Lecture 1

CIS 341: COMPILERS

Administrivia

- **Instructor:** Steve Zdancewic
  Office hours: Mondays 4:00-5:00pm & by appointment.
  Levine 511

- **TAs:**
  - Lef Ioannidis (Mondays 12:30-1:30, loc: TBD)
  - Stephen Mell TBD
  - Sumanth Shivshankar (Tuesdays 4:30-5:30, loc: TBD)

- **E-mail:** cis341@seas.upenn.edu
- **Web site:** http://www.seas.upenn.edu/~cis341
- **Piazza:** http://piazza.com/upenn/spring2022/cis341

Announcements

- Dr. Zdancewic will be away next Tuesday
  - No class on Tuesday, January 18th

- HW1: Hellocaml
  - available on the course web site soon
  - due Wednesday, January 26th at 11:59pm

- Note: We're in the process of modernizing/updating the CIS341 course project infrastructure
  - switch from ocamlbuild to dune
  - use Gradescope for project autograding
  - deploy code via Github Classroom

Why CIS 341?

- 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 CIS341 Compiler

- Course projects
  - HW1: Hellocaml! (OCaml programming)
  - HW2: X86lite interpreter
  - HW3: LLVMlite compiler
  - HW4: Lexing, Parsing, simple compilation
  - HW5: Higher-level Features
  - HW6: Analysis and Optimizations

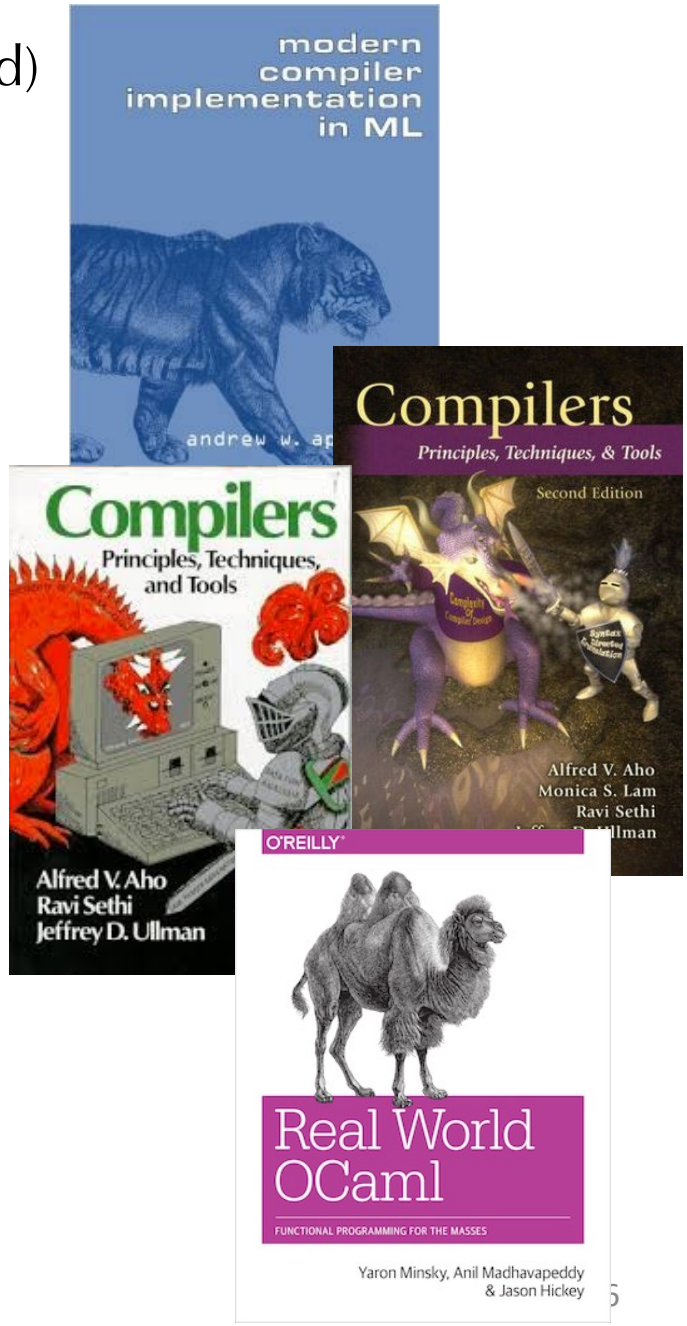
- Goal: build a complete compiler from a high-level, type-safe language to x86 assembly.

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)



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

- Forgot about OCaml after CIS120?
  - 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

- Homework 1 available soon on the course web site.
  - Individual project – no groups
  - Due: *Wednesday, 26 Jan. 2020 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 CIS341 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 Piazza to report any troubles with the toolchain!

Homework Policies

- Homework (except HW1) should 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
- 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 University's Code of Academic Integrity
  - "low level" and "high level" discussions across groups are fine
  - "mid level" discussions / code sharing are not permitted
  - General principle: *When in doubt, ask!*

Course Policies

Prerequisites: CIS121 and CIS240 (262 useful too!)

- 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:

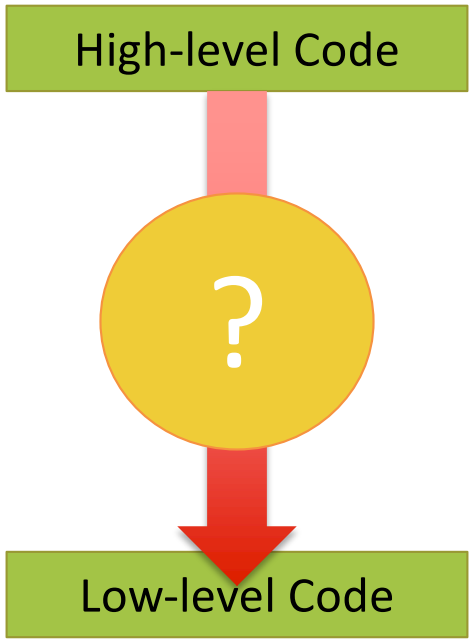
- 70% Projects: *Compiler*
  - Groups of 2 students
  - Implemented in OCaml
- 12% Midterm: in class … tentatively March 3rd
- 18% Final exam
- Lecture attendance is crucial
  - Active participation (asking questions, etc.) is encouraged
- *When in person, no laptops (or other devices)!*
  - It's too distracting for me and for others in the class.

What is a compiler?

COMPILERS

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: FORTRAN compiler built at IBM
  - Team led by John Backus
- 1960's: development of the first bootstrapping compiler for LISP
- 1970's: language/compiler design blossomed

1980s: ML / LCF
1984: Standard ML
1987: Caml
1991: Caml Light
1995: Caml Special Light
1996: Objective Caml
2005: F# (Microsoft)
2015: Reason ML
2020: OCaml Platform

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

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:

```
#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 x86 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?

- 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

- 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)

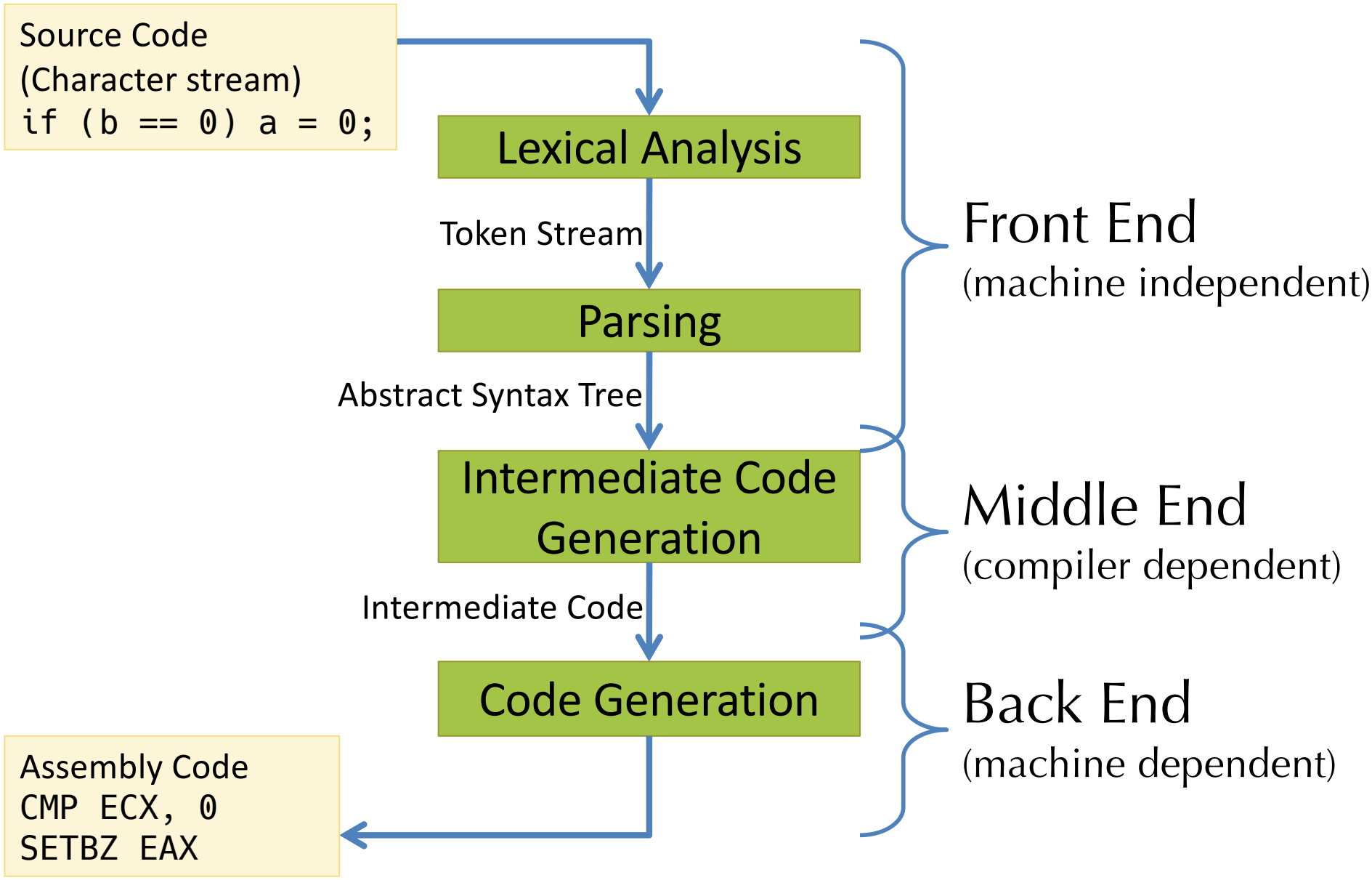
Idea: Translate in Steps

- Compile via a series of program representations

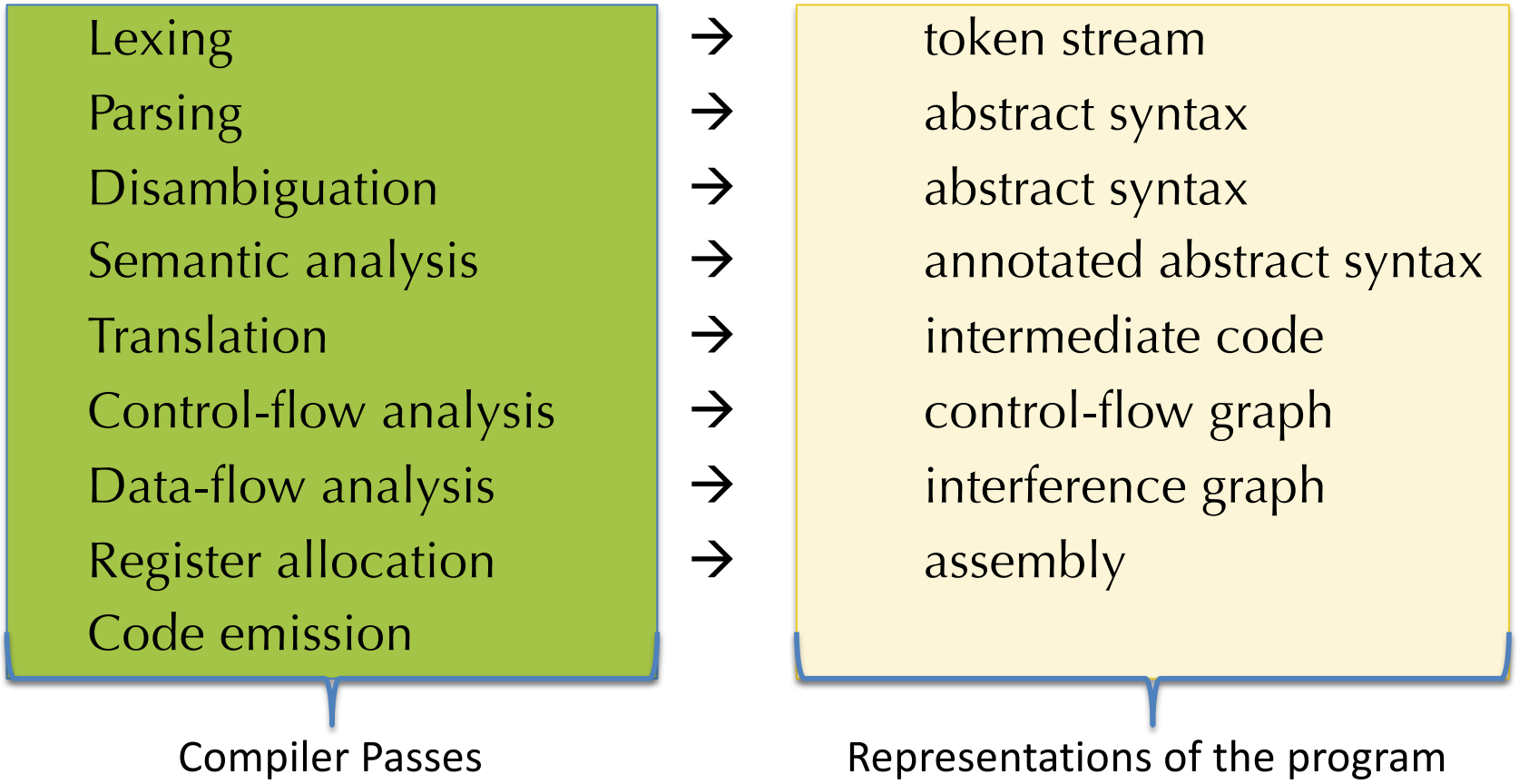
- Intermediate representations are optimized for program manipulation of various kinds:
  - Semantic analysis: type checking, error checking, etc.
  - Optimization: dead-code elimination, common subexpression elimination, function inlining, register allocation, etc.
  - Code generation: instruction selection

- Representations are more machine specific, less language specific as translation proceeds

(Simplified) Compiler Structure

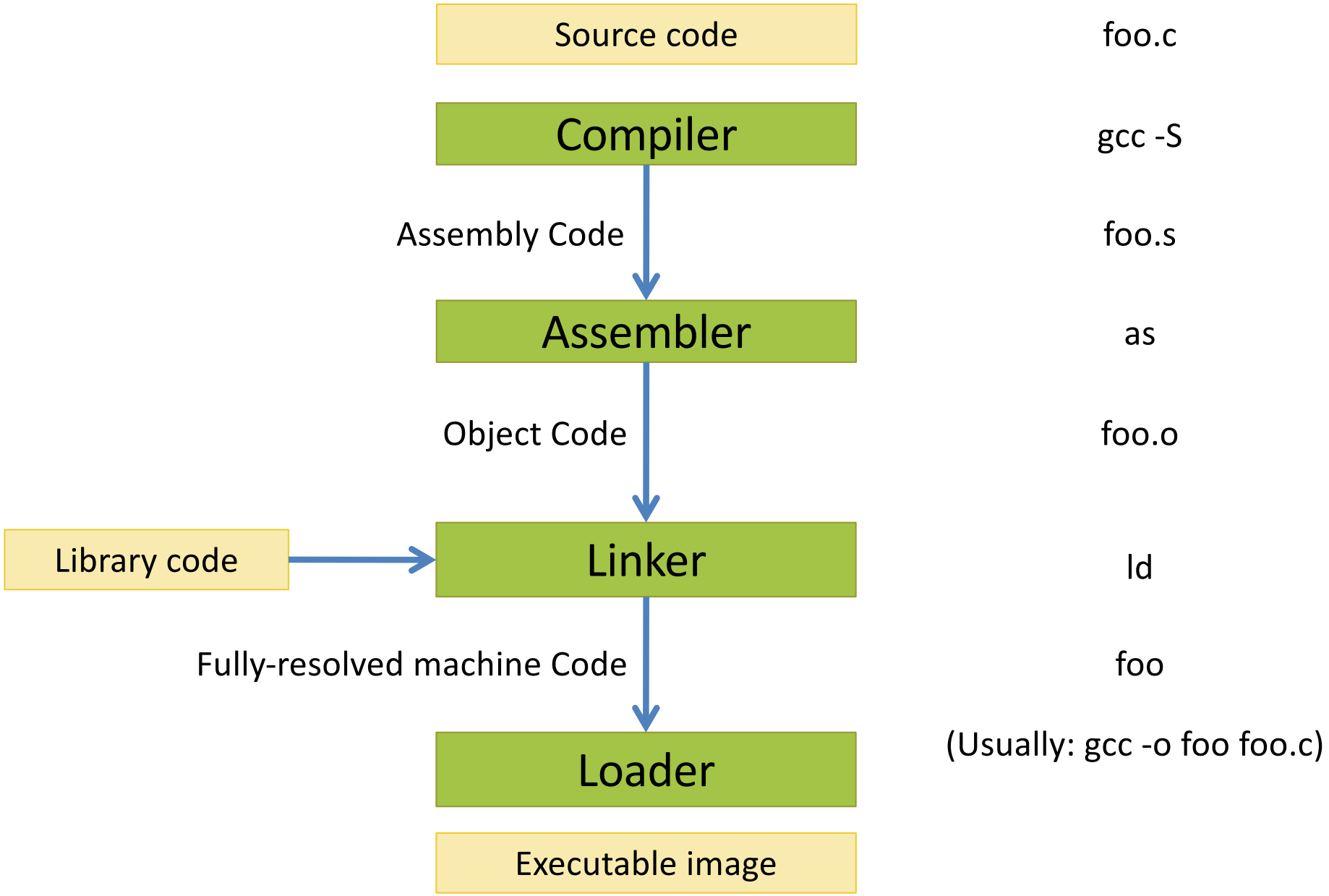


Typical Compiler Stages



- 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



See lec01.zip

COMPILER DEMO

Short-term Plan

- Rest of today:
  - Refresher / background on OCaml
  - "object language" vs. "meta language"
  - Build a simple interpreter

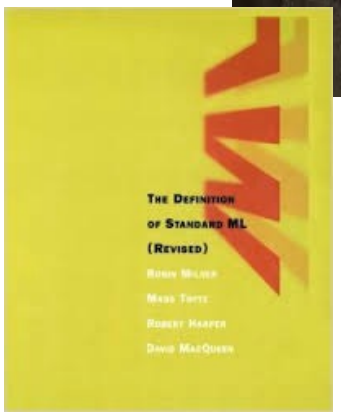
- Next week:
  - I will be out of town on Tuesday (attending POPL)

Introduction to OCaml programming

- A little background about ML
- Interactive tour of OCaml via UTop & VSCode
- Writing simple interpreters

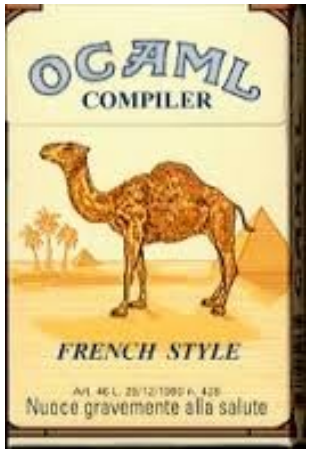
ML's History

- **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…
- **2005:** Microsoft's F# language is a descendent of OCaml
- **2013:** ocaml.org
- **2020:** OCaml Platform
- **2022:** Multicore OCaml



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 CIS341

- 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

How to represent programs as data structures.

How to write programs that process programs.

INTERPRETERS

Factorial: Everyone's Favorite Function

- Consider this implementation of factorial in a hypothetical programming language that we'll call "SIMPLE"
  (Simple IMperative Programming LanguagE):

```
X = 6;
ANS = 1;
whileNZ (x) {
    ANS = ANS * X;
    X = X + -1;
}
```

- We need to describe the constructs of this SIMPLE
  - ***Syntax***: which sequences of characters count as a legal "program"?
  - ***Semantics***: what is the meaning (behavior) of a legal "program"?

"Object" vs. "Meta" language

Object language:
the language (syntax / semantics) being described or manipulated

Metalanguage:
the language (syntax / semantics) used to *describe* some object language

Today's example:

SIMPLE	interpreter written in OCaml

Course project:

OAT ⇒ LLVM ⇒ x86_64	compiler written in OCaml

Clang compiler:

C/C++ ⇒ LLVM ⇒ x86_64	compiler written in C++

Metacircular interpreter:

lisp	interpreter written in lisp

Grammar for a Simple Language

```
<exp>  ::=
       |     <X>
       |     <exp> + <exp>
       |     <exp> * <exp>
       |     <exp> < <exp>
       |     <integer constant>
       |     ( <exp> )

<cmd>  ::=
       |     skip
       |     <X> = <exp>
       |     ifNZ <exp> { <cmd> } else { <cmd> }
       |     whileNZ <exp> { <cmd> }
       |     <cmd>; <cmd>
```

BNF grammars are themselves domain-specific metalanguages for describing the syntax of other languages…

- Concrete syntax (grammar) for a simple imperative language
  - Written in "Backus-Naur form"
  - <*exp*> and <*cmd*> are ***nonterminals***
  - '::=' , '|' , and <…> symbols are part of the *metalanguage*
  - keywords, like '`skip`' and '`ifNZ`' and symbols, like '`{`' and '`+`' are part of the *object language*

- Need to represent the ***abstract syntax*** (i.e. hide the irrelevant of the concrete syntax)
- Implement the ***operational semantics*** (i.e. define the behavior, or meaning, of the program)

OCaml Demo

simple.ml