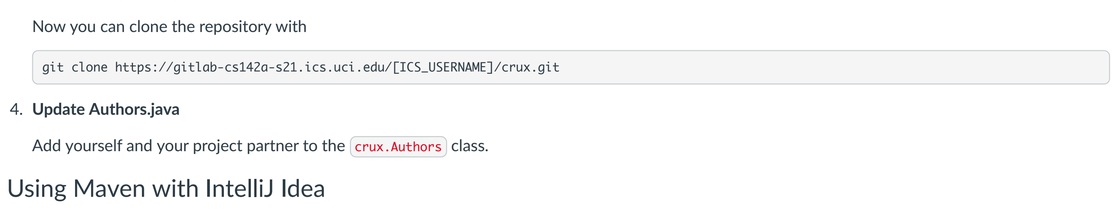
Project Overview

Introduction

In this class, we will build a compiler which translate programs constructed from a fairly small imperative programming language into executable x86 assembly. We will follow a traditional compiler design approach of partitioning our compiler into a sequence of stages, starting with a frontend and proceeding toward a backend in incremental steps. These stages will transform the input source code progressively into executable output.

This approach enables us to test and verify the correctness of our implementation of every stage. This is very important, as every stage builds on previous stages, so an error in our implementation in one stage may be only easily recognizable in other phases, which makes locating the source of a bug very cumbersome. The stages and intermediate data structures of our compiler are illustrated for reference in following diagram:



Development environment

The programming language used to implemented the compiler is Java 11. The frontend will use the parser generator library ANTLR. We will use Maven to automatically manage dependencies, build and test the project.

Each group will use their own Git repository provided by us to store and exchange the source code. There will be one repository which we will update with the templates and tests at CS142A GitLab. Use your ICS credentials to log in.

You may want to use IntelliJ Idea as IDE, as it has out of the box Git and Maven support and also has a Plugin for ANTLR. You can use any other IDE or, but note that eclipse needs some extra configuration to work nicely with ANTLR and it is a bit more of a hassle to use Maven.

Setting up your Git repository

Read the Git Guide to install and familiarize yourself with Git.

1. **Creating the fork**

   Then go to the template repository and press `Fork` in the top right corner. Follow the instructions and you will have your own personal copy of the project.

   If you work with a partner, only one of you should create a fork. The partner that creates the repository and thus owns it is also responsible for turning it in (see Submission).

2. **Adding your partner**

   To add a partner, go to `https://gitlab-cs142a-s21.ics.uci.edu/[ICS_USERNAME]/crux/-/project_members`. On this page, you can invite the other group member by their ICS username. Make sure to assign them the role `Maintainer`.

3. **Clone the repository**

   Now you can clone the repository with

   ```
   git clone https://gitlab-cs142a-s21.ics.uci.edu/[ICS_USERNAME]/crux.git
   ```

4. **Update Authors.java**

   Add yourself and your project partner to the `crux.Authors` class.

Using Maven with IntelliJ Idea

You can download IntelliJ Idea from Jetbrains. The community edition is free to use. We also recommend installing the ANTLR v4 grammar plugin, which brings syntax highlighting to ANTLR grammars that are used in the first assignment.

After the installation you can just open the folder containing the `pom.xml`. This will load the Maven project. To compile, open the Maven pane by clicking on the button labelled `Maven` button on the right side of the window. Then expand the tree so you can double click on `crux > Lifecycle > compile`.

Other interesting commands are `test` to run all unit tests and `package` to create .jar-file in the folder `target` to run the compiler easily from the command line:

```
crux/target $ java -jar crux-1.0-jar-with-dependencies.jar my-program.crux
```

To run and debug the compiler from within IntelliJ Idea, open the `Compiler` class and click on the green arrow next to the class definition by clicking on `Run 'Compiler.main()'`. To provide additional arguments click on `Edit 'Compiler.main()'...` and add them in the designated field.

Alternatively, you can also run Maven from the command line. To do so, follow the installations instructions for Maven. Make also sure, you have at least Java 11 installed and available in your path.

Using openlab

Alternatively you can use the openlab environment to develop your code. Remember that you have to sign in with ICS credentials:

```
ssh [ICS_USERNAME]@openlab.ics.uci.edu
```

If you are under Linux or Mac OS, ssh should be already installed. Windows 10 also provides SSH with recent updates. If you are on Windows and SSH is not available, you can use PuTTY instead.

Every time after signing in you need to run these commands to make the necessary tools available:

```
module load openjdk/12
module load maven
```

Then, after acquiring the code, navigate to the root folder of your project (the one that contains pom.xml). Maven provides many commands, the important ones however are:

- `mvn compile` to compile the project,
- `mvn test` to test the project,
- `mvn package` to create an executable jar file, and
- `mvn exec:java -Dexec.args="[arguments]"` to run the compiler.

Code structure

All code should be within the package `crux`. The following classes compose the starting template:

- `Driver` - The backbone of the compiler. This class is responsible for connecting the individual stages of the compiler and managing the execution. We will use this class for all testing purposes, so make sure this one works.
- `Compiler` - A pure convenience class aimed to show the features of the compiler and provide an easy-to-use command-line tool.
- `Authors` - You will have to add all group members to the `all` array in this class.
- `CompilerStageTests` - This is the test class we will use to run tests for grading on your compiler. This class only depends on the `Driver`, running tests by providing input programs and comparing the output of the compiler with some expected output. The files are loaded all from the resources folder in tests. You can simply add more tests by adding `TESTNAME.crx` and `TESTNAME.out`.

We will provide updates throughout the project to `Driver` and `Compiler`, so we recommend to not change these. We will replace `CompilerStageTests` with our own version, so do not change this in order to ensure our version will work and behave correctly.

There is also a file `pom.xml`, which is the build file describing the project and is required to compile the project with Maven. **Do not modify this file.**

Submission

Create file `README` in your repository with the following statement:

> We, your name and your partner, hereby certify that the files we submitted represent our own work, that we did not copy any code from any other person or source, and that we did not share our code with any other students.

To submit the assignment, you will have to retrieve the hash of commit you want us to grade with the command

```
git rev-parse HEAD
```

and submit this on GradeScope in a file named "sha" (please ensure the name is all lowercase and there is no file extension as well, so if you are on Windows make sure you can see file extensions). Of course you can resubmit as often as you want with a different hash before the deadline ends. **Make sure to push all commits to the repository and don't keep them just locally.**

This step is necessary, because we want to prevent students from manipulating timestamps of commits in git. As a result, we won't accept any other commit after the deadline, even if its timestamp indicates, that it was created before the end of the deadline.

Each time you submit, your repository will be cloned with that specific commit hash and it will run all public and private tests to determine the score of the given assignment. After the deadline, submissions will have an automatic point deduction based on how late the submission is (note that submission to gradescope will be what we use to determine
lateness of an assignment, not the commit hash).

You have to ensure, that the submitted commit is part of branch `master` and compiles. Otherwise we won't be able to grade your submission.

Resources

- Crux Language Specification
- Git Guide
- ANTLR
- IntelliJ Idea
- ANTLR v4 grammar plugin.