Project Overview

Introduction

Welcome to the CS:3820 project. For the project, you will implement and demonstrate a small programming language. You will choose your language from two options that I have provided. For each language, I have provided a description of the language, several (hopefully interesting and informative) examples, and a number of (potentially less interesting) test cases. This document gives an overview of what you need to do to complete the project, and how your project will be evaluated.

Languages

You will pick one of the following two languages to implement. Each langauge demonstrates an interesting choice in programming language design and implementation, different from tha choices made in Haskell.

Meta

The Meta language is based on the functional programing language Scheme. The particular feature demonstrated in Meta is *meta-programming*: writing programs to write programs. While meta-programming features have turned up in other languages — for example, C++ templates are a meta-programming feature — meta-programming is particularly fundamental to Scheme, and is the focus of the Meta project.

Logic

The Logic language is based on the logic programming language Prolog. Logic programming is based on a different way of thinking about programs. Rather that a program being a list of instructions or formulae, a logic program is a list of facts — some of which may depend on other facts. Executing a logic programs consists of searching the list (or database) to determine whether given facts are true.

Groups

You may do the project either individually or in groups of 2. You must decide on whether or not you want to work as a pair the week the project is released (week 8), and will submit your choice (as the problem of the week). Whether or not you choose to work as a group will not effect how your project is evaluated. If you wish to work in a group, you must identify and agree with groupmates yourself; I will not assign groups.

Artefacts

We will use GitHub classroom for distribution and submission of the project.

https://classroom.github.com/g/BbO3sQeO (https://classroom.github.com/g/BbO3sQeO)

Because this can be done as a group, GitHub will ask you to create or join a group when you follow the link. If you are working in a group, you should agree in advance which group member will create the group. If you are working alone, you will still create a group, but with only one member.

Please do not join a group uninvited. Please let me know immediately if someone has joined your group uninvited.

The starting point for your GitHub repository will include the source files for *both* projects. When you have chosen which project you are doing, you should remove the directory for the other project from your repository.

I will evaluate your project using the last commit in the `main` branch before the deadline.

Each project will include a skeleton `Main.hs` file, including a `main` method which reads in a list of files and calls an unimplemented `go` function on the (concatenated) contents of those files. You do not need to preserve this function if you would prefer to implement `main` differently. However, you should preserve the external interface to your implementation: that it can be called with a list of files, which it will process in order.

Cabal

The project templates include skeleton cabal files. You can use cabal to build and run your project. For example, for the Logic project, I see the following:

```
> cabal run
Resolving dependencies...
Build profile: -w ghc-8.10.2 -O1
In order, the following will be built (use -v for more details):
 - Logic-0.1.0.0 (exe:logic) (first run)
Configuring executable 'logic' for Logic-0.1.0.0..
Preprocessing executable 'logic' for Logic-0.1.0.0..
Building executable 'logic' for Logic-0.1.0.0..
[1 of 2] Compiling Parser           ( src\Parser.hs, ...\logic-tmp\Parser.o )
[2 of 2] Compiling Main             ( src\Main.hs, ...logic-tmp\Main.o )
Linking ...\logic\logic.exe ...
Your implementation starts here!
>
```

For more information, see The Cabal documentation (https://cabal.readthedocs.io/en/3.4/).

Process

So how do you implement a programming language?

In essence, you'll need to define three (maybe four) things:

1. The *abstract syntax tree* (or AST). This is going to be a Haskell data type that captures what programs look like. Think of it as an expanded version of the `Formula` and `Expr` datatypes you've seen in several examples over the semester.
2. A *parser* from the concrete syntax of your language to your AST. You probably also want to consider some kind of *printer* for at least the results of running programs in your language, so that you don't have to read the output of `Show` instances.
3. The *evaluator* for your language. You could potentially separate this into two tasks. On the one hand, you have the type of your evaluator, any `Functor` or `Monad` instances you want to write, and the primitive operations (or API) that accesses them. On the other hand, you have the evaluator itself, which interprets constructs of your AST using those primitives.

So, start with part 1, continue through part 3, right? Probably not. Trying to get your AST and associated types perfect before you start the evaluator or trying to write the entire evaluator at once are both recipes for frustration, rewriting, and potentially eventual disappointment. Instead, you want to identify independent subsets of the langauge that you can reasonably parse, evaluate, and print by themselves. For example, perhaps you can start with just numeric constants and operations on them. Working this way will avoid having to redo large chunks of code when you discover that your initial implementation ideas are either wrong, or just clumsier than you'd like.

Basically, you want to identify the smallest amount that you can test, build and test it. Then, expand that core adding one or two language features at a time.

Assessment

There will be two major and two minor components of the assessment of your project. The major components will be the functional correctness of your program, and your final exam discussion of the program. These will both be assessed only at the end of the semester. The minor components will be evaluating the style of your program (also only at the end of the semester) and a midpoint discussion during week 12.

Functional correctness

Your program will be evaluated against a suite of around 100 individual test cases, covering all aspects of the specification. The functional correctness of your program will be the number of tests passed out of the total number of tests, and will account for 30% of your semester grade (i.e., for 43% of your project score).

Test cases will consist of a set of files, to be passed to your evaluator, along with expected results. I will try to be flexible in interpreting your responses. For example, spacing, line breaks, and such wil be ignored in comparing your output the intended output. If there seem to be minor issues in comparing your output to the expected output, I may contact you to resolve the

differences. However, if there are serious issues, such as your code failing to compile, you will receive no credit for functional correctness.

A significant, representative subset of the test cases will be made available to you during the project. My testing harness will also be made available. Hopefully this should minimize any surprise in your final functional correctness score.

If you choose to work in a group, both group members will receive the same score for functional correctness.

Final exam

The final exam will take the form of an individual discussion during finals week, and will account for 25% of your semester grade (i.e., 36% of your project score). These discussions will be about 20 minutes long, and will consist of questions and guided exploration of your project. You should plan to share your screen for most of the discussion, to make some changes to your program (either repairing defects or adding new functionality), and to explain your code and your process for making these changes.

A rubric for the final exam will be distributed later in the semester.

Even if you have chosen to work in a group, your final exam and final exam score will be individual. If you identify particular areas on which you focused, I may attempt to focus the final exam on those areas. However, I expect both group members to be able to speak cogently to all aspects of the project.

Midpoint discussion

We will have a short (15 minute) discussion of your progress during Week 12 (November 9-13), worth 5% of your semester grade (i.e., 7% of your project score). The aim of the discussion is to ensure that you are making reasonable progress towards submitting the project at the end of the semester. As such, there are not specific objectives or deliverables for this discussion. I will expect you to lead the discussion, showing what you have accomplished to that point, and identifying particular challenges that you face in the second half of the project.

A rubric will be distributed before the midpoint discussions. IF you choose to work in a group, the midpoint discussion will include both members of your group. I would expect both members to contribute to the discussion. Both group members will receive the same score for the midpoint discussion.

Style

The final component of your project assessment will be an evaluation of programming style, contributing 10% of your semester grade (i.e., 14% of your project score). This will evaluate how well your program is constructed: for example, whether data types are well designed and have

clear meanings, functionality is sensibly broken down, and abstractions are well-used. It is not an evaluation of comment frequency or indentation style.

If you have chosen to work in a group, both group members will receive the same score for style.