COMP 430: Language Design and Compilers
Spring 2022

Instructor: Kyle Dewey (kyle.dewey@csun.edu)
Course Web Page: https://kyledewey.github.io/comp430-spring22/
Office: JD 4419 (I will not physically be there); Zoom link for office hours posted on Canvas and available via email.

Special COVID-19 Message
The course is being run as a synchronous online course. We will never meet in person, though we will meet virtually via Zoom regularly each week at the assigned time (see SOLAR for meeting time details). Assuming you're enrolled in the course or on the waitlist, I have emailed the Zoom link to you. If somehow you do not have the Zoom link, email me at kyle.dewey@csun.edu.

The synchronous lectures will be recorded and made available through Canvas, in case students want to view them asynchronously. These lectures will only be accessible to CSUN students either enrolled or waitlisted for the course. However, **by enrolling in, or waitlisting this course, you consent to having any voice or webcam recorded.** That said, **I will never require you to use your webcam or speak**; only what you voluntarily send will be recorded. Questions can either be asked verbally or textually through the Zoom chat. I will verbally repeat any questions in the chat before answering them, but I will not identify who said the question. Even so, it's possible that your name will end up in the meeting recording, identifying you as a participant (Zoom sometimes unavoidably shows participant names in the recording).

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:
  - Concrete and abstract syntax
  - Statically-checked types
  - Expressions
  - Subroutines
  - Mechanisms for computation abstraction
- Implement a compiler for the designed language, complete with:
  - 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 on 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 will 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. Moreover, it's not uncommon for modern compilers to forego these things 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. From my own experience, I've been implementing programming languages in various capacities for the past eight years, and I've never needed to reach for any of this.
- Very low-level concerns like register allocation or machine-specific optimizations. It's rare nowadays to compile directly to machine code, given existing backends like LLVM and the JVM. Additionally, these topics are completely irrelevant if everyone selects high-level compilation targets for their projects.
- Topics related to runtime systems (e.g., garbage collection). This course focuses on code translation, not on how common translation targets work. We could easily spend an entire course on these topics.

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 somehow tie to the project. These are below:

Component	Percentage
Language Design Proposal	5%
Lexer - Initial	1.5%
Parser - Initial	2.5%
Typechecker - Initial	8%
Basic Expression Translation - Initial	8%
Subroutines/Control Structures Translation - Initial	6%
Abstraction of Computations Translation - Initial	9%
Language User Documentation	9%
Presentation	6%
Lexer - Final	1.5%

Component	Percentage
Parser - Final	2.5%
Typechecker - Final	8%
Basic Expression Translation - Final	8%
Subroutines/Control Structures Translation - Final	6%
Abstraction of Computations Translation - Final	9%
Project-Specific Custom Component	10%

The components with the initial/final split likely look a little strange; the reasoning for this split follows. Ideally, for each one of these components (the lexer, the parser, etc.), you will turn it in on time, it will be completely correct, and it will never need to be revisited. However, these are all unrealistic assumptions. As such, each component will be evaluated twice:

1. At the specific component deadline. This is to ensure steady progress is being made on the project, and to give timely feedback.
2. At the end-of-class overall project deadline. This is to ensure that any issues revealed at the first deadline were fixed, and that new bugs haven't crept in.

If a component is submitted late, the late penalty will only be applied to the first deadline, as opposed to the end-of-class overall deadline.

In rare situations, I may award bonus points for a particular portion. I will only do this if I feel you have gone above and beyond the requirements, particularly in relation to the rest of the projects. Bonus points are entirely at my discretion.

Grade Replacement

If your final grade for a component is greater than your initial grade for a component, then your initial grade for the component will be replaced with the final grade, **as long as you received at least 25% for the initial component grade**. This is to keep you motivated to work on your compiler even in the face of a bad grade. The 25% restriction is to ensure that some effort is put into the project consistently, and to discourage attempts to do everything at the absolute last minute.

How Grading Is Performed

For code-based graded components, students effectively grade themselves, using industry-standard techniques. Specifically, for every code-based component, students are expected to:

- Implement the code necessary for it
- 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 initial 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+
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 20%. Assignments that are submitted more than 5 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. Such penalties are not eligible for grade replacement. 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 (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.

Attendance
I will take attendance for the first two class sessions. If you miss both sessions and have not made alternative arrangements with me, you must drop the class, as per University policy (http://catalog.csun.edu/policies/attendance-class-attendance/). This policy is in place to help motivated waitlisted students enroll in the class. After the first week I will not take attendance, though you are strongly encouraged to attend.

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
Early in the course, you must submit a language design proposal defining your project. We'll cover content in the course that is relevant to at least one project. As the course progresses, content will likely get more specialized to individual projects. I'll make it clear when this happens; you're not responsible for content that's irrelevant to your project. It is also expected that some days (or parts of some days) will be used as a lab, in order to allow you to work on your project.

Class Schedule and Component Due Dates (Subject to Change):
Items in **bold** are tentative deadlines. Per the prior section, this is an approximation.

Week	Monday	Wednesday	Friday
1	1/24: Introduction, motivation, project information	1/26: Project information, team formation, feature survey	1/28:
2	1/31: project information, team formation, feature survey	2/2: feature survey, grammars, lexing	2/4:
3	2/7: grammars, lexing	2/9: lexing, ASTs	2/11: **Language Design Proposal**
4	2/14: ASTs, architecting around ASTs, parsing	2/16: parsing	2/18:
5	2/21: parsing	2/23: parsing, type theory basics	2/25: **Lexer**
6	2/28: type theory basics	3/2: type theory basics, typechecking	3/4: **Parser**
7	3/7: typechecking	3/9: typechecking, compilation basics	3/11:
8	3/14: compilation of expressions	3/16: compilation of expressions	3/18:
9	~~3/21~~: Spring Recess (no class)	~~3/23~~: Spring Recess (no class)	3/25:
10	3/28: compilation of expressions	3/30: compilation of expressions	4/1: **Typechecker**

Week	Monday	Wednesday	Friday
11	4/4: compilation of control structures, subroutines	4/6: compilation of control structures, subroutines	4/8:
12	4/11: compilation of control structures, subroutines	4/13: compilation of methods, inheritance	4/15: **Basic Expression Translation**
13	4/18: compilation of methods, inheritance	4/20: compilation of methods, inheritance	4/22:
14	4/25: compilation of methods, inheritance	4/27: additional topics as projects demand	4/29: **Subroutines / Control Structures Translation**
15	5/2: additional topics as projects demand	5/4: additional topics as projects demand	5/6:
16	5/9: additional topics as projects demand	5/11: additional topics as projects demand	5/13: **Abstraction of Computations Translation**
17	~~5/16~~: Classes over	~~5/18~~: Classes over	~~5/20~~: Classes over, **Project-Specific Component; Final Version of Whole Compiler**