FP in OCaml Report

A Short and Sweet Dive
into
Interpreters with OCaml

Jul 7th 2019

Written by:

Khinshan Khan
Using emacs org-mode, latex, html (for code block minting), as well as embed languages.

Enjoy.

1 Introduction

This project is a Lisp interpreter implemented in OCaml to evaluate basic Lisp code. This Lisp resembles ELisp the most closely, as I'm more familiar with it. Although this means I'll forgo rational types, a way to represent any rational number without precision errors. However, this is not completely ELisp, as it never was meant to be. This project more so explored how to utilize OCaml to evaluate Lisp code. It exposed me to ocamllex, menhir, sexp, and a deeper appreciation for pattern matching.

2 Structure Overview

Currently the program structure looks like

```
├── Makefile
├── sample
└── src
    ├── error
    ├── eval
    ├── main.ml
    ├── reader
    └── syntax
```

Where the `Makefile` makes it easy to compile and clean with instructions, `sample` has a view well written lisp files which one can test with, and `src` holds the OCaml code. In `src`, `main.ml` holds my "main" function (function that runs the entire program). This calls the reader function in the next directory. `reader` holds my parser and lexer files, as well as my sexp atom types (will discuss later) and the actual function that reads from a lexbuffer and then evaluates using the next folder. `eval` has the functions which evaluate sexps given from the reader. `syntax` has functions I considered to be not really evaluate, but to deals with typing matching of my custom type t. `error` holds functions which makes my error handling a bit cleaner.

3 Parsing and Lexing

The most critical and difficult part of the project is probably the lexer and parser. Making this well defined required me to understand types a bit more deeply, as it would determine how much easier or harder (and like spaghetti code) my project would have. Lisp is a nice language to use as it consists of S-expressions (symbolic expressions, I will use the shorthand sexp). I used OCamllex and Menhir. I provide `lexer.mll` for OCamllex, and it holds the lexical analyzer code. Then for Menhir, I provide `parser.mly` which holds the grammar rules to parse the tokens provided by the lexer. I decided to take a very list oriented approach. I shall gloss over various things such as `prog` in `parser.mly`. The important sexp I defined was either an atom or cons. A cons would simply be a list of sexp, delimited by surrounding `()`. An atom is much more complex, it's a single "unit" of either an int, float, bool, string, char, or a sexp list. To be exact, the type is

```
type t =
  | Atom of [ `Int of int | `Float of float | `Bool of bool | `String of string
            | `Char of char | `Sym of string | `Tuple of t list]
  | Cons of t list
```

where units are delimited by at least one space (extra non important spaces are ignored). I suppose another important distinction is that symbols are strings well, so later I can just pattern match something like "+" to `( + )`. Just making it all connect and stay clean proved to be hard.

4 Taking in input

I actually had a little difficulty here too, because I didnt really understand the "Lexing" methods and the examples seemed to use the `Batteries` module, but never specified it, so I have no idea where `Lexing.from_input` was coming from. However, the fix for that was simply creating a lexbuffer based on stdin or the file opened throwing it into a surrounding `try with` block. However, the form

```
try
with _ ->
  exit 0
```

created some sort of blocking when it was separated from main function. But this was the only way to make it modular in my `main.ml` to support interactive and non interactive. However, the fix was rather easy: simple throw it into a module and calling that instead. Making the program loop proved to be much harder, so I disregarded it, hence the REP ~~L~~.

5 Evaluate

The type makes it rather easy to evaluate Lisp code, as it follows prefix notation. So to evaluate a single cons, the first argument in the list will be what is applied to the rest of the atoms of the list. Then, if another cons is encounter within the cons, simply evaluate that cons as well. Eventually, everything will end up as a single atom. This was a bit tricky, and I had to use a mutually recursive function:

```
let rec eval_sexp = function
  | Sexp.Cons t ->
    begin
      match t with
      | (Sexp.Atom h)::t ->
        let a = List.map atomizer t in
        sym_ops a h
      | _ -> Error._failwith "cons fail"
    end
  | _ -> Error._failwith "sexp fail"
and atomizer t =
  match t with
  | Sexp.Atom t -> t
  | _ -> eval_sexp t
```

However, this made everything a simpler. Now, I only had to well define `sym_ops` and never touch anything in a different form, ie I only had to work with the form `(function argument-list)`. In `sym_ops`, I matched the function with what the function should do with the list.

6 Variables and Functions

I didn't get to implement this, but it seems easy enough, given my current structure. Theoretically, I only need to pass around an associative list of symbol and its value. `setq` would be as simple as storing the symbol and its value in said list, and `sym_ops` would first look through this list and then through the "builtins". Since OCaml supports partial applications, I can just set a function name to its definition which is evaluated without any arguments, and so it'll wait for its arguments (actually, this may prove to be a little tough).

Author: Khinshan Khan
Created: 2019-07-07 Sun 19:37
Validate