Project Information

The entire course is designed around your project, and all graded components are related to it. This project is to be done in the following stages:

1. Select group members. Generally, numbers between 2-5 students are ideal, with 3-4 generally being optimal. Larger groups are permitted, but usually have problems communicating. A group of 1 is only permitted if you can show you have prior experience working on compilers.
2. Select a language design proposal, and possibly tweak it slightly. If you have prior experience working on compilers or language design, you may also elect to create your own.
3. Write a tokenizer for your language.
4. Write a parser for your language.
5. Write a typechecker for your language.
6. Write a code generator for your language, along with any runtime environment it may need (if applicable).
7. Give a short presentation going over your language.
8. Write user-facing documentation for your language.

Language Design Proposals

For the language design proposals, you may select any one of the following:

- pOOP Base, a basic, class-based object-oriented language with a syntax based on S-expressions (i.e., lots of parentheses, but is easy to parse). Compiles to C, which isn't object-oriented, so that adds some challenges when it comes to your code generator.
- pOOP Improved Syntax, similar to pOOP Base, but it uses a more human-friendly syntax. It compiles to JavaScript, which is object-oriented, but is prototype-based instead of being class-based. Compared to pOOP Base, the parser is more complex, in exchange for the code generator being easier.
- ScalellScript Inferred, a basic functional language supporting higher-order functions, type inference, alegbraic data types, and pattern matching. Syntax is based on S-expressions.
- ScalellScript Noninferred, like ScalellScript Inferred, but doesn't support type inference. The syntax is not based on S-expressions, so it looks a bit cleaner. This all means the typechecker is easier than the inferred version, but the parser is harder.
- Lowlang Structs, a C-like language that compiles to MIPS asembly. The syntax is based on S-expressions. It notably supports pointers and structs.
- Lowlang Function Pointers, like Lowlang Structs, except it supports function pointers instead of structs. This allows for calling a function without knowing exactly which function it is until runtime.
- Traitor, a vaguely Haskell-like or Rust-like language which supports higher-order functions and Typeclasses (called `traits` in Traitor, similar to Rust). Notably, Traitor does NOT support generics, which dramatically simplifies how the implementation of typeclasses works. The syntax is NOT based on S-expressions.
- Refcount, a simplistic language that supports structs and not much else, at least from the user's standpoint. Unlike Lowlang Structs, Refcount uses references instead of pointers, and all structs are heap-allocated behind references. Additionally, Refcount compiles to C, which already has support for structs. At runtime, reference counting is used to automatically reclaim allocated structs which are no longer in use. Structs are immutable, preventing cyclic reference issues which would lead to memory leaks. The syntax is NOT based on S-expressions. Most of the complexity is expected to be in the code generator and runtime environment, which will need to keep track of reference counts for structs, and automatically deallocate structs which hit a reference count of 0.
- Reclaimer, a very similar language to Refcount, except it:
  - Uses S-expressions
  - Has mutable structs

- Uses garbage collection instead of reference counting to reclaim structs which are no longer in use.

As with Refcount, most of the code complexity is expected to be in the code generator and runtime environment, with an even more complex runtime environment than Refcount. The tradeoff is that the parser for Reclaimer is simpler.

Once you've selected a proposal, you *may* choose to modify it *slightly*. You may want to change the name, tweak the syntax a little, or maybe add a simplistic feature. If you do elect to make any changes, use Track Changes in Word to make it clear as to what is being changed. If you have more significant edits you wish to make, it's strongly recommended to talk to me first.

Custom Proposals

If you want to make significant changes to one of these proposals, or are interested in developing a whole custom proposal, see here.

Compiler Language

You can select whatever language you like for implementing the compiler itself. However, certain implementation languages will be much more difficult to work with than others, and there is absolutely no benefit to making your life harder than it needs to be here. All else being equal, I'd recommend a high-level language which supports pattern matching. Pattern matching allows you to very directly do the following:

1. Look at some input value
2. If it matches this *pattern*, do this thing
3. Instead, if it matches this other pattern, do this other thing
4. ...and so on...

This sounds a lot like the more traditional `switch`, and they are related. However, pattern matching gives you some extra functionality, namely:

- Compilers will generally warn you if you forgot one of the possible cases. This helps avoid bugs related to forgetting a case. In practice, when working with languages, there are frequently dozens of cases to consider, making it easy to forget one.
- Patterns allow you to immediately extract out values from the input. This leads to very concise code. For example, consider the following Scala code which makes use of pattern-matching:

```
value match {
  case Plus(e1, e2) => ...
  ...
}
```

  The intention with this code is that it checks to see if an input is of the form `e1 + e2`, and if so, will do something accordingly. In contrast, this code with `switch` would be something like the following (in JavaScript):

```
switch (value.id) {
case PLUS:
  var e1 = value.e1;
  var e2 = value.e2;
  ...
}
```

  In JavaScript, separate statements were needed to extract out `value.e1` and `value.e2`.
- Most importantly, patterns can be nested, allowing you to concisely look for very specific things. For example, consider the following Scala code:

```
value match {
  case Plus(IntValue(0), e) => ...
  ...
}
```

The above code specifically looks for `0 + e`. With `switch`, the above code would be more like the following (in JavaScript):

```
switch (value.id) {
case PLUS:
  if (value.e1.id == INT_VALUE && value.e1.int_value == 0) {
    var e = value.e2;
    ...
    break;
  }
...
```

This can be very important for making things like peephole optimizations concise.

It may be worth looking into a high-level language with pattern matching, if you're not already familiar with one. If you're used to Java, I'd personally recommend looking at Scala, which runs on the JVM, interoperates easily with Java, and tends not to get in your way when you're first learning it. You may also be interested in OCaml, especially if you're interested in using LLVM bitcode as a target language (LLVM has native OCaml bindings).

All that said, diving into a new language is itself a big risk, so you may be better off suffering through some unclean code. If you're using an object-oriented language for implementation, you may be interested in the visitor design pattern, which can handle certain features of pattern matching (specifically, making sure all cases are covered). However, from my personal experience, the visitor design pattern is usually more trouble than it's worth, and I usually will do something like `switch` and will potentially cast objects frequently. This isn't considered good practice, but the fundamental problem is in the implementation language itself fighting you unnecessarily.

Grading (and Testing your Code)

Given that every project is different (perhaps radically so), grading becomes difficult. As such, I'm effectively offloading much of this responsibility onto **you**. It is **your responsibility** to demonstrate to me that your code works. The most effective way of doing this is via unit tests (to test individual components), and integration tests to make sure components work together correctly. If your tests test a wide variety of circumstances, this is good. It's even better if your tests have high code coverage (measured with a code coverage tool). Code without tests is unlikely to be evaluated well, especially if the code is difficult to follow.

Working in Teams

Each project is expected to be a massive undertaking, and the difficulty level has been designed with the assumption that each project will be backed by 2-5 people. To be clear, each project will require you to do the following:

- Learn specifics of how your chosen languages features work, along with how they are typically implemented.
- Learn specifics of your chosen target language, and strategies (if any) which people use to compile to your target language.
- Experiment with possible code translations by hand, to ensure that a translation fundamentally makes sense.
- Write likely thousands of lines of code which do complex, unfamiliar things.

- Write a **lot** of test code, and ensure that the test code sufficiently tests the compiler code.

None of this is expected to be easy. From my own experience, it took me **eight weeks** to solo implement a naive, non-optimizing compiler from Prolog to C. From another project, I already had a lexer, parser, and typechecker available. Matching my own development timeline to the course timeline, this development wasn't fast enough; I would have missed deadlines. The point: you almost assurredly cannot do this alone.

Working with teams is a scary prospect, given the classic problem of unbalanced workloads between team members. Using the team-oriented nature of this class to coast along on someone else's effort is **not** acceptable. To make sure people aren't coasting, I will be doing two things for each deadline:

1. On each deadline, you will submit an evaluation of yourself and of your team members. If you think someone is coasting, say so.

2. We will be using GitHub for all code development, and I will measure the number of code contributions (lines added/edited/removed) that each team member has performed across all branches. If I see that someone's contribution count is significantly lower or higher than everyone else's, and no explanation is in the peer evaluations, I'll investigate. There are certain benign situations where this can happen; e.g., someone starts reading a lot of documentation on a target language, and diverts time away from coding **with the team's consent**. If you have far less code without explanation, **you, and only you, will not receive credit for the deadline**. Conversely, if you have far more code without explanation, I will contact you to ask what's going on.

   The reason for doing *both* peer evaluations and GitHub monitoring is to avoid situations where someone feels pressured to give someone a good peer evaluation, and similarly to avoid situations where it's one person's word against another's. Peer evaluations are easy to spoof, but code contributions are not.

Language Documentation

You must provide documentation for your language. You have total control over the format of this documentation; for example, you could create a website for it with links to different parts (GitHub has free web hosting and makes this relatively easy), or create a more traditional document. No matter the format you use, your documentation must cover the following:

- **Why this language, and why this language design?** What kind of problems might this language solve, and why was it designed this way? If you picked one of the existing languages, you can just say why you picked it, and it's perfectly acceptable to say you thought it'd be easy. However, you should still give an assessment of what you think you could realistically do in the language. If you created your own custom language, say why you picked the design you picked; saying it was arbitrary is also acceptable. Similarly, it's fine to say that some component of your language was intentionally restricted to make implementation easier - all real languages do this! Overall, I'm interested in getting a window into your thought process.
- **Code snippets in your langauge highlighting features and edge cases, along with relevant explanations.** These snippets should give a general sense of what your language provides, and what sort of restrictions your language has. It's ok, and even encouraged, to show snippets that don't work correctly or are otherwise strange; no programming language is perfect, and it's best to point out flaws upfront rather than have a user (or me) find a flaw after some digging.
- **Known limitations.** What can your language not do? Is there anything particularly hard to do with your language? This may be redundant with your code snippets, depending on how detailed your snippets are.
- **Knowing what you know now, what would you do differently?** If you had to start the project all over again from the start, is there anything that you would do differently? For example (no need to answer any of these specifically; this is just to get you thinking about possibilities):
  - Would you design anything differently?
  - Would you choose a different development tool (e.g., editor, build environment, communication method, etc.)?

    - Would you choose a different target language?
    - Would you change anything about how you communicated within the group?

  Overall, I'm interested in the sorts of lessons learned during the development.
- **How do I compile your compiler?** What command you use to compile things, and any special setup involved.
- **How do I run your compiler?** Similar as the compilation.
- **Formal syntax definition.** This should be copy/pastable from your language proposal.

Your documentation can include more information than the above if you so choose, but it must cover the above elements.

In terms of the size, there are no hard minimum or maximum sizes. That said, I would expect that you'd need at least three pages to cover everything, and that you're probably going into too much detail if you need more than ~20 pages.

Internally, I will use this documentation to assist in the grading process. The documentation is your opportunity to guide me through what you've done, and to point out both the good and the bad. It's easy to put documentation off and to do a subpar job of it, but keep in mind that the documentation is worth a decent chunk of your grade (see the syllabus for details). I intentionally have this scored this way to incentivize you to spend time on the documentation. If you'd like early feedback on the documentation, just let me know and I'm happy to look at early drafts without grading anything.

Presentation

Your group must present your language, tentatively during the final exam slot. This presentation overlaps with some of your documentation. This presentation must cover the following:

- If you created a custom proposal, a high-level description of your language design. Are there any languages which you based this on? (e.g., is it Java-like, C-like, etc.)
- What is your target language, and what language did you implement your compiler in?
- What sort of key features does your language have, including your major and minor features?
- What sort of key limitations does your language have?
- What were the biggest challenges that arose during development? How did you respond to these? (e.g., did you change your syntax, did you change how something worked, etc.)
- What sort of lessons were learned? Closely related: if you could do it all over again knowing what you know now, what would you do differently?

Your presentation may cover additional content, if you wish.

As for the format of your presentation, the only hard constraint is that you will be given only a certain number of minutes, TBD. Some related details follow:

- You may use slides if you wish, but they are not required.
- It is not necessary for all members of your group to speak.

The overall purpose of the presentation is to show off what you've done to the class, and to give an opportunity for us to ask questions directly before final grading.