HARVARD
John A. Paulson
School of Engineering
and Applied Sciences

CS153: Compilers
Lecture 1: Introduction

Stephen Chong

https://www.seas.harvard.edu/courses/cs153

Contains content from lecture notes by Steve Zdancewic, Greg Morrisett, and John Tristan

What is this course about?

Source Code

Expressive,
high-level/abstract

Compiler!

Low-level,
hard to read,
not much ambiguity
or redundancy

Target Code

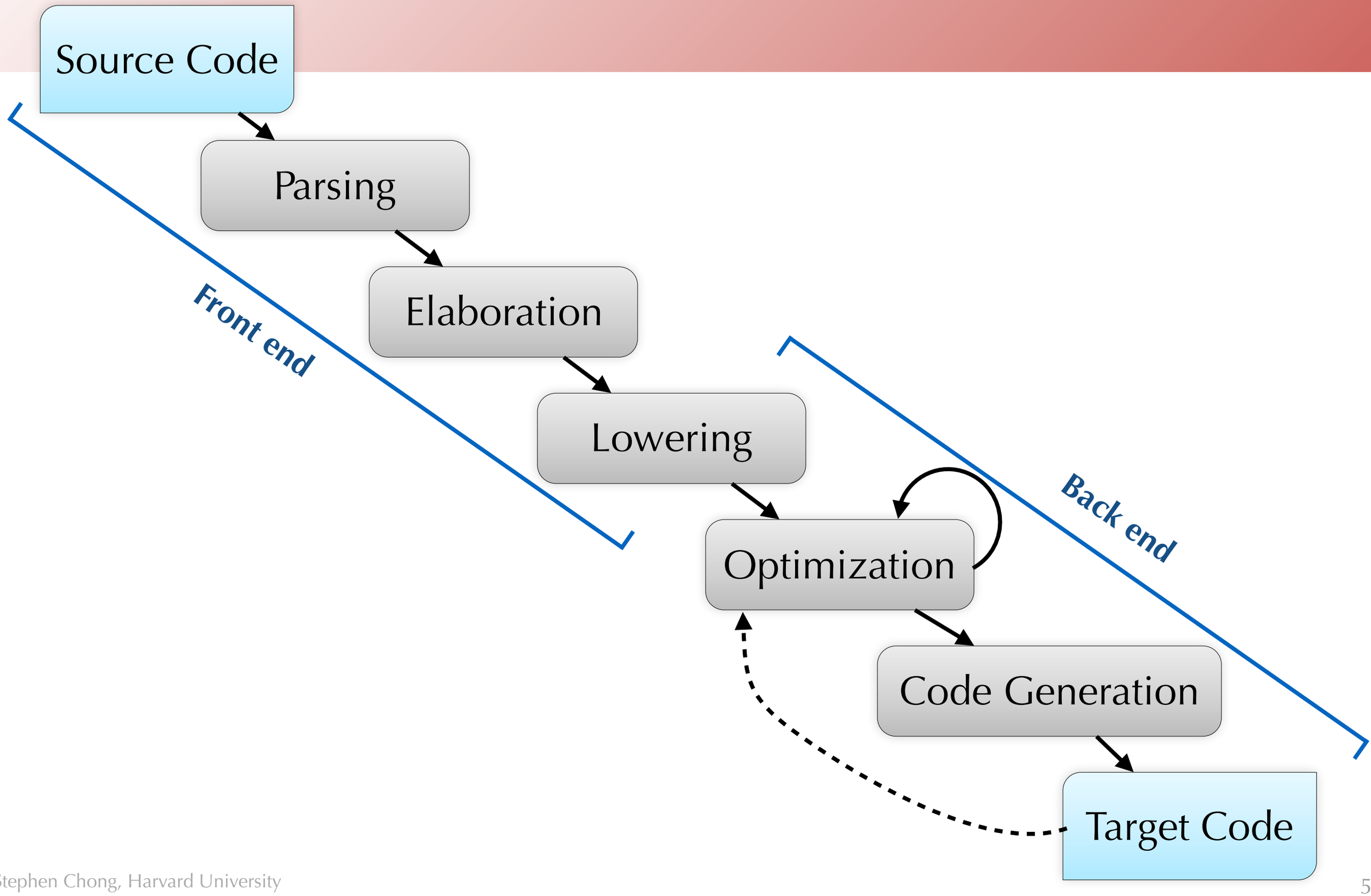
What is this course about?

- How are programs written in a high-level language transformed into machine code?
- How can we ensure a program is somewhat meaningful?
- How is the program's memory managed?
- How can we analyze programs to discover invariant properties and improve their runtime performance?

Historical Note

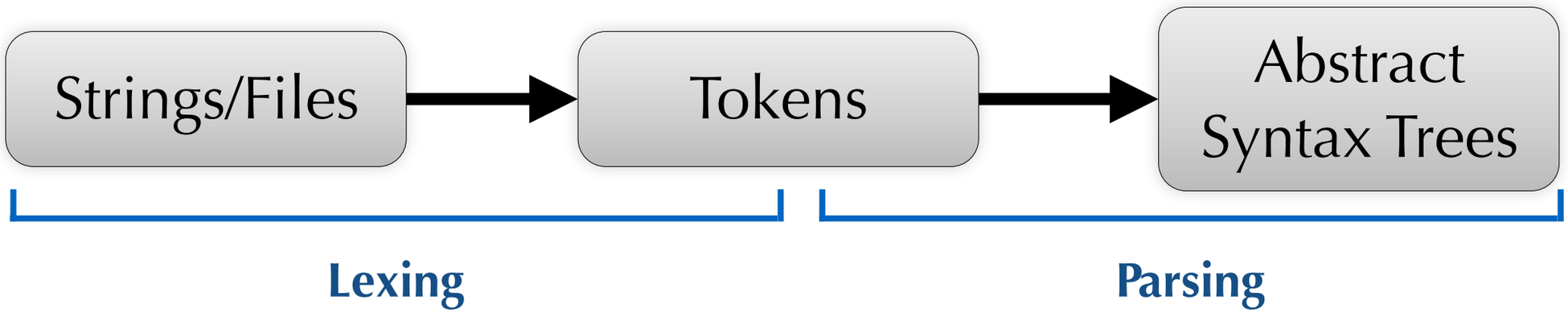
- Until the 1950's: computers were programmed in assembly.
- 1951–1952: Grace Hopper developed the A-0 system for the UNIVAC I
  - She later contributed significantly to the design of COBOL
- 1957: the FORTRAN compiler was built at IBM
  - Team led by John Backus
- 1960's: development of the first bootstrapping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

Basic Architecture



Front end

- Lexing & Parsing
  - From strings to data structures
  - Usually split into two phases:

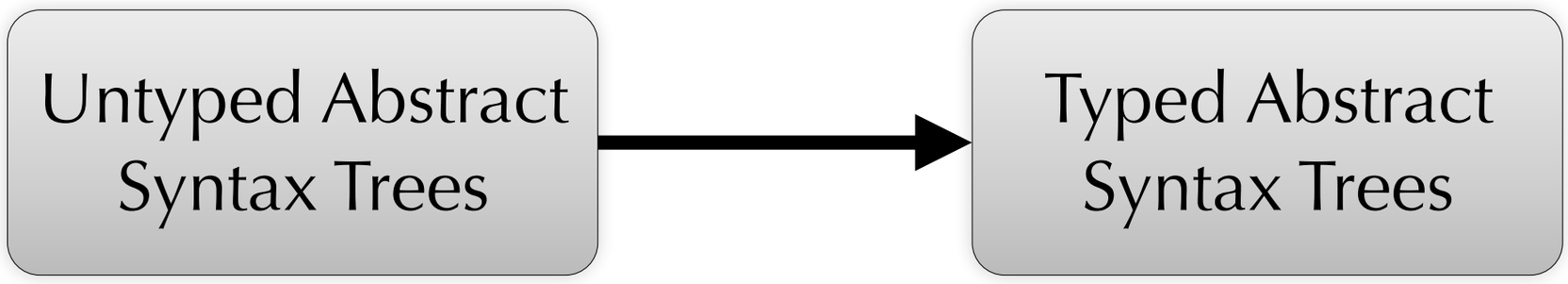


Parsing tools

- Parsing is something that happens in essentially all applications.
  - E.g., Google search bar, calendar, etc.
  - First step in going from raw data to information
- Lots of CS theory (121) to help out
  - Regular expressions (finite state automata)
  - Context-free grammars (push-down automata)
  - These abstractions show up in optimization too!
- Lots of tool support
  - E.g., Lex and Yacc, Menhir, Antlr, parsing combinators, etc.

Elaboration

- Type-checking
  - Resolve variables, modules, etc.
  - Check that operations are given values of the right types
  - Infer types for sub-expressions
  - Check for other safety/security problems

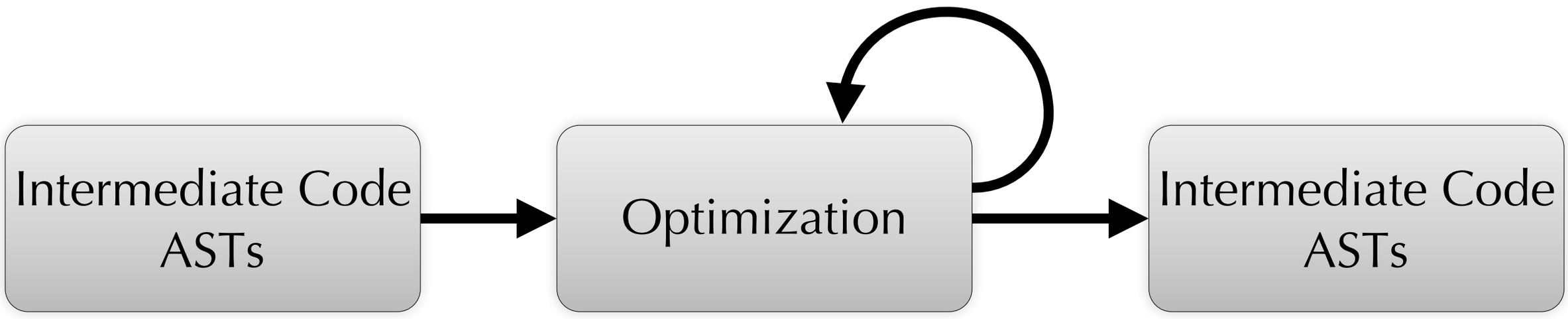


Lowering

- Translate high-level features into a small number of target-like constructs
  - e.g., `while`, `for`, `do`-loops all compiled to code using `goto`'s
  - e.g., objects, closures to records and function pointers
  - e.g., make type-tests, array-bounds checks, etc. explicit

Optimization

- Rewrite expensive sequences of operations into less expensive
  - e.g., constant folding: 3+4 → 7
  - e.g., lift an invariant computation out of a loop
  - e.g., parallelize a loop



Code generation

- Translate intermediate code into target code
  - Register assignment
  - Instruction selection
  - Instruction scheduling
  - Machine-specific optimization

Who should take CS153?

- People fascinated by languages & compilers
  - Why does[n't] this language include this feature?
  - Systems programmers
- Know the one tool that you use every day.
  - What can['t] a compiler do well?
- Architecture designers
  - Interdependence between compiler and architecture
  - See Intel iAPX 432 and Intel Itanium (compiler-related) failures; register windows
  - These days, architecture folks use compilers too!
- API designers
  - A language is the ultimate API
  - c.f., Facebook's Hack language

Learning outcomes

- 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, program analysis, & types
  - How to manipulate complex data structures
  - How to be a better programmer

Suggested prerequisites

- Ideally CS51 **and** CS61
  - CS51 alone is likely enough
- We assume
  - Knowledge of OCaml
    - Homework 1 will refresh you on OCaml/help you get up to speed
  - A bit about how machines work
    - e.g., 2's complement arithmetic
- First two projects are a good time to get up to speed
- If you don't know something, ask!

Course Staff

Nenya Edjah

Michael Horton

Teddy Liu

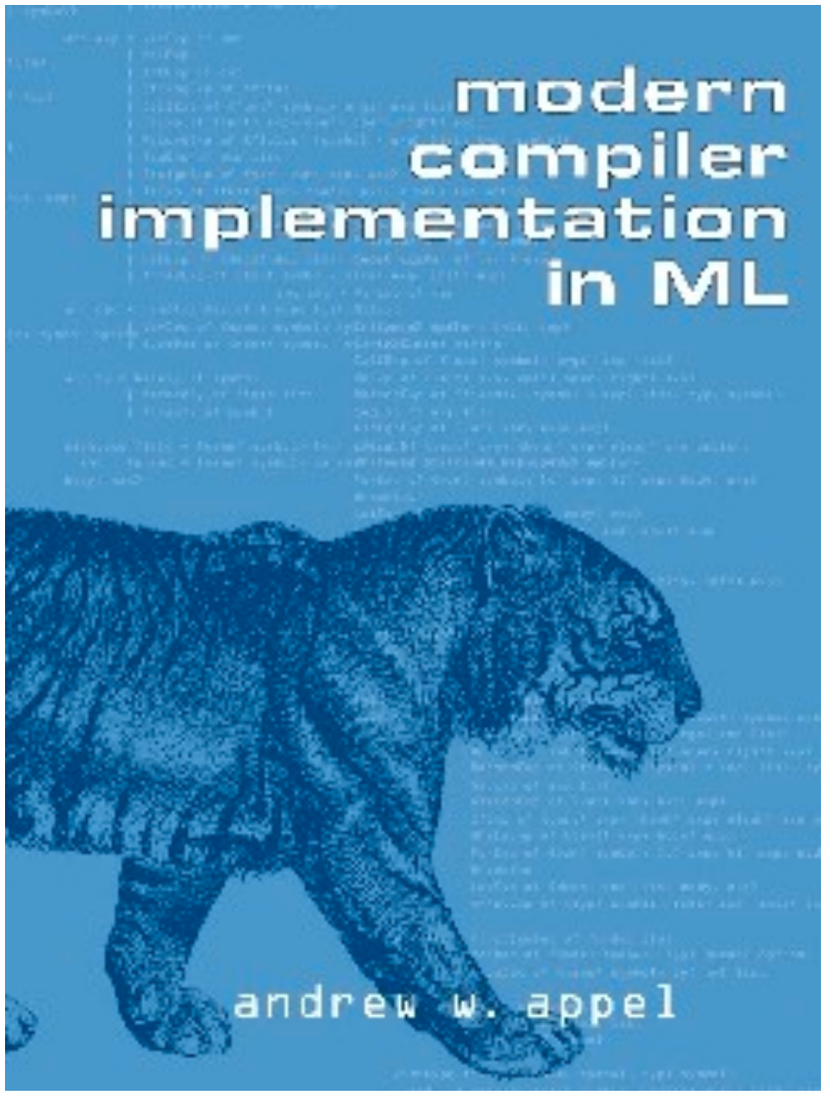
- Contact course staff at
  cs153-staff@seas.harvard.edu
  or via Piazza
- Don't send email to individual course staff
  (slower response, all of us need to be informed)

Administrivia

- Website
  - https://www.seas.harvard.edu/courses/cs153
- Piazza
  - https://piazza.com/class#fall2019/cs153
    - In general, post questions publicly unless you think question or answer might be inappropriate sharing of project solutions
    - Can post anonymously to classmates if you wish
- Office hours
  - Will be announced later. Start next week
  - Look on course website
- No section
  - TF hours will be spent on office hours instead

Textbook

- *Modern Compiler Implementaton in ML* by Andrew W. Appel
  - Recommended but not required.
- In most cases, class materials should suffice
- Lecture slides posted after the lecture
  (or before if they are ready in time)



Embedded EthiCS Module

- Ethical reasoning is essential skill
- Teach ethical reasoning by integration into many courses
  - Collaboration between Harvard CS and Philosophy Department
- Teach students to:
  - Identify and anticipate ethical and social issues in their work.
  - Think clearly about those issues, both alone and collaboratively.
  - Communicate their understanding of those issues effectively.
  - Design systems that take into account ethical and social concerns.
- One lecture and assignment.
  - All students are expected to attend.
  - Date to be announced soon
- More info on Embedded EthiCS at https://embeddedethics.seas.harvard.edu/

Programming environment

- OCaml
  - Ideal for writing compilers!
    - Designed to enable easy manipulation of *abstract syntax trees*
    - Type-safe, mostly pure, functional language with support for polymorphic (generic) algebraic datatypes, modules, and mutable state
  - HW1 will refresh you on OCaml
  - See https://www.seas.harvard.edu/courses/cs153/2019fa/resources.html for some links to resources, tools, etc.
- …

Assessment

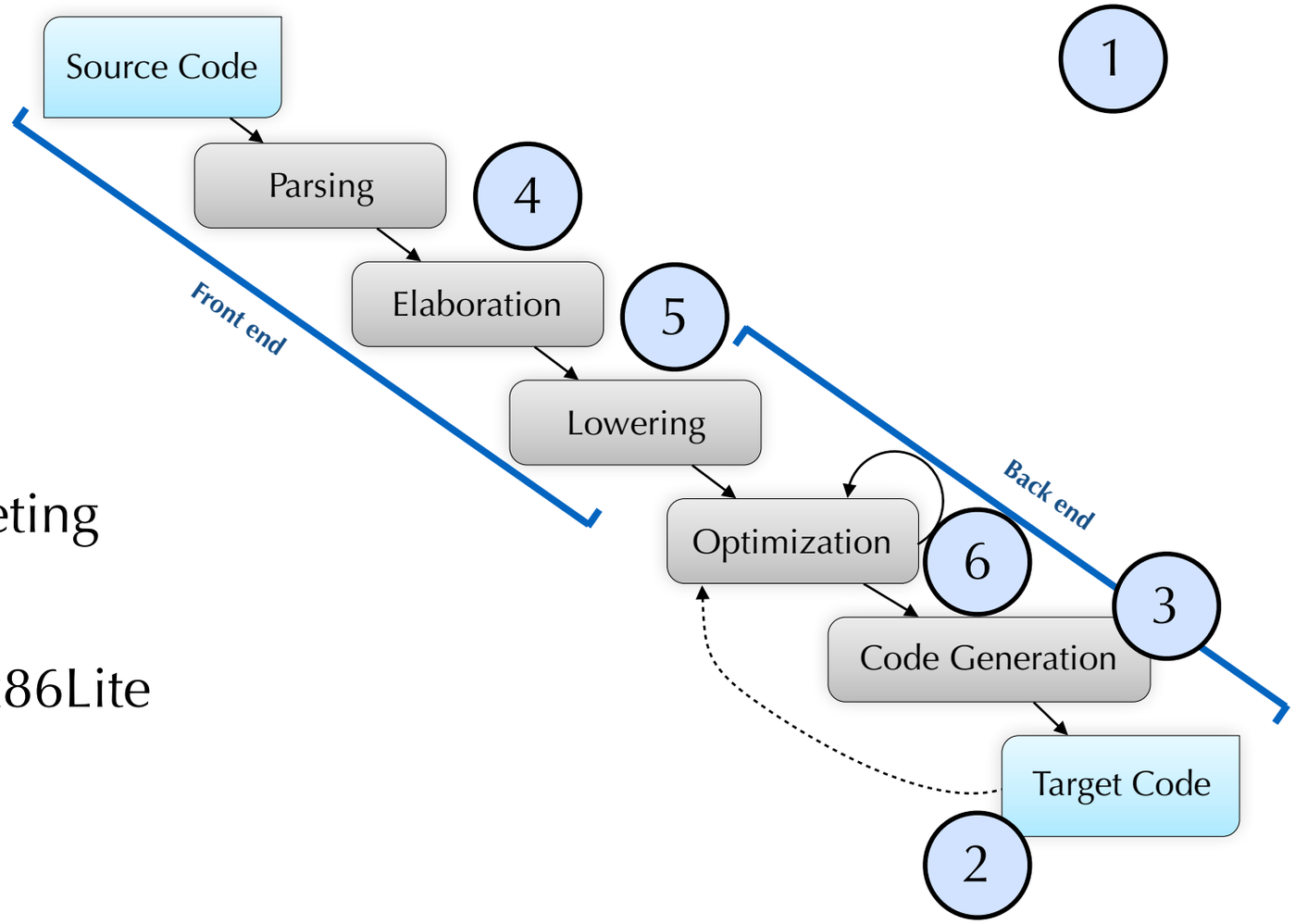
- Roughly:
  - 75% projects (~6 projects total, not equally weighted)
  - 20% final exam
  - 5% participation
    - E.g., Piazza posting/answering, attend lectures and engage in discussion, attend office hours, ...

Projects

- 6 projects, mostly implementing parts of a compiler
- Typically 2-3 weeks per project
- In OCaml
- Implementation heavy course!
  - All submissions must type-check and compile
  - Multiple projects out at same time
- Late minutes
  - 14,400 late minutes = 10 days
  - At most 4,320 (=3 days) can be used on any one assignment
  - Plan ahead! Late minutes are to help you manage your time effectively, not a substitute for starting projects early
  - Note: Late minutes are not meant to be used for health issues (including mental health issues), family emergencies or other extenuating circumstances. Contact Prof Chong (or have your resident tutor do so).

Projects

- 1. HellOCaml
  - Refresh OCaml coding
  - Implement interpreter
- 2. x86Lite
  - x86 Assembler and Simulator
  - Understand the machine we are targeting
- 3. LLVMlite Compilation
  - Compile simbple subset of LLVM to x86Lite
- 4. OAT v1
  - Parsing and lexing
  - Simple front-end
- 5. OAT v2
  - Structs, function pointers,
  - Simple lowering & code generation
- 6. Dataflow analysis and optimizations



Collaboration/Academic integrity

- Projects are done individually: you must write all your own code.
  - Don't share your code with others
  - Don't accept code from others
  - Don't look for solutions online
  - Don't post course materials to websites or course-content archives
    - Make sure your GitHub repo is private!
- **But:** study groups are a very effective way to learn from your peers
- Guidelines for working with others:
  - OK to talk about high-level ideas: understanding concepts, how to approach an implementation
    - Use words, not code
  - OK to talk about low-level details
    - How to use an OCaml library, etc.
- **If you are ever in doubt, ask the course staff to clarify what is and isn't appropriate.**

Diversity and Inclusion

- Aim: create a learning environment that supports a diversity of thoughts, perspectives and experiences, and honors your identities (including race, gender, class, sexuality, religion, ability, etc.)

Diversity and Inclusion

- To help accomplish this:
  - If you have a name and/or set of pronouns that differ from those that appear in your official Harvard records, please let me know!
  - If you feel like your performance in the class is being impacted by your experiences outside of class, please don't hesitate to come and talk with me. I want to be a resource for you. If you prefer to speak with someone outside of the course, members of the SEAS Committee on Diversity, Inclusion, and Belonging are excellent resources.
  - I (like many people) am still in the process of learning about diverse perspectives and identities. If something was said (by anyone) in class, office hours, Piazza, or project group work that made you feel uncomfortable, please talk to me about it.
  - As a participant in course discussions, office hours, and group projects, you should also strive to honor the diversity of your classmates.
- If you ever are struggling and just need someone to talk to, feel free to stop by office hours, or to reach out to me and we can arrange a private meeting.

Accessibility

- We all learn differently.
- If aspects of this course prevent you from learning or exclude you, please let me know ASAP
- Some resources
  - Accessible Education Office
    - https://aeo.fas.harvard.edu/
    - Accommodation letters: please get them to me soon, say end of 2nd week
  - Academic Resource Center
    - https://academicresourcecenter.harvard.edu/
    - Academic counseling, peer tutors, …

Extension School

- Offered through Extension School as CSCI E-153
- Same lectures, homeworks
- See https://www.seas.harvard.edu/courses/cs153/2019fa/extension.html for specific policies related to Extension School
- Will offer remote office hours
  - Details to be determined

Course Expectations

- You will do well! 😀
- Stay up to date on material and homeworks
  - Course moves fast!
- Look after yourself
  - Sleep, eat, keep well
- Attendance
  - Do not have to attend lecture
    - Lecture videos and notes will be available shortly after lecture
  - But figure out what learning method works best for you
    - Attendance often helpful

Course Expectations

- Devices in class
  - Opt-in device-free zone in lecture
  - Use device only to improve your learning
- Preparation for lectures
  - No prep required
  - But you may find it useful to look at lecture notes before lecture
  - I will try to release lecture notes the night before
- Office hours
  - No prep required for office hours of Prof Chong or TFs
  - Can ask about any aspect of the course or beyond
  - OK to come to office hours and work on your project
    - See collaboration policy
  - Encourage you to talk with other students! Introduce yourself if you don't know them
  - If you want to attend OH but can't, let us know

Course Expectations

- Course participation
  - Engage in the material!
  - Piazza, OH, lecture, extra credit, ...
  - I would like to see every student at least once in my office hours 😃
- Projects and extensions
  - Expect you to manage your time
  - Start early
    - Late minutes are for flexible scheduling
  - Extenuating circumstances: contact Prof Chong as soon as you can
- Collaboration
  - Already discussed

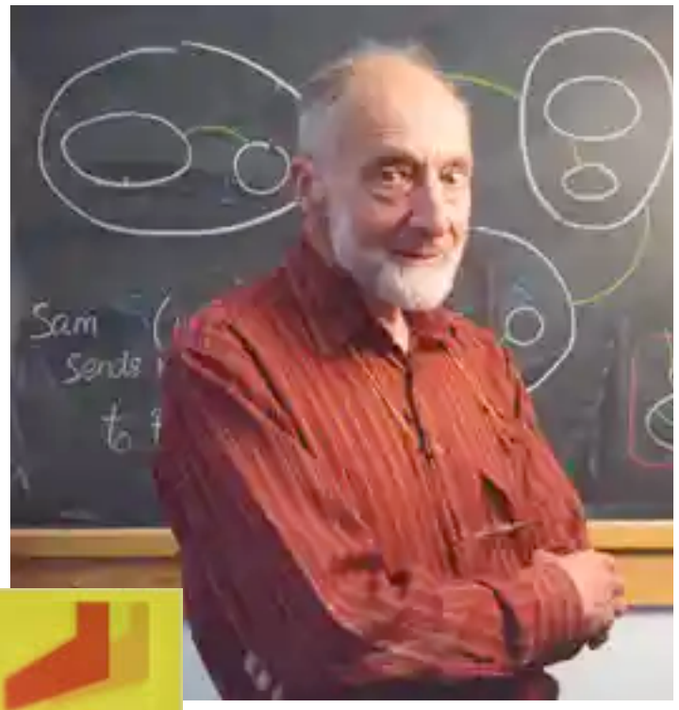
Questions/comments?

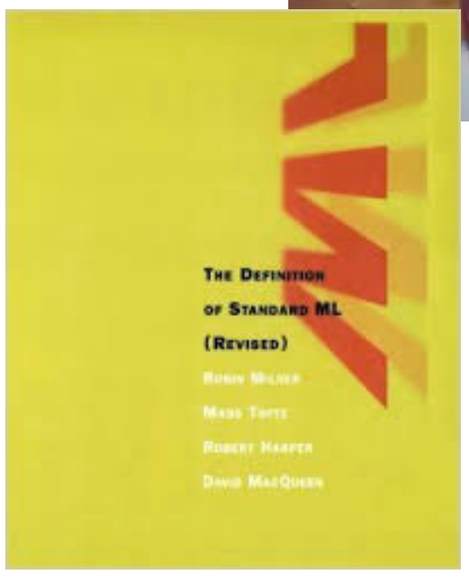
Next...

- Homework 0 (google form)
  - Help us get to know you!
  - https://forms.gle/P65LytJYbKA5MzBj9
- Homework 1: HellOCaml
  - Released today via Canvas
  - Due Tuesday Sept 10, 11:59pm
  - Get you back up to speed on OCaml

ML's History

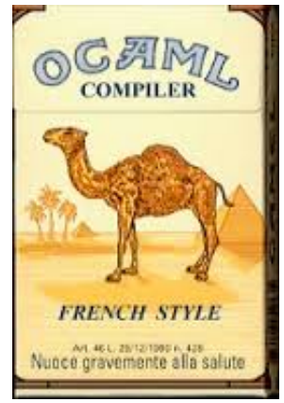
- 1971: Robin Milner starts the LCF Project at Stanford
  - "logic of computable functions"
- 1973: At Edinburgh, Milner implemented his theorem prover and dubbed it "Meta Language" – ML
- 1984: ML escaped into the wild and became "Standard ML"
  - SML '97 newest version of the standard
  - There is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





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
- See https://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
- ocamlbuild — a compilation manager
- utop — a more fully-featured interactive top-level

- opam — package manager

A Simple Language

- Consider a simple language with expressions and commands
- Example program: computes factorials

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

- To describe this language we need:
  - Syntax: what sequences of characters are legal programs?
  - Semantics: what does a program mean?

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>
```

- **Concrete syntax** (grammar) for a simple imperative language
  - Written in "Backus-Naur form"
  - <*exp*> and <*cmd*> are nonterminals
  - '`::=`' , '`|`' , and <…> symbols are part of the meta language
  - Keywords, like '`skip`' and '`ifNZ`' and symbols, like '`{`' and '`+`' are part of the object language
- Need to represent the **abstract syntax** (i.e., hide irrelevant parts of concrete syntax)
- Implement the operational semantics (i.e. define behavior/meaning of the program)
- See file `simple.ml`