《编译原理》

第1讲: 导言

何冬杰
重庆大学

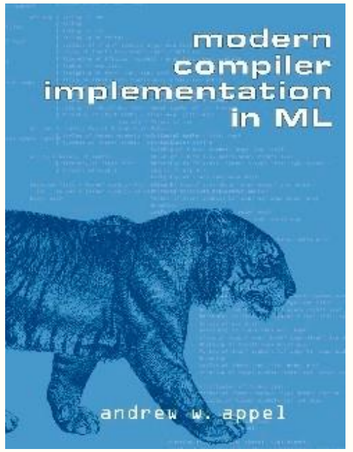
课程概要

群聊：编译原理课程群

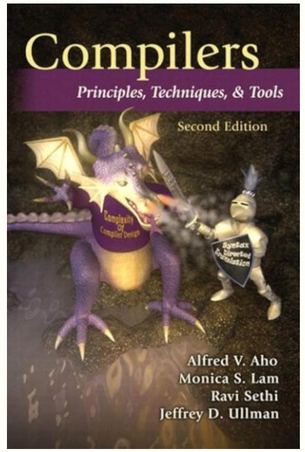
- 理论课：周一3-4节、周三1-2节[1-4,6-11周]
  - 地点：DZ403，DZ216
- 实验课：周日6-9节[5,7-8,10周]
  - 地点：DS1512
- 学时：40+16=56节
- 网站：https://plc-cqu.github.io/teaching/index.html
- 主讲教师：何冬杰
  - 研究方向：编程语言与编译器
  - 先后在中科院计算所和UNSW的编译器组深造
- 助教：熊锋
  - 用OCaml编写了课程实验全套工具：编译器、解释器、汇编器、链接器等

该二维码7天内(9月13日前)有效，重新进入将更新

Textbook



虎书

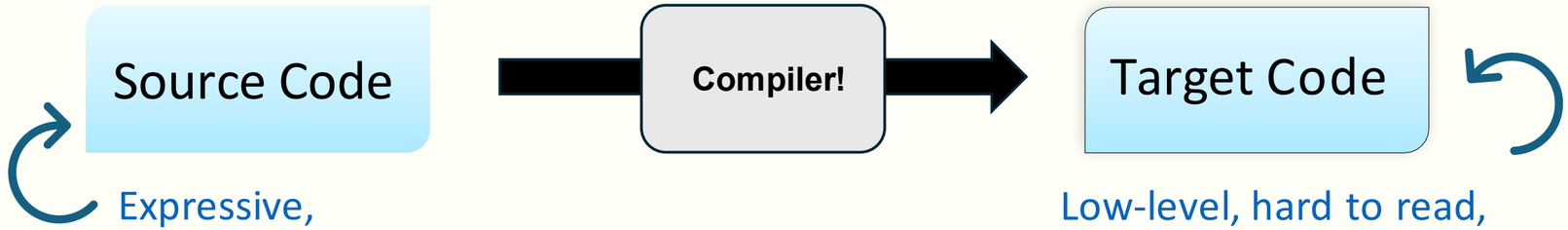


龙书

- In most cases, class materials should suffice
  - Lecture slides posted after the lecture
- Textbooks (*Recommended but not required*):
  - *Modern Compiler Implementation in ML* by Andrew W. Appel
  - *Compilers: Principles, Techniques, and Tools (2nd Edition)* by Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman

- Why Choose OCaml as the Programming environment?
  - 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
  - Less coding, more focus on compiler principles.

What is this course about?

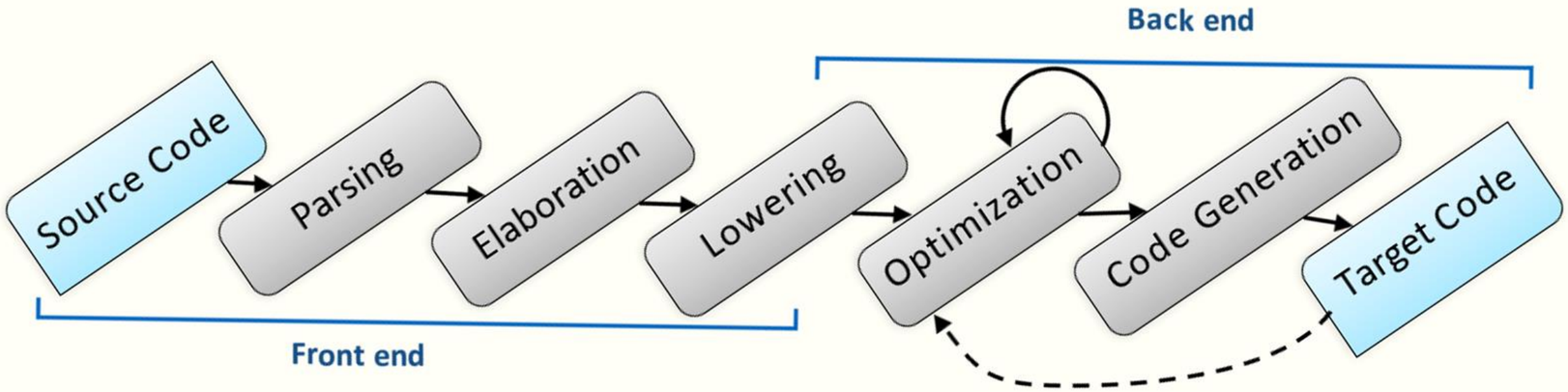


- How are programs written in a high-level language transformed into machine code?
- How can we ensure a program is somewhat meaningful?
- How is the program's memory managed?
- How can we analyze programs to discover invariant properties and improve their runtime performance?

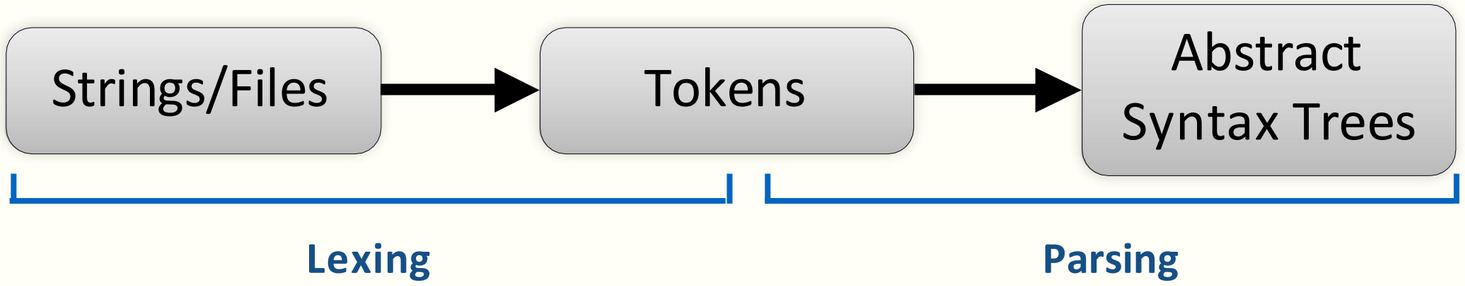
Historical Note

- 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...

Compiler Architecture



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

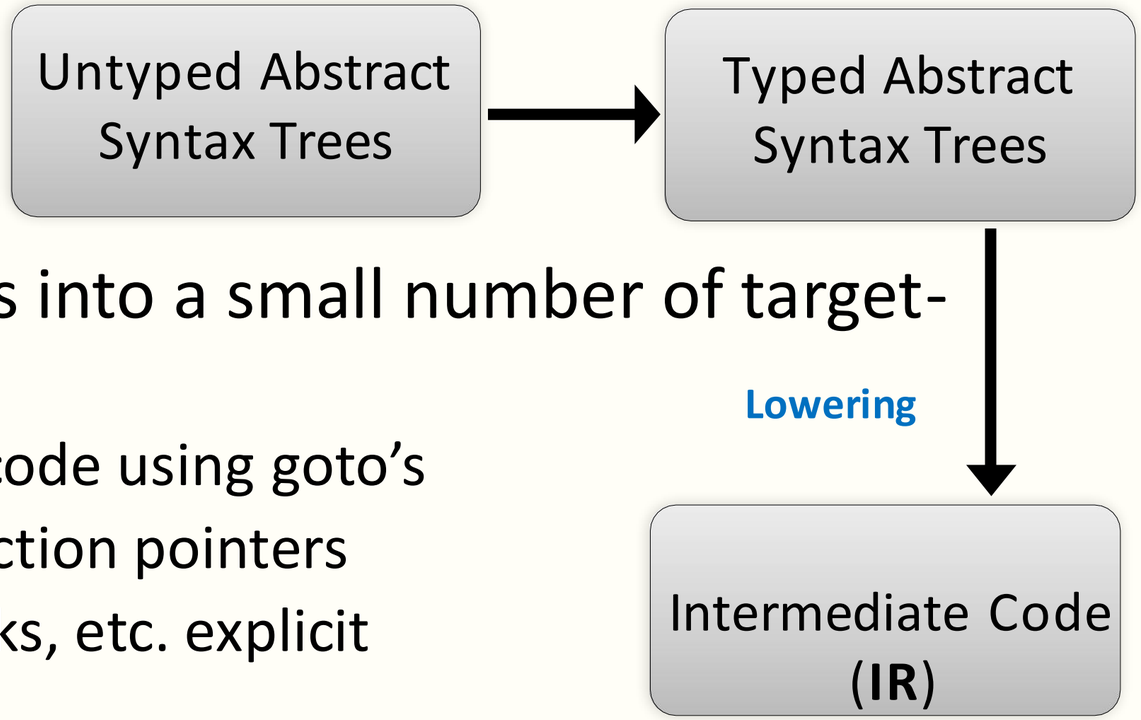


Parsing tools

- 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 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.

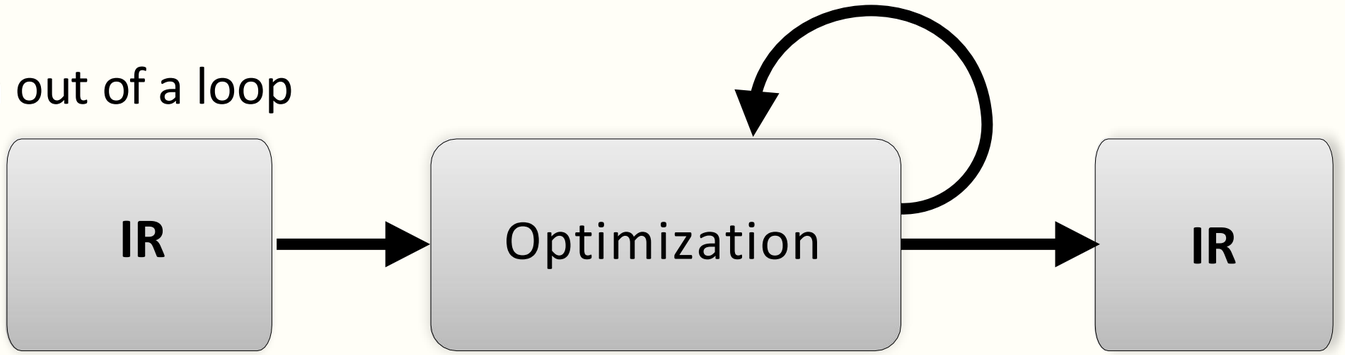
Elaboration && Lowering

- Elaboration: mainly type-checking
  - Check that operations are given values of the right types
  - Resolve variables, modules, etc.
  - 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 && Code generation

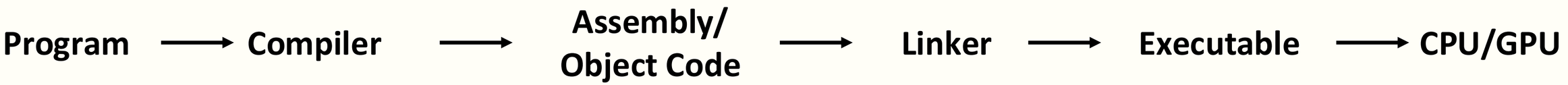
- Optimization:
  - Rewrite expensive sequences of operations into less expensive
    - e.g., constant folding: 3+4 → 7
    - e.g., lift an invariant computation out of a loop
    - e.g., parallelize a loop



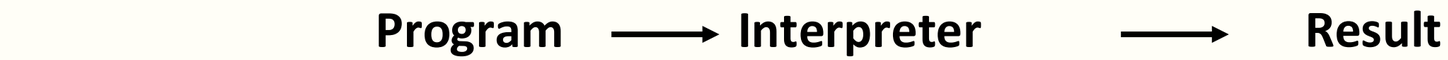
- Code generation:
  - Translate intermediate code into target code
    - Register assignment
    - Instruction selection
    - Instruction scheduling
    - Machine-specific optimization

Compiling vs. Interpretation

- Compiler translates a program
  - C/C++, Rust, Go, Fortran, ……



- Interpreter executes a program
  - Python, Ruby, JavaScript, PHP, ……



- Modern Implementation often combine both
  - Java, C#, Python(Cpython), Ocaml, ……

Example： compile, assemble, link, interpret

File: a.ml

```
let add x y = x + y
```

File: main.ml

```
let () = let r = A.add 3 4 in
        Printf.printf "3 + 4 = %d\n" r
```

Compile, Assemble, Link, and Execute

```
# ocamlopt  -S –c a.ml
// produce a.cmi, a.cmx, a.s, a.o
# ocamlopt  –S –c main.ml
// produce main.[cmi|cmx|s|o]
# ocamlopt –o main a.cmx main.cmx
// link and produce the executable
# ./main
// output 3 + 4 = 7
```

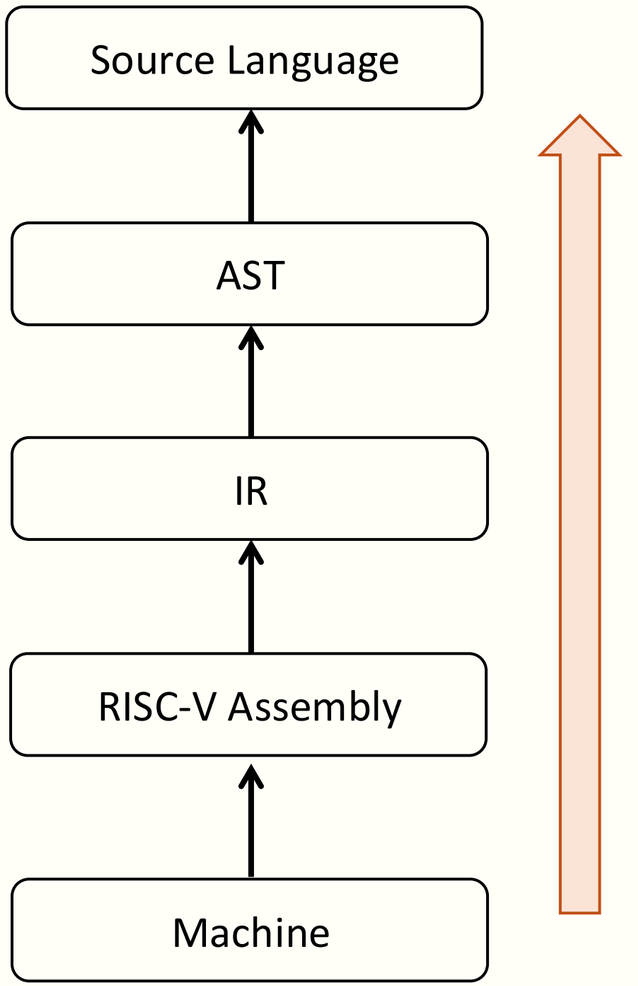
Compile, Link, and Interpret

```
# ocamlc –c a.ml
// produce a.cmi, a.cmo
# ocamlc –c main.ml
// produce main.cmi, main.cmo
# ocamlc –o main a.cmo main.cmo
// link and produce the executable
# ocamlrun main
// interpret and output 3 + 4 = 7
```

Course Design

#	课程内容	核心内容
1	OCaml 简介	**函数式**编程、Pattern Matching、ADT、**模**块
2	IR **与解**释执行	IR、**基本**块、控制流、Interpreter
3	RISC-V 汇编、调用约定与链接	**指令、寄存器、**栈帧、Calling Convention、Linking
4	**指令**选择	IR → Machine Instructions、Tree Matching
5	**寄存器分配**	Virtual/Physical Registers、Liveness、Graph Coloring
6	编译器优化	CFG、**数据流分析**、Constant Propagation、DCE **等**
7	**运行**时系统与垃圾收集	Stack、Heap、Allocation、GC
8	**抽象**语法与 AST	Expressions、Statements、AST、Pattern Matching
9	语义分析：作用域与符号表	Scope、Name Resolution、Symbol Table
10	类型检查	Type System、Type Checking、Type Errors
11	IR **生成**：AST → IR	Lowering、Control Flow、Temporary Values
12	语法分析	CFG、Recursive Descent、LL、LR
13	词法分析	Regular Expression、NFA、DFA、Lexer
14	**面向**对象语言支持	Object Layout、Inheritance、Dynamic Dispatch
15	**函数式**语言支持	First-class Functions、Closure、Environment
16	现代编译技术概览	SSA、JIT、LLVM/MLIR、GC、**并行**/GPU、AI 编译等

The Abstraction Ladder



Who should take this course?

- 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

Assessment

- Roughly:
  - 60% final exam
  - 30% projects (~6 projects total, not equally weighted)
    - Submit them online
  - 10% participation and assignments
    - E.g., attend lectures and engage in discussion, finish assignments, ...
    - 要求所有学生全勤，感觉人数不够63时，会根据点名册把缺勤学生找出来

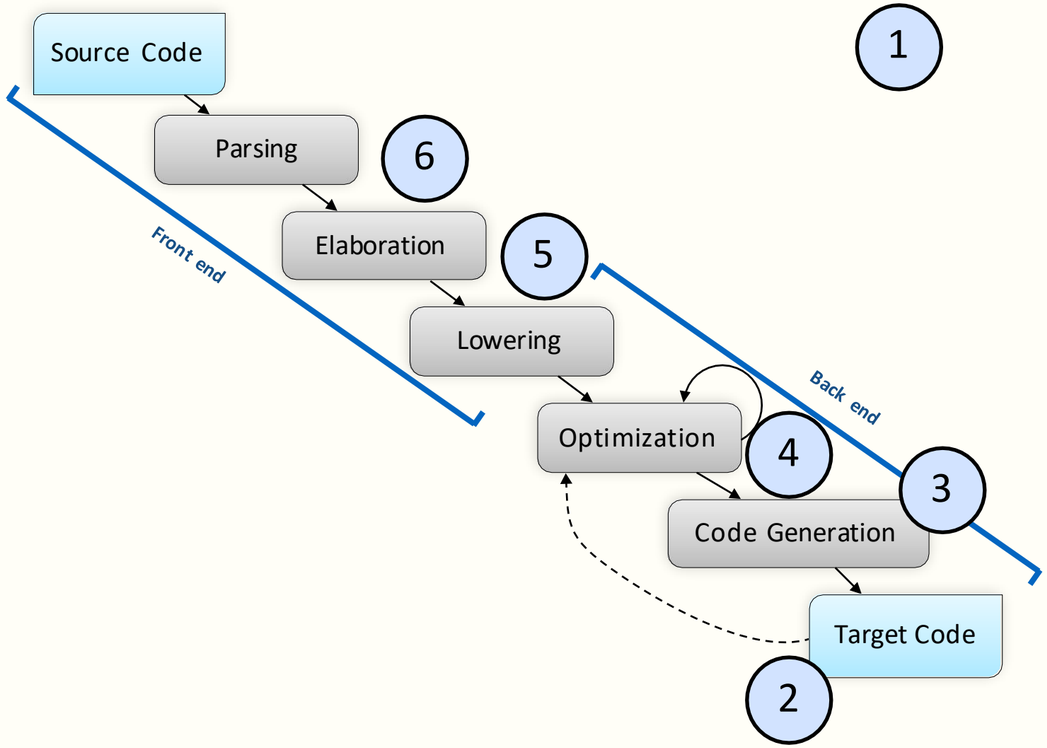
Projects

- 6 projects, mostly implementing parts of a compiler
- Typically 2-3 weeks per project
- Programming environment: in **Ocaml** 

https://ocaml.org/
  - Ideal for writing compilers!
- Implementation heavy course!
  - All submissions must type-check and compile
  - Multiple projects out at same time
- Online Judge:
  - http://47.109.176.53:8088 （ http://compiler.plc-cqu.cn 备案中）

Projects

- ❑1. HellOCaml
  - Refresh OCaml coding
  - Implement interpreter and Understand IR
- ❑2. RISC-V
  - RISC-V Assembler and Linker
  - Understand the machine we are targeting
- ❑3. Code Generation
  - Compile IR to RISC-V
  - Garbage collection and Register allocation
- ❑4. Dataflow analysis and optimizations
- ❑5. AST
  - Type checking
  - Simple lowering: from AST to IR
- ❑6. Lang: our language
  - Simple front-end: Lexing and parsing
  - Structs, function pointers, Classes



作业与实验说明

❑作业

- 独立完成，电子形式（PDF）提交到助教邮箱278487953@qq.com
- 允许纸上完成后拍照或扫描上传，需确保内容完整、字迹清晰
- 截止时间：发布后一周内提交（100%），两周(60%)，三周（30%），其他0%
- 作业反馈：发布后第二周公布参考答案

❑实验

- 独立完成，OJ上提交，自动评测（最多允许提交10次，取最高分）
- 允许讨论，咨询AI工具，但禁止分享代码
- 截止时间：发布后三周内，逾期提交不计分

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.

References

- We build this course by primarily referencing the following materials:
  - CS 153: Compilers, Harvard University, by Prof. Stephen Chong
    - https://groups.seas.harvard.edu/courses/cs153/2019fa/schedule.html
  - Compiler Construction, University of Cambridge, by Jeremy Yallop
    - https://www.cl.cam.ac.uk/teaching/2324/CompConstr/materials.html
  - CS 143: Compilers, Stanford University, by Keith Schwarz
    - https://web.stanford.edu/class/archive/cs/cs143/cs143.1128/

Questions/comments?

群聊：编译原理课程群

- **用于即**时/**离**线交流
- 请各位同学使用**“学号-姓名”格式作**为名片/**昵称**

该二维码7天内(9月13日前)有效，重新进入将更新