COMP 430: Language Design and Compilers
Spring 2024

Instructor: Kyle Dewey (kyle.dewey@csun.edu)
Course Web Page: https://kyledewey.github.io/comp430-spring24/
Office: JD 4419

Important Note About a Possible Work Stoppage During Spring'24
The California Faculty Association (the labor union of Lecturers, Professors, Coaches, Counselors, and Librarians across the 23 CSU campuses) is in a difficult contract dispute with California State University management. It is possible that we will call a strike or other work stoppage this term. I promise to promptly inform you of any schedule disruption. Our working conditions are your learning conditions; we seek to protect both. For further information go to www.CFAbargaining.org

Course Description (From the Catalog)
Examination of the issues involved in the design and subsequent implementation of programming languages. Considerations of implementation difficulties, including various features in a programming language. Tools and techniques to facilitate both the processing of programming languages and the building of programming processors.

Learning Objectives
Successful students will be able to:
- Design a programming language with (at least):
  - Concrete and abstract syntax
  - Statically-checked types
  - Expressions
  - Subroutines
- Implement a compiler for the designed language, complete with:
  - A tokenizer
  - A parser
  - A typechecker / static semantic analyzer
  - A code generator

Course Motivation
A common question with compilers courses: *When am I ever going to need to implement a compiler?* I'll answer that upfront: probably never. However, consider some related questions:
- *When am I going to need to reuse my own code?* - All the time.
- *When will I need to understand how a language works?* - All the time.
- *When will I need to work in a team?* - All the time.
- *When will I need to understand why a language was designed in a certain way?* - At least as often as you evaluate new programming languages. Languages rise and fall frequently, and it's important to know what's worth your time and what isn't.

In this class, you'll incrementally build your own compiler, likely starting from scratch. You'll live with your coding decisions (good or bad), and will likely have to revisit them. You'll gain a better understanding of how languages work, and understand what is involved with language design. Perhaps most importantly, you'll learn that **languages aren't magic**: you can implement your own, and they are surprisingly straightforward to implement once you understand the basics.

What this Course Is and Is Not

This is a project-based, implementation-oriented course. Our focus will be on modern compiler development. Grading is based on a series of assignments which will allow you to incrementally implement your compiler. While these assignments accomplish the same high-level tasks between all students, exactly what these assignments are depends on your particular language (e.g., the assignment implementing typechecking will differ for Teams A and B, since both teams implement different languages).

Because this is a project-based course, there are no exams, though the final exam slot may be used for team presentations. Your compiler code serves as evidence of understanding class concepts. Additionally, since the focus is on modern compilers, I'm intentionally **not** planning to cover certain topics in-depth, namely:

- Efficient parsing (e.g., LL, LR, flex, bison, etc.). We won't need very efficient parsers for what we plan to do. Some modern compilers forego efficient parsing entirely. Many of these algorithms and tools were developed at a time when memory and computing resources were scarce, but this is no longer true.
- Very low-level concerns like register allocation or machine-specific optimizations. It's rare nowadays to compile directly to machine code, given existing general-purpose low-level targets like LLVM, the JVM, and CIL. Additionally, these topics are completely irrelevant if everyone selects high-level compilation targets for their projects.

Textbook

No textbooks are required. That said, the following books may be of interest to you:

- Programming Language Pragmatics, by Michael Scott. Discusses a variety of programming language features from a variety of paradigms, and certain key interactions. Good for language design, though light on compilation.
- Compilers: Principles, Techniques, and Tools, by Alfred Aho, Monica Lam, Ravi Sethi, and Jeffery Ullman. AKA the "Dragon Book" (there is a dragon on the cover). Classic, commonly-used textbook, which discusses low-level compiler details. Heavy focus on parsing, and it's better-suited as a reference than an introduction.
- Modern Compiler Implementation in Java/ML/C (these are each separate books), by Andrew Appel. Good introduction to compilers, with lots of example code. Not very general, but will get you jumping in head-first.
- Engineering a Compiler, by Keith Cooper and Linda Torczon. Great resource on optimizations and low-level concerns. Not a gentle introduction to compilers, and limited information about things like types.
- Types and Programming Languages, by Benjamin Pierce. Specifically about typechecking and *type theory*, specifically as it relates to a number of common programming language features. Highly specialized and math-heavy; can be

intimidating, and likely not useful unless your language has some advanced type features.

Graded Components

All graded components tie directly to your project. Because of this, exactly what you need to do for each component is dependent on your language design. Similarly, how difficult each component will be is very dependent on your language design. As a result, when you submit your language design proposal, you will suggest how much each component should be worth, for a total of 100%. I'll provide feedback on this and potentially adjust these percentages in a way that reflects how much work each one is expected to be. This is why there is a minimum and maximum range for project-specific components.

Component	(Typical) Minimum Percentage	(Typical) Maximum Percentage
Language Design Proposal	5%	
Lexer	2%	10%
Parser	5%	20%
Typechecker	10%	50%
Code Generator	10%	50%
Language User Documentation	9%	
Presentation	6%	

For each of the lexer, parser, typechecker, and code generator, you will submit each component **twice** - once for an initial round of feedback and a tentative grade, and a second time for a final grade. **If the tentative grade for the initial round of feedback is below 25%, your final grade for the component will automatically be deducted by 50%.** This means you cannot simply skip the initial feedback round and wait until the last minute to submit everything.

For example, say your parser proposal had your parser worth 10% of your final grade. The deadline for the initial round of parser feedback comes up, but you only implement a tiny portion of the parser, which receives only a 15%. Since $15 < 25$, your final submission for the parser would be capped at 50% of what it should be worth, meaning you'll only get 5% towards your final class grade (50% of 10 is 5).

How Grading Is Performed

For the lexer, parser, typechecker, and code generator, students will effectively grade themselves using industry-standard techniques. Specifically, for these components, students are expected to:

- Implement all necessary code
- Write unit tests testing the written code, including relevant assertions
- Run the unit tests, and generate code coverage information showing what parts of the code are covered by the tests. This coverage report will be provided along with your code submission.

On my end, I will:

- Check that the code coverage is relatively high (at least 80%, ideally 90% or above)
- Check that the assertions in the test suite are present, logical, and actually check meaningful things
- Check that all your tests collectively handle all the behavior planned from your language design proposal

With this in mind, the tests you write are just as important as your implementation code.

Students are not expected to understand exactly how to do this ahead of time; we will cover unit testing and code coverage in class.

Code cleanliness is not explicitly part of the grade. However, if I see an issue with a component (e.g., a failing test, an unhelpful assertion, a missing feature), cleaner code will be easier for me to read through and understand what the root cause of the problem is. If I can understand the problem, partial credit is more likely to be awarded.

Final Grades

Plus/minus grading is used, according to the scale below:

If your score is >=...	...you will receive...
92.5	A
89.5	A-
86.5	B+
82.5	B
79.5	B-
76.5	C+
72.5	C
69.5	C-
66.5	D+

If your score is >=...	...you will receive...
62.5	D
59.5	D-
0	F

Late Policy
Unless prior arrangements have been made, for each day a a component is late, it will be deducted by 10%. Assignments that are submitted more than 10 days late will not receive any credit. The only exception is the final deadline for everything, where **everything must be submitted by the deadline (don't miss this one!)**.

Special Note About Teams
Considering the amount of work this class demands, it is highly recommended to form teams. To ensure a fair distribution of work among teams, peer evaluations will be used. Additionally, I will count the number of edits made to code per student per graded component (we will use revision control via GitHub, making this easy to do). If your number of edits is significantly lower than everyone else's without explanation, I will penalize you (and only you, not your team), accordingly. If you think a component is completely implemented already, code can **always** be added in the form of tests. If you are worried that you might not be contributing enough, talk to me and we can try to figure out a way to distribute work.

Plagiarism and Academic Honesty
You are permitted to collaborate as much as you'd like. Because of the nature of the course, it's not really possible to take someone else's code as your own from within the class (i.e., Team A's typechecker won't work for Team B). If, however, you use code from somewhere else, you must **cite** it. Any violations can result in a failing grade for the assignment, or potentially failing the course for egregious cases. A report will also be made to the Dean of Academic Affairs. Students who repeatedly violate this policy across multiple courses may be suspended or even expelled. From an industry perspective, blindly taking code from other sources can have severe consequences, as this could violate licensing agreements and open a company up to a lawsuit.

Communication
In general, any questions should be made through Canvas. You can also email me, though I'm usually much faster to respond to Canvas than my general email. Within your team, it's recommended to create a Discord or Slack for rapid messaging.

Class Feedback
I am open to any questions / comments / concerns / complaints you have about the class. If there is something relevant you want covered, I can push to make this happen. I operate off of your feedback, and no feedback tells me “everything is ok”.

Class Structure and Content
The initial course content is relevant to everyone, so this will be delivered via typical synchronous lectures. You'll then submit a language design proposal defining your project. Soon afterwards, exactly which parts of course content are relevant to you will depend on your language design proposal. For example, someone implementing an object-oriented language probably won't find content on higher-order functions to be relevant, and someone implementing a language that compiles to JavaScript won't find content specific to an assembly-level target relevant.

For these reasons, content that could be project-specific is tentatively planned to be delivered in an asynchronous format, via prerecorded lectures. We will still meet synchronously at this phase of the course, but the class time may be used like a lab wherein you can collaborate or ask me questions. This structure was adopted based on student feedback from prior versions of the course.

Class Schedule and Component Due Dates (Subject to Change):

Items in **bold** are tentative deadlines.

Week	Monday	Wednesday	Friday
1	1/22: Introduction, motivation, project information	1/24: Project information, team formation, feature survey	1/26:
2	1/29: project information, team formation, feature survey	1/31: feature survey, grammars, lexing	2/2:
3	2/5: grammars, lexing	2/7: lexing, ASTs	2/9: **Language Design Proposal**
4	2/12: ASTs, architecting around ASTs, parsing	2/14: parsing	2/16:
5	2/19: parsing	2/21: parsing, type theory basics	2/23: **Lexer - Initial Feedback**
6	2/26: type theory basics	2/28: type theory basics, typechecking	3/1:
7	3/4: typechecking	3/6: typechecking, compilation basics	3/8: **Parser - Initial Feedback**
8	3/11: compilation basics	3/13: compilation basics	3/15:
9	~~3/18~~: Spring Recess (no class)	~~3/20~~: Spring Recess (no class)	3/22:
10	3/25: open lab	3/27: open lab	3/29:
11	~~4/1: César Chávez Holiday; Campus Closed~~	4/3: open lab	4/5: **Typechecker - Initial Feedback**
12	4/8: open lab	4/10: open lab	4/12:
13	4/15: open lab	4/17: open lab	4/19:
14	4/22: open lab	4/24: open lab	4/26:

Week	Monday	Wednesday	Friday
15	4/29: open lab	5/1: open lab	5/3: **Code Generator - Initial Feedback**
16	5/6: open lab	5/8: open lab	5/10:
17	~~5/13~~: Classes over	5/15: **Presentation During Final Exam Time (10:15 AM - 12:15 PM in JD 1538)**	~~5/17~~: Classes over, **Final Version of Whole Compiler; Documentation**