Brandon Foster May 9th, 2019

Hunter College Functional Programming in Ocaml

OCamlLisp

OCamlLisp, or OCaml Common Lisp Interpreter, is an interpreter that is built with OCaml to evaluate basic Lisp-code. The program must be provided with the Lisp-code file as the first and only argument to run. If the Lisp-code file contains syntax that is currently supported by OCamlLisp, the correct output from the evaluation will be returned. A detailed error message will be printed to output if there is a syntax error or an evaluation error. The data-types that OCamlLisp currently supports are integer, ratio, char, string, and symbol. A symbol can refer to a function or another data-type and the other data-types are self-explanatory. The "main.ml" file holds the main function and "commonLisp.ml" contains the types for Common Lisp objects and helper functions for them.

The development of this project required a comprehensive study about the structure of an interpreter. From the research, an interpreter is built by following several steps. The first step is to incorporate a lexical analyzer, or a scanner, to read a stream of characters. After reading the characters, the scanner produce tokens from that stream that will be used for the second step. Each token represents a concatenation of characters that is a valid syntax for the interpreter's language. The second step involves reading the stream of tokens to a syntax analyzer, or parser. The parser generates parse tree based on the stream and every single token becomes a leaf in that parse tree. The parse tree is then structured into an abstract syntax tree. The abstract syntax tree is a different structuring from the parse tree that allows an easier and more optimal evaluation. The abstract syntax is finally evaluated from the root to produce the appropriate output for that language. The project follows this structure to smoothly create the Lisp interpreter.

Several OCaml libraries were used during the development. These libraries are OCamlLex and Menhir. OCamlLex is used as the lexer generator for the project. The standard for OCamlLex is to always provide a "lexer.mll" file to hold the lexical analyzer's code for the language. Multiple rules were created to handle certain tokens for Common Lisp. These rules are: read, does a basic read of each character where it either branches out to another specific rule or generates a token for a white space, a parenthesis, or a general lisp's token; comment, skips every other character for that line; balanced_comment, skips every other character until it closes; make_string, accumulates each character for a string; and sharp_sign, generates a token that is used to specific a data-type, such as an integer or ratio at a certain base or a single character. Menhir is used as the parser generator for the project. A menhir file is usually created as "parser.mly" to hold the grammar that follows the syntax to parse the tokens. Some OCaml helper functions were also used during this step to help with checking for the proper data-type of a Lisp's token. If the syntax rules hold for the token stream, the tokens are properly reformed into an abstract syntax tree which in this case are s-expressions. The OCamlLex and Menhir compiler will produce the two files, "lexer.ml" and "parser.ml", that will be used by the OCaml compiler to build the program. An s-expression that represents the root of the Lisp's code is returned from this process and will be used for the evaluation.

The evaluation of OCamlLisp is handled in the "eval.ml" file. The format of the s-expression that is given by the parser has the left node as the Lisp's object to evaluate then the right node is the next s-expression that follows this same method. This results is a tree where the left child represents the block of Lisp's code to evaluate and the right is the remaining blocks. This format is used for the individual block of Lisp's code as well. The first step of the evaluation begins by sending the s-expression as the first argument to Eval.run_program. The second step of the process calls the eval function from with and passes an empty environment as the first argument and the expression passed as the second argument then returns unit. Before explaining the nature of the eval function, for every function in "eval.ml" that handles some form of evaluation, an environment must be passed as the first parameter. These functions are always prefixed with either "eval" or "lisp". An environment holds global and local variable bindings and the purpose of an environment being passed is to keep track of the global and local bindings made during the evaluation. Each of these eval functions must return a value of type EvalReturn, which is a 2-tuple pair of an environment and an eval_type. The reason why an environment must be returned is to update the global bindings of the current environment with the global bindings of the evaluated one. An eval_type must be returned as well to allow the evaluation to properly execute a certain operation based on value that the s-expression evaluated to. The third step evaluates the list of s-expressions in the eval function. This is done by calling eval_expr on the left child, but if the s-expression is a leaf, eval_expr is called on that leaf. After eval_expr is called, if the s-expression has a right child, that right child is recursively called with eval again, with an environment that is updated with the left child's evaluated environment. This results with the list being evaluated and the global bindings being passed on as well. The fourth step is the eval_expr function that takes the expression and attempts to evaluate it. If the expression is a node, it is assumed to be a function and eval_fun is performed. However, if the expression is a leaf, it will properly evaluate that leaf and return an appropriate eval_type value. For function evaluation, certain functions that require cycling through the remaining portions of the list of s-expressions for arguments will use the car and cdr functions. These four steps explain the evaluation process for OCamlLisp and the structure of this process allows for more Lisp functions and behaviors to be implemented in the future.

The difficult parts of this project that I have faced during the development was understanding OCamlLex, Menhir, and Lisp, as well as finding a proper way to utilize all three to build the interpreter. I have restarted from scratch many times because I felt that the lexical analyzer and the syntax analyzer was implemented incorrectly. I slowly gain a decent understanding of Lisp from chapter 2 and 3 of the Common Lisp's Standard Documentation, but I still believe that I am missing the bigger picture. Time is another issue with this project and because the time was limited, I had to decide on basic functionalities to handle, such as the PRINT and SETQ function, the DOTIMES loop, and basic arithmetic. One good aspect that I can take from this is that I now have a better insight in building interpreters, especially in a functional programming way.