Austin Williams | CS51 - MiniML Extensions | May 4, 2022

This document serves as a summary of the extensions integrated with this implementation of MiniML. The extensions vary from large to small and include the interpreter evaluating expressions under a lexical environment semantics, adding floats to the 'expr' ADT, abstracting away similarities within the different evaluation functions, and changing the interpreter's output format. This implementation of MiniML thoroughly unit tests all functions, making sure to test the evaluation of expressions that evaluate to different values under different semantics. The remainder of this document will briefly touch on each extension.

Extending the interpreter to evaluate expressions under the lexical environment semantics meant writing a new evaluation function. Many of the evaluation semantics were the same when compared to substitution and dynamic environment semantics. Functions are evaluated differently as functions evaluate to closures under the lexical environment semantics so that when the function is applied, it's applied in the context of the environment in place at time of definition, which agrees with the semantics used in OCaml. The evaluation of a let expression and application expression also slightly differ from the other semantics in that the let expression's definition may evaluate to a closure and the first expression of the application should evaluate to a closure. Evaluating a Letrec expression differs much under this semantics. As per the textbook's description of Letrec under the lexical environment semantics, by first extending the incoming environment with a binding of the Letrec variable, call it 'f', to 'Unassigned', and evaluating the expression's definition to some value 'v_d', we can mutate what 'f' maps to in our extended environment to 'v_d'. Assuming instances of 'f' are in the expression's definition, these instances will also map to 'v_d', so that when we evaluate the expression's body, assuming there's an instance of 'f' in the body which itself has an instance of 'f', the inner instance of 'f' will be mapped to 'v_d' so that evaluation doesn't get stuck.

Adding floats as atomic types of MiniML required adding to the expr ADT, unop ADT, and binop ADT, and editing the lexical analyzer and parser so that floats and expressions with floats could be parsed and interpreted by the interpreter. Adding to these ADTs required updating nearly all functions that power the interpreter as most functions within MiniML deconstruct expressions to determine how to evaluate. Not only was there a new atomic type to match with, but new larger expressions to match with (unop and binop expressions) with new code for how all these should be evaluated. The evaluation logic was checked by adding more unit tests to the tests.ml file.

All three evaluation functions have many commonalities, so it made sense to abstract away these commonalities using a general evaluation function that may get called from one of eval_s, eval_d, or eval_l being called. It made sense to abstract away redundant computation from the evaluation of unop, binop, and conditional expressions. This common evaluation function, called 'eval_comm', also takes one of the specific evaluation functions to properly maintain the evaluation path within whatever semantics is being used.

Lastly, the interpreter's output is augmented in that the output is split into four sections: (1) A - Abstract Syntax, (2) S - Substitution Semantics, (3) D - Dynamic Environment Semantics, and (4) L - Lexical Environment Semantics. Having the abstract syntax as part of the output is nice for testing purposes and for someone unfamiliar with the abstract syntax trees of MiniML expressions. Having every inputted expression evaluated under the three considered semantics is nice especially when an expression evaluates differently under different semantics.