Final Project Assignment

Your final project will be to implement an interpreter for a programming language. There are two main choices on what features to add to your language, that we will discuss below. But in all cases your submission will contain the following:

- You need to choose a good name for your language. This page[1] may act as inspiration. Or not.
- You will create a new GitHub repository, named after the name you choose for your language. It will contain the following.
  - For group work, the repository will be officially in one student's GitHub page. You should NOT fork it to the other students' accounts. Everyone will be accessing the same repository.
  - Under Settings in the repository's GitHub page, you should enable the "Issues" by checking the corresponding "Options" checkbox.
  - Still in Settings, in the "Collaborators" sidebar, you should add each teammate as a collaborator.
  - You should use GitHub issues to manage the project. You should create issues, use labels and milestones, and "assign" issues to team members if one member is assigned to look at a particular issue.
  - Part of your project grade will be based on an evaluation of the extent to which you used the various GitHub features, and how you linked them to the individual commits.
  - I strongly encourage you to work in pair programming[2] style, rather than simply split the work up among the team members.
  - If you run into problems with Git or GitHub, do not hesitate to ask for help. On occasion merging the work between two people can get complicated, if they have both checked out the same version, then did conflicting changes to it.
  - Always check out the most recent version of the repository before starting your work. Always commit your most recent work before ending the session. This will minimize the number of merge conflicts you have to deal with.
  - You can also create "branches" for trying out some experimental features without messing with the "main" branch. Talk to me if you want to do this. Git makes it fairly easy to do it.
- You should have a file named README.md in your root directory, with instructions on how one should compile and run the interpreter for your language. This should follow the so-called "GitHub flavored markdown" format, described here[3]. The same is true for all .md files.
- A document semantics.md that describes the specific syntax and semantics of all features in your language.

[1] http://c2.com/cgi/wiki?ProgrammingLanguageNamingPatterns
[2] http://www.extremeprogramming.org/rules/pair.html
[3] https://help.github.com/categories/writing-on-github/

- A document details.md that explains the choices you made in the surface and core languages, problems you encountered and how you addressed them and so on. You can add to this file as you go on, then clean it up towards the end.
- A file with examples of programming in your language. One should be able to send that as input to your interpreter as is. The examples should be rich enough to showcase the power of the language.
- The lexer, parser and interpreter for your language. These will typically be 5 files: driver.ml, lexer.mll, parser.mly, types.mli and types.ml, as described in the programming assignments.
- There should be a test file tests.ml that should test the types.ml file. So you should be writing expressions directly into the core language and evaluate them, and test if the result is expected, and also expressions written in the surface language and test that they convert to the right core language constructs (or that they evaluate to the right value).
- You should create a .gitignore file that lists the "derived" files that should not be part of commits (e.g. the parser.ml and parser.mli files that are automatically generated). You can look at the .gitignore file that the assignments project has for inspiration.
- Your language should at the very least contain numbers. You can distinguish between integers and floats, or even have fractions, or just treat them all as one type. One should be able to perform basic arithmetic operations on numbers. You can add more operations if you wish.
- Your language should at the very least contain booleans. It should have conditionals, logical operations, and comparison operators, including equality.
- Remember to try to keep the core language as small as possible. Each time you want to add a new feature, think whether you can do it simply in the surface language, without changing the core language.

You can choose to implement one of the following two types of languages. These sections describe features of your language, but give you plenty of flexibility to evolve the language from that starting point.

Functional Programming

This language follows OCAML closely. You can choose to use a Racket-like syntax instead. In general you can use whatever syntax you want externally. For instance you don't have to call the let statements "let", they can be "define" or whatever else.

Basic expectations:

- Function closures should be primitive values, usable anywhere where a value is expected.
- There should be no mutation.
- It should contain tuples and lists as primitive structures, along with the necessary primitives to work with them.

- It should have let-in statements for creating new scopes.
- It should have top-level let statements.
- All functions should be automatically ready for recursion, i.e. you can call the function from within its body.
- The language should have type-checking. Function arguments should have their types specified when they are defined.

Bonus items. You can choose to implement one or more. The more you do the better.

- Add a racket-`cond` feature (including an `else` as a synonym for true).
- Add the ability to define type aliases.
- Add pattern-matching.
- Make sure that function closures only store the variables they need from the current environment, instead of the whole environment.
- Implement type-inference and have function argument types inferred rather than specified.
- (harder) Ensure that the type-inference generalizes functions properly, i.e. that a function with type `'a -> 'a` will be able to be used in the rest of the file with many differents types `'a`, and it would not commit itself to the first occurence. This item will make more sense only after you have implemented basic type-inference.

Class-based Object-Oriented Imperative

This language would be a bit close to Java, except for no type-checking.

Basic expectations:

- Scope should be specified with some version of braces (in addition to function bodies).
- There should be a way to declare new variables.
- There should be an assignment operation that can change the value of variables (variable mutation).
- In order for mutation to work, you will need to use a "store" in addition to the environment.
- Since variable mutation is present, you need to be very clear about the behavior you expect for the parameters that are passed to a function. Make sure that changing the value of those parameters in the function does not actually change the entry that was provided by the caller.
- The main construct in your language would be a "class".
  - A class contains a "constructor" that given some inputs returns a new object of that class.
  - It also contains the object methods.
  - There would be a way to call the constructor to create new "objects". Objects need to hold a list of the initialized variables and also to know what class they belong to.

  - Within an object method, there should be a keyword, this or self or something else, used to refer to the object itself. It needs to be part of the environment when we evaluate an object's method call.

- Your language should support while and for loops.
- You will need to make a distinction between statements (assignment, for, while, if-then-else) and expressions.
  - Your program will be a series of function and class definitions. Function/Method definitions will be a series of statements.
  - Expressions are expected at specific places.
  - Functions and methods should use an explicit return like construct for returning a result. You will have to decide how your constructors would behave.
- Object variables should be only accessible from within an object's methods. Outside entities can only call an object's methods.

Bonus items. You can choose to implement one or more. The more you do the better.

- Add support for inheritance. This may include some sort of super keyword to access the parent class. Subclasses should have access to the variables defined by the superclass.
- Allow class methods and class variables.
- Offer a separation of methods and/or object variables into private/public.
- Add a switch/case construct. Make sure to be clear about fall-through behavior. You may also include a default construct.
- (harder) Add some basic "garbage collection" mechanism: Based on the current environment it determines which parts of the "store" are no longer needed, and shrinks the store accordingly.