- Many updates, extensions, since…
- Microsoft's F# language is a descendent of OCaml
- **2013:** ocaml.org



OCaml Tools

- ocaml – the top-level interactive loop
- ocamlc – the bytecode compiler
- ocamlopt – the native code compiler
- ocamldep – the dependency analyzer
- ocamldoc – the documentation generator
- ocamllex – the lexer generator
- ocamlyacc – the parser generator

- menhir – a more modern parser generator
- dune – a compilation manager
- ~~ocamlbuild – a compilation manager~~
- utop – a more fully-featured interactive top-level

- opam – package manager

Distinguishing Characteristics

- Functional & (Mostly) "Pure"
  - Programs manipulate values rather than issue commands
  - Functions are first-class entities
  - Results of computation can be "named" using `let`
  - Has relatively few "side effects" (imperative updates to memory)

- Strongly & Statically typed
  - Compiler typechecks every expression of the program, issues errors if it can't prove that the program is type safe
  - Good support for type inference & generic (polymorphic) types
  - Rich user-defined "algebraic data types" with pervasive use of *pattern matching*
  - Very strong and flexible module system for constructing large projects

Most Important Features for CS516

- Types:
  - int, bool, int32, int64, char, string, built-in lists, tuples, records, functions
- Concepts:
  - Pattern matching
  - Recursive functions over algebraic (i.e. tree-structured) datatypes
- Libraries:
  - Int32, Int64, List, Printf, Format

OCaml Demo

intro.ml