Lecture 1

CIS 4521/5521: COMPILERS

Administrivia

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

- **TAs:**
  - Gary Chen Office Hours: TBD
  - Noé De Santo Office Hours: TBD
  - Alex Shapulya Office Hours: TBD

- **E-mail:** cis5521@seas.upenn.edu
- **Web site:** http://www.seas.upenn.edu/~cis5521

Look for announcements on ED

Why CIS 4521 / 5521?

- 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…

CIS 4521/5521 vs. CIS 341(0)

CIS 341 / 3410 is the old version of this class; the content is identical.

Masters and PhD students should register for CIS 5521.

If you want to change, send mail to cis5521@seas.upenn.edu.

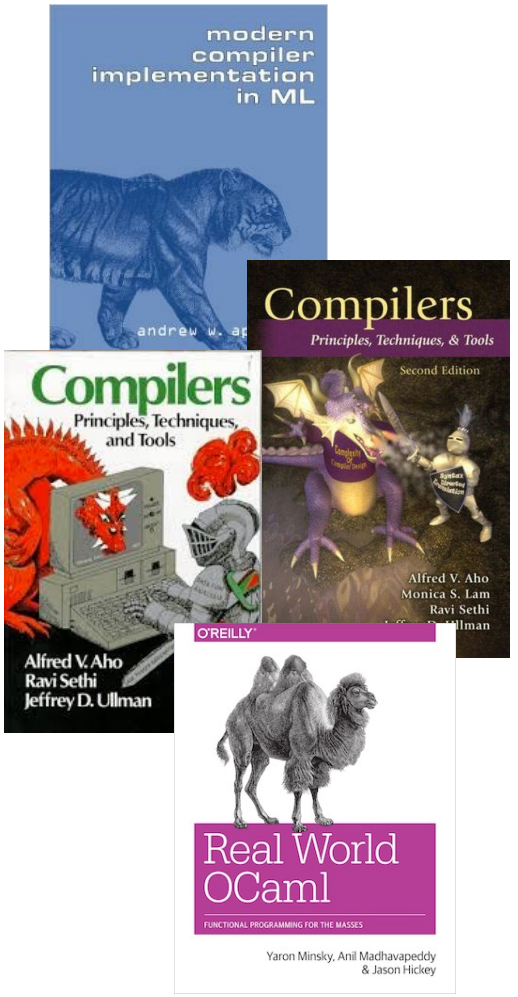
The CIS4521/5521 Compiler

- 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 data types, 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 CIS1200 or never used it?
  - 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 on the course web site.
  - Individual project – no groups
  - **Due:** ***Wednesday, 29 Jan. 2024 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 + Docker
  - the projects will build a "dev container" for you
  - See the course web pages about the CIS4521 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 Ed 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:

- Significant programming experience, familiarity with trees, graphs, low-level coding (CIS 1210/5490/5730 and CIS 2400/5010)
- Some familiarity with computation models (automata, stack machines, etc.), is useful too! (CIS 2620/5110)
- If HW01 is a struggle, this class might not be a good fit for you (HW01 is *significantly* simpler than the rest…)

Grading:

- 60% Projects: *Compiler*
  - Groups of 2 students (except for HW01)
  - Implemented in OCaml
- 20% Midterm: in class … tentatively March 6th
- 20% Final exam
- Lecture attendance is crucial
  - Active participation (asking questions, etc.) is encouraged
- *When in person, I'd prefer no laptops/devices!*
  - It's too distracting for me and for others in the class.

Announcements

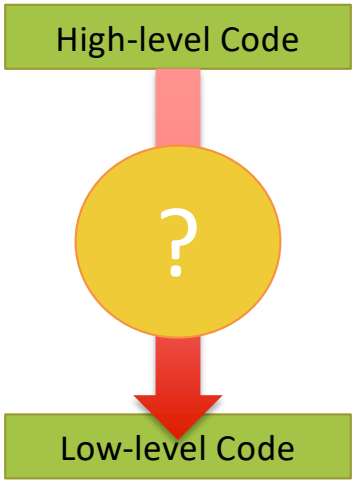
- Dr. Zdancewic will be away next week
- Class lectures (OCaml demo / implementing interpreters) will be covered by TAs Noé and Alex.

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

- 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)
2015: Reason ML
2020: OCaml Platform
2022: Multicore OCaml

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) 64-bit x86 assembly code for the factorial function

```
_factorial:
## %bb.0:
    pushq%rbp
    movq %rsp, %rbp
    movl %edi, -4(%rbp)
    movl $1, -8(%rbp)
LBB0_1:
    cmpl $0, -4(%rbp)
    jle  LBB0_3
## %bb.2:
    movl -8(%rbp), %eax
    imull-4(%rbp), %eax
    movl %eax, -8(%rbp)
    movl -4(%rbp), %eax
    subl $1, %eax
    movl %eax, -4(%rbp)
    jmp  LBB0_1
LBB0_3:
    movl -8(%rbp), %eax
    popq %rbp
    retq
```

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!

```
00000000  cf fa ed fe 07 00 00 01  [illegible] 00 00 01 00 00 00  |................|
00000010  04 00 00 00 60 01 00 00  [illegible] 00 00 00 00 00 00  |....`.... ......|
00000020  19 00 00 00 e8 00 00 00  [illegible] 00 00 00 00 00 00  |................|
00000030  00 00 00 00 00 00 00 00  [illegible] 00 00 00 00 00 00  |................|
00000040  74 00 00 00 00 00 00 00  [illegible] 00 00 00 00 00 00  |t...............|
00000050  74 00 00 00 00 00 00 00  [illegible] 00 00 00 07 00 00  |t...............|
00000060  02 00 00 00 00 00 00 00  [illegible] 74 65 78 74 00 00  |........__text..|
00000070  00 00 00 00 00 00 00 00  [illegible] 54 45 58 58 54 00  |........__TEXT..|
00000080  00 00 00 00 00 00 00 00  [illegible] 00 00 00 00 00 00  |................|
00000090  61 00 00 00 00 00 00 00  [illegible] 00 00 04 00 00 00  |a...............|
000000a0  f8 01 00 00 03 00 00 00  [illegible] 00 80 00 00 00 00  |................|
000000b0  00 00 00 00 00 00 00 00  [illegible] 63 73 74 72 69 6e  |........__cstrin|
000000c0  67 00 00 00 00 00 00 00  [illegible] 54 45 58 58 54 00  |g.......__TEXT..|
000000d0  00 00 00 00 00 00 00 00  [illegible] 00 00 00 00 00 00  |........a.......|
000000e0  13 00 00 00 00 00 00 00  [illegible] 00 00 00 00 00 00  |................|
000000f0  00 00 00 00 00 00 00 00  [illegible] 00 00 00 00 00 00  |................|
00000100  00 00 00 00 00 00 00 00  24 00 00 00 10 00 00 00  |........$.......|
00000110  00 0c 0a 00 00 02 0e 00  02 00 00 00 18 00 00 00  |................|
00000120  10 02 00 00 03 00 00 00  40 02 00 00 20 00 00 00  |........@... ...|
00000130  0b 00 00 00 50 00 00 00  00 00 00 00 00 00 00 00  |....P...........|
00000140  00 00 00 00 02 00 00 00  02 00 00 00 01 00 00 00  |................|
00000150  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
*
00000180  55 48 89 e5 89 7d fc c7  45 f8 01 00 00 00 83 7d  |UH...}..E......}|
00000190  fc 00 7e 15 8b 45 f8 0f  af 45 fc 89 45 f8 8b 45  |..~..E...E..E..E|
000001a0  fc 83 e8 01 89 45 fc eb  e5 8b 45 f8 5d c3 66 90  |.....E....E.].f.|
000001b0  55 48 89 e5 48 83 ec 10  89 7d fc 48 89 75 f0 bf  |UH..H....}.H.u..|
000001c0  06 00 00 00 e8 00 00 00  00 89 c6 48 8d 3d 0f 00  |...........H.=..|
000001d0  00 00 b0 00 e8 00 00 00  00 31 c0 48 83 c4 10 5d  |.........1.H...]|
000001e0  c3 66 61 63 74 6f 72 69  61 6c 28 36 29 20 3d 20  |.factorial(6) = |
000001f0  25 64 0a 00 00 00 00 00  55 00 00 00 02 00 00 2d  |%d......U......-|
00000200  4e 00 00 00 02 00 00 15  45 00 00 00 00 00 00 2d  |N.......E......-|
00000210  07 00 00 00 0f 01 00 00  00 00 00 00 00 00 00 00  |................|
00000220  01 00 00 00 0f 01 00 00  30 00 00 00 00 00 00 00  |........0.......|
00000230  12 00 00 00 01 00 00 00  00 00 00 00 00 00 00 00  |................|
00000240  00 5f 6d 61 69 6e 00 5f  66 61 63 74 6f 72 69 61  |._main._factoria|
00000250  6c 00 5f 70 72 69 6e 74  66 00 00 00 00 00 00 00  |l._printf.......|
```

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)

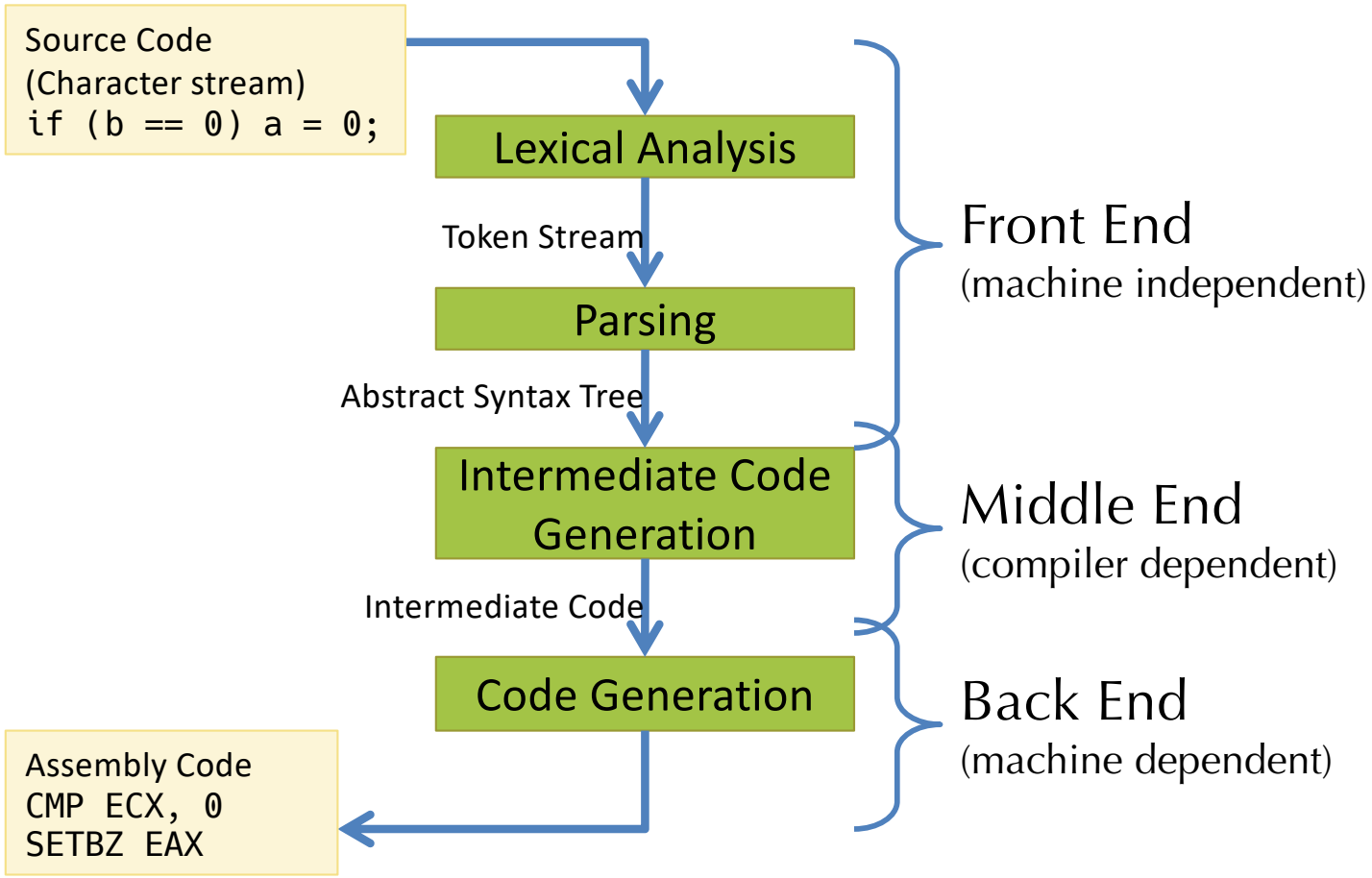
Idea: Translate in Steps

- Compile via a series of program representations

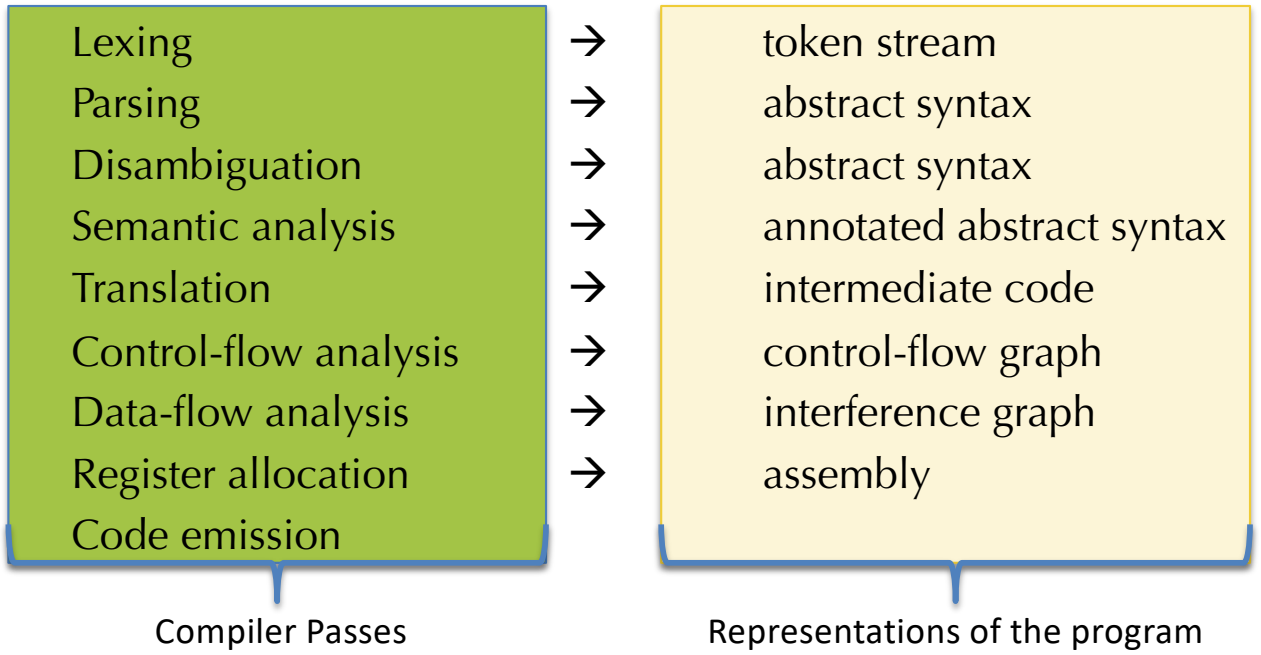
- Intermediate representations are optimized for program manipulation of various kinds:
  - Semantic analysis: type checking, error checking, etc.
  - Optimization: dead-code elimination, common subexpression elimination, function inlining, register allocation, etc.
  - Code generation: instruction selection

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

(Simplified) Compiler Structure

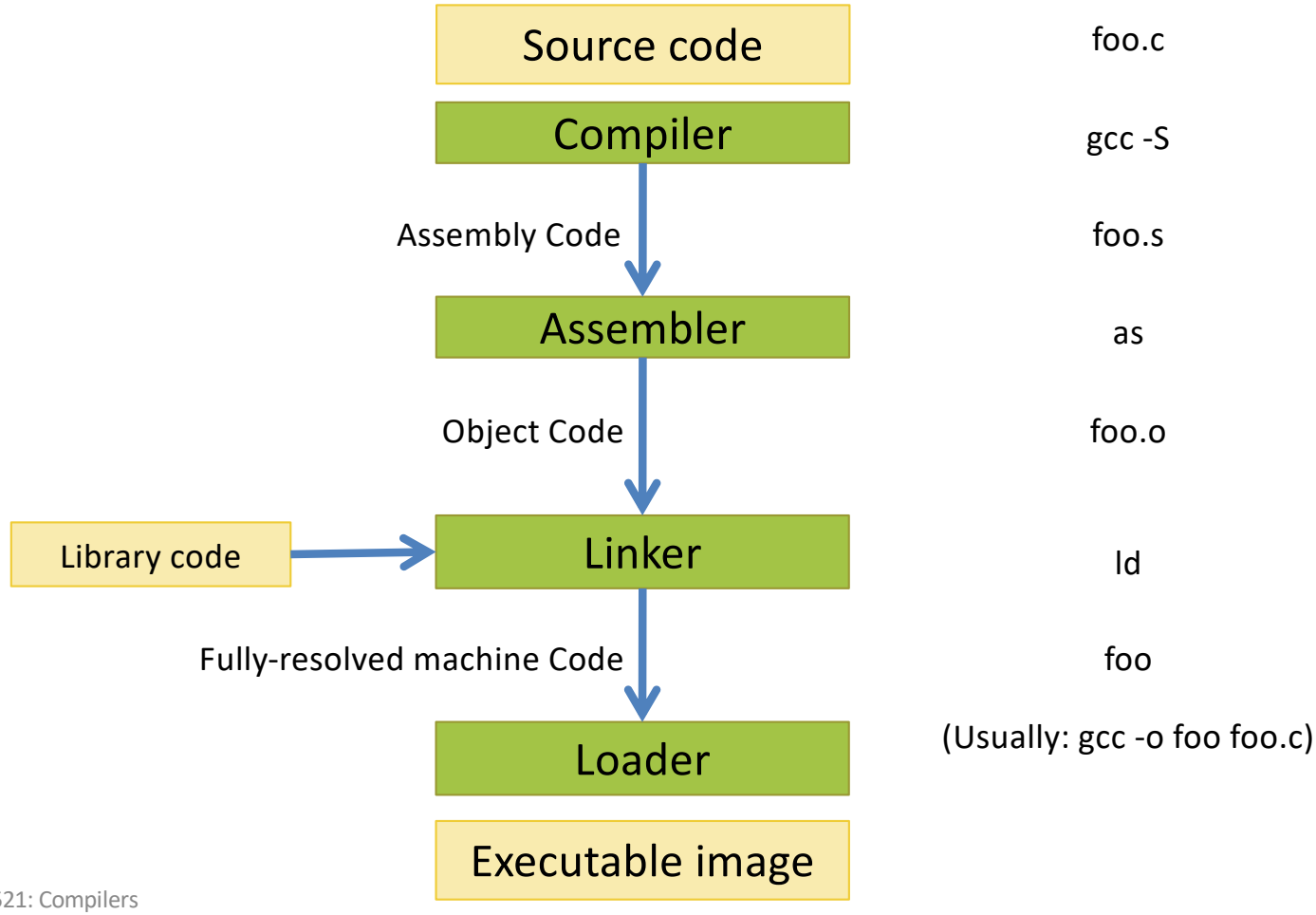


Typical Compiler Stages



- 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

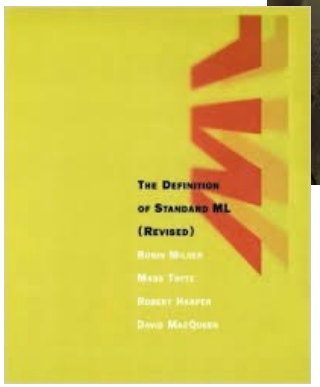
Introduction to OCaml programming

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

OCAML

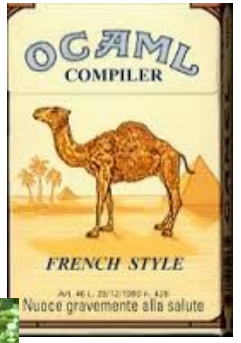
ML's History

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



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
- 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 CIS4521

- 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 language
  - ***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