Project Requirements

The goal of this project is to **build a single system that uses multiple languages** to do its work.

It is very difficult to quantify the "size" of a project, but it should be enough to justify marks for 26% of a 3rd year CMPT course. In particular for this project, you should have:

- several pieces of interacting functionality (which give us the excuse to use multiple languages);
- overall complexity that leads to a thousand to several thousand lines of code (while also noting that lines of code is a horrible way to measure complexity of software);
- complexity added by the technical challenge of getting multiple languages to work together;
- a system that can be easily deployed and actually *works*.

A couple of possible topics for the project can be found on the project ideas page. You **do not have to choose from those ideas** and are free to choose any topic if it meets the requirements specified below.

Language Requirements

You project must be implemented **using three different languages**. The focus of the project will be cross-language implementations and communicating between languages. Some restrictions on your language choices:

- At least one "systems" language (like C, C++, Go, Rust).
- At least one "scripting" language (like Bash, JavaScript, Julia, Perl, PHP, Python, R, Tcl).

- C and C++ count as one language.
- Only *programming* languages count toward your language count (e.g. not SQL, HTML, CSS, etc).
- Languages should be reasonably appropriate to the work done with them.

It is expected that you write *some* code in each of the three languages, but not that you write an *equal* amount in each language. It is reasonable to have one language that is used fairly little (possibly just a little code to connect a library from that language), and possibly another that is a relatively minor part of the system.

Cross-Language Communication

You must **use two different ways to communicate** between languages your languages. These might include:

- A foreign function interface where you can somehow call a function from one language in another. If you search for "call *language1* function from *language2*", you'll probably find something (if it exists). See also SWIG which helps call C/C++ from many languages.
- A message queue such as RabbitMQ or ZeroMQ. The RabbitMQ tutorials give good RPC (remote procedure call) examples in many languages: you can use one language as the client and the other as the server.
- Create an RPC (XML-RPC or REST) server in one language and make a request to it from another. You can probably find an XML-RPC or REST library for either language and write relatively little code to connect them to your function/function call.

- A compiler toolchain that will compile both languages and let you call one from the other, like the GraalVM polyglot features or the various languages that compile to the JVM.
- Run the other language's code as an executable (exec and friends, or often a "system" function in many languages) and capture its output (stdout) into a string. Or have it write a temporary file and read it from the other program.

Option 1: A Virtual Machine + Chef

We have provided you with a template for the project using a VM that contains (mostly commented-out) setup for running code in many languages in a virtual machine. You can modify the setup recipe (`cookbooks/polyglot/recipes/default.rb`) to install whatever tools you need so your system can run.
By the time you're done, starting the VM (with `vagrant up`) should get your system into a runnable state. You may have a few components that need to be started by hand (like an RPC server). That is fine, but only one or two, and make sure they are clearly documented in your project summary (described below).
See the Project VM Deployment page for more information on working with the virtual machine and Chef recipes.

Option 2: Docker Compose + Containers

We have provided you with a template for the project using Docker containers that contains (mostly commented-out) setup for running code in many languages in separate containers. You can modify the setup recipe (`docker-compose.yml` and the various `*.Dockerfile` files) to start whatever containers you need so your system can run.

By the time you're done, starting the containers (with `docker-compose up`) should get your system into a running state. You may have a few components that need to be accessed by hand (like running an interactive program in a container). That is fine, but only one or two, and make sure they are clearly documented in your project summary (described below).
Note that with Docker, you are limiting your cross-language communication methods somewhat. It will be much harder to do a FFI-based call or to execute a program compiled from another language, because it will involve setting up tools for different languages in one container (which isn't usually done, and won't be done in base Dockerfiles you'll find).
See the Project Docker Deployment page for more information on working with a Docker Compose setup.

Project Summary

In your project repository, include a file `cmpt383.txt` or `cmpt383.md` and briefly summarize:

- What is the overall goal of the project (i.e. what does it do, or what problem is it solving)?
- Which languages did you use, and what parts of the system are implemented in each?
- What methods did you use to communicate between languages?
- Exactly what steps should be taken to get the project working, after getting your code? [This should start with `vagrant up` or `docker-compose up` and can include one or two commands to start components after that.]

- What features should we be looking for when marking your project?

The purpose of this is not to make you write an essay, just to guide the marking so we don't miss anything important. (There will be many projects, and the TAs are human.)

Marking

The marking scheme will include marks for: [These may change, but will give an idea of what we're looking for.]

- Problem: Is the problem you are solving interesting and appropriate to a polyglot system?
- Languages: Three appropriate languages have been used meeting the requirements of the project.
- Cross-Language Communication: Two different methods to communicate between languages have been used appropriately.
- Difficulty: The project includes features that required non-trivial implementations or techniques.
- Extras: The project includes interesting features, or uses techniques that go beyond the minimum requirements.