For this project, you will implement a compiler, interpreter, or translator for a language with a specific, domain-specific task. The project may be completed individually or in a team of up to three students. The tasks are split into sprints, each with its own deadline:

1. [Team Signup and Proposal](https://mcs.utm.utoronto.ca/~peters43/488/project/index.shtml#signup) (by email): ~~Wed, Jan 11~~ Fri, Jan 13 @ 5 p.m.
2. [Sprint 0: Language Specification](https://mcs.utm.utoronto.ca/~peters43/488/project/index.shtml#spec): Fri, Jan 27 @ 5 p.m.
3. [Sprint 1: Parser](https://mcs.utm.utoronto.ca/~peters43/488/project/index.shtml#parser): Fri, Feb 10 @ 5 p.m.
4. [Sprint 2: Backend](https://mcs.utm.utoronto.ca/~peters43/488/project/index.shtml#backend): Fri, Mar 3 @ 5 p.m.
5. [Sprint 3: Type Extension](https://mcs.utm.utoronto.ca/~peters43/488/project/index.shtml#extensions): Fri, Mar 17 @ 5 p.m.
6. [Public demo](https://mcs.utm.utoronto.ca/~peters43/488/project/index.shtml#demo): Wed, Mar 29 in class
7. [Sprint 4: Optimization](https://mcs.utm.utoronto.ca/~peters43/488/project/index.shtml#extensions): Fri, Mar 31 @ 5 p.m.

Each sprint will be submitted to your git repository. You will need to submit a brief report describing what was implemented (or not) and noting any important design decisions. You will also need to submit a script that runs your code and demonstrates that the submission meets the requirement for the phase. If you choose a target we cannot easily access (for example, if you are compiling a ROM), then you may submit a video or artifacts in lieu of a script.

**Team Signup and Proposal**

Your first task is to form a team and inform Andrew. Send an email including the utorids of yourself and your teammates. (Please copy all of your teammates on the email, so that I know that they aren't being "volunteered" onto a team.) Your email should also include a brief description of your proposed project. In particular, tell me (a) what your input language will be, (b) what your target will be, and (c) why you think this project is particularly interesting. In response, I will prepare a git repository for you and send feedback on the project proposal.

You are free to select the domain for your project, including the input language and the target. However, the input language and target must have a few specific attributes. First, the input language must include:

* at least two primitive types and related operators,
* a conditional, loop, and procedure construct, and
* (for sprint 3) a non-primitive type or structure and related operations.

Second, the target must be clearly defined, and ideally, it would allow you to execute or otherwise validate your output. For example, compiling to C is easy, and it would allow you to easily execute your code. Compiling to a document would also lead to easy validation. Compiling to a specific architecture (like MIPS or x86) would be more difficult, and you should allocate extra time early in the term to make sure you can compile to the target.

**Sprint 0: Language Specification**

The goal of the language specification is to convince us that you have a viable project. You will be submitting a two-page PDF document that includes a description of the project (polished from the proposal) and one (or more) examples that demonstrate how the compiler will be used. In addition, you will submit a text file containing the lexical specification for the language. The lexical specification should be formatted as a grammar, as demonstrated in the week 2 exercise.

As you prepare your documents, consider these criteria that will be used to evaluate it:

1. Is the input clearly described, and does it include the required elements?
2. Is the target clearly defined, and is it a viable target? (i.e., Will additional work be required to make the output of this compiler executable for verifiable?)
3. Is the lexical specification complete (for the language features specified) and correct?

In addition to the required documents, you may wish to begin working on Sprint 1. In particular, you may wish to prepare a text document specifying the syntax for your language. The two text specifications will be used to prepare a scanner, using flex, and a parser, using cup.

**Sprint 1: Parser**

In the last phase, you created the lexical specification for your language. In this phase, you'll write the context free (parsing) grammar and then implement both. We hope the examples we have completed over the past few weeks will be useful. You may wish to start by reviewing the parsing exercise from week 4.

The primary deliverable for this sprint is code (the scanner and parser), but you will also need to prepare documents and artifacts that will help us to evaluate whether your code is operating correctly. We will expect:

1. The code for your scanner and parser. This code should take the input language specified in sprint 0 and transform it into syntax tree or other intermediate representation suitable for optimization and code generation.
2. A set of example inputs designed to demonstrate that your code fully parses the input language.
3. A script that, when run, compiles your scanner and parser and then runs it on the suite of examples to generate verifiable output files. (For example, you may output your ASTs as XML files.)
4. A brief (one or two page) PDF document that describes your intermediate representation, states any language structures that are not parsed, briefly describes the example inputs provided, and explains how to run the script and verify the output files.
5. Your context free grammar, expressed in Backus-Naur form, in a text file.

These criteria will be used to evaluate your submission:

1. Expressiveness: Does your CFG (text file) accurately represent the structures in your language? Is it free from ambiguity?
2. Correctness: Does your code implement your chosen input language?
3. Verifiability: Can we verify the correctness of the intermediate representations generated by your code? Are the examples provided complete?

The next section, the backend, could present significant challenges for many of you. If you're compiling to an executable format, the amount of detail you need to know (and implement) to actually product executable code will be daunting. If you're building a source-to-source translator, there's the possibility that your target will not work as you expect. Either way: I recommend pushing to finish sprint 1 quickly, so you that you can spend extra time on the backend.

**Sprint 2: Backend**

The primary deliverable for this sprint is the backend code -- the code that translates the output of your parser into your target language. However, like the last sprint, you will also need to prepare documents and artifacts that will help us to evaluate whether your compiler is operating correctly. We will expect:

1. The code for your backend. This code should take the output of your parser from sprint 1 (or some intermediate representation created from that output) and transform it into an artifact in your target language. For some of you, that will be executable code. For others, it might be a diagram or image.
2. A set of example inputs designed to demonstrate that your code can compile the input language.
3. A script that, when run, builds your compiler and then runs it on the suite of examples.
4. A brief (one or two page) PDF document that describes the intermediate representation you are using, explains how your backend translates it into the target language, states any language structures that are not supported, briefly describes the example inputs provided, and explains how to run the script and verify the output files.

These criteria will be used to evaluate your submission:

1. Correctness: Does your compiler generate executables (or non-executables!) that correctly represent in the input?
2. Completeness: Does your compiler handle all of the language features that you included in your proposal?
3. Verifiability: Can we verify the correctness of the compiled code? Are the examples provided complete?

**Extensions: Sprints 3 (Type Extension) and 4 (Optimization)**

The final two sprints are opportunities to extend your compiler with new features, to implement error checking, or to improve the quality of the code generated. While our original intention was for you to perform some sort of extension to the type system in the first sprint and some sort of analysis and optimization in the second, that split may not make sense for your particular project. Instead, you could, if necessary, implement two analyses / optimizations.

Possible examples of a type extension include adding support for a new type (pointers? compound types like structs? functions as first class objects?), implementing type checking or type inference, or enhancing the scopying system. Possible analyses and optimizations include peephole optimizations (performed in the backend), creation and use of a dataflow IR in the middle end, [static code analysis](https://en.wikipedia.org/wiki/Static_program_analysis) (to detect potential errors), or [the addition of pragmas to guide code generation](https://en.wikipedia.org/wiki/Directive_(programming)).

When selecting an extension, please consider (1) viability and (2) demonstrability. For the former: will you have time to complete the extension, given the number of people on your team, the deadline, and impending assigments in other courses? For the latter: will you be able to demonstrate that the feature works, both to the graders (Michael and Andrew) and to attendees of the demo session? Will the feature you choose be *interesting* to attendees of the demo session, and does it provide an opportunity to showcase your work to others?

Like the last two sprints, you will need to prepare documents and artifacts that will help us to evaluate whether your extensions are complete and correct. We will expect you to submit:

1. Updated code, including (if necessary), flags in the compiler to output code in various stages and with some optimizations turned on or off so that we can see the result of transformations being applied.
2. A set of example inputs designed to demonstrate how your extension or optimization works. These examples may be the same as in past sprints, or they may be a small addition to examples from past sprints.
3. A script that, when run, builds your compiler and then runs it on the suite of examples.
4. A brief (one or two page) PDF document that describes and motivates the extension or optimization you've implemented, explains how you implemented it, briefly describes the example inputs provided, and explains how to run the script and verify the output files.

For both sprints, we will be evaluating:

1. Quality of documentation: Is the extension or optimization implemented described clearly, so we understand what it does, how it works, and why it is worth implementing?
2. Correctness and Robustness: Does the extension or optimization generate correct code, and does it provide appropriate checks to handle unexpected or incorrect input?
3. Verifiability: Can we verify the correctness of your compiler and, in particular, the extension or optimization for this sprint?

In addition, for the final sprint, we will be evaluating the correctness, robustness (ability to handle errors or unexpected intputs), and quality of the design of your final, overall submission.

**Demo**

In the last class meeting of the term, you will demo your project to the other members of the class -- and to members of the broader UTM computer science community. The demo is slightly before the final due date for the project, so it is expected that your compiler won't be *quite* finished, but you should be very sure that the version you are demoing is stable. You may wish to create a branch immediately after sprint 3 as a backup.

On the demo day, half the groups will present in the first hour, and the other half will present in the second hour. When you're not presenting, you will be visiting the other groups to see their projects. Each group will set up at a table, using the whiteboard and the TV attached to their table to display any materials they need. You'll present a (1) description of your compiler, (2) motivation for creating it, and (3) a demonstration of an interesting feature (one of the extensions?) to people visiting your table.

The format of the demo will require that your pitch and demonstration be succinct. You'll need to pack a lot of information (at least "what" and "why") into 2-3 minutes. Michael and Andrew will visit each group to hear the pitch. We will be evaluating:

1. Clarity: Does your pitch clearly explain what you implemented?
2. Motivation: Can you get us interested in your project and the features you have implemented? Do we understand why you chose to implement the compiler that you did?
3. Visual Aids: Is your pitch well-supported by a polished demonstration or appropriate artifacts from your compiler? (Code probably isn't it ...)

At the end of the demo, any visitors will be asked to vote on the ``most polished presentation,'' ``most interesting project,'' and ``most impressive feature.'' The projects winning those accolades will get ... the adulation of your peers. And a bonus on the demo mark.