Taspia Khan
CS51 Final Project
May 3rd, 2023

The CS51 Final Project consisted of implementing a small subset of the Ocaml language, called Miniml. This language consisted of a type expression that encompassed different atomic types such as integers and bools and other types of constructs available in the Ocaml language, such as functions, conditionals, let statements, and even definitions of recursive functions. The Miniml language evaluates expressions in a way that incorporates substitution semantics, which is a method for substituting certain expressions for free variables in another expression. It can also evaluate expressions in an dynamically-scoped environment. As a part of the project, we had to incorporate our own extensions to try and extend the Miniml language. The extensions that I incorporated consisted of adding more functionality to the language with the addition of two different operators, and implementing an additional form of evaluation that evaluates expressions in a lexically-scoped environment.

The two operators I added to the Miniml language were a GreaterThan operator and a Divide operator. This would let you divide two integers in the Miniml language and then compare either integers or bools with the greater than symbol to output a bool. For example, the expression "Binop(Divide, Num(5), Num(5))" would output an expression of Num(1). Similarly, an expression of "Binop(GreaterThan, Num(5), Num(5))" would output an expression of Bool(false) because 5 is not greater than 5. I implemented this by tweaking the type definition of an expression and then adding different match cases to my evaluation functions. I also had to add to the parser to make sure the REPL for the Miniml language worked to encompass this added functionality.

My second extension was a little trickier to implement. Dynamically-scoped semantics refers to ways of evaluating expressions, specifically functions, in the environment in which the function is applied while lexically-scoped semantics refers to ways of evaluating functions in the environment in which the function is defined. The two evaluation functions for a lexical versus a dynamic scope were pretty similar, but the lexical evaluation function differed for evaluations of functions, function applications, and recursive let statements. For an evaluation of a function in a lexically-scoped environment, you want to return a closure, which returns the function along with the environment in which the function was defined. Then, during function application, you evaluate it within the environment returned by the closure. For a recursive let, the process is more complex. You first need to evaluate the definition of the let in an environment where the variable maps to Unassigned, and then evaluate the body of the let in that environment, where instead of Unassigned, the variable maps to whatever result came from evaluating the definition. Using the functions from the environment module helped create this lexical evaluation function. So for an expression like "let x = 1 in let f = x + 5 in let x = 2 in f", the lexical evaluation would spit out 6 while the dynamic evaluation would spit out 7. Implementing these extensions helped improve my understanding of semantics in general, and gave me a better sense of what it takes to implement a functionality of a language.