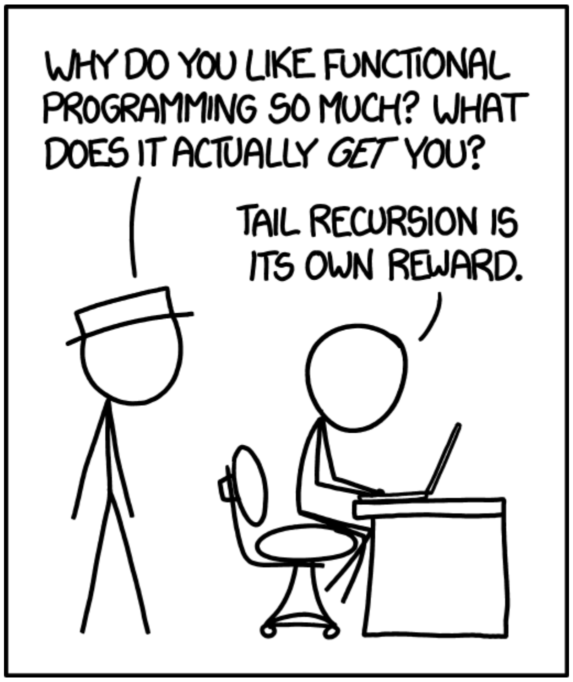
EXAMINING CLOJURE BNF GRAMMARS AND ITS FUNCTIONAL PROGRAMMING PRACTICES

Principles and Practices in Programming Languages
CSCI 3155 Spring 2018

Lubrano, Jason; Smith, Brad

April 30, 2018


WHY DO YOU LIKE FUNCTIONAL PROGRAMMING SO MUCH? WHAT DOES IT ACTUALLY *GET* YOU?
TAIL RECURSION IS ITS OWN REWARD.

1 ABSTRACT

The purpose of this project is to utilize our gained knowledge of BNF grammars where we create a parser the Clojure programming language. Since our last lab was going over parsing, we decided to write a parser using these grammar rules. Being that we are doing the project in Clojure on a short time period, we will unfortunately not be able to get through the entirety of the language [1]. Because the syntax of Clojure is different than Scala, language rules will vary differently than what we have done in labs.

2 GETTING STARTED

We chose Clojure for a number of reasons. The first of which is that its widely popular in the industry. Unlike graphing and statistic software, Clojure allows the developer freedom from having bulky menus and presets. With just a single change of a line, correlations between axises can completely change an observers thoughts of data sets. Other interesting things of Clojure are its power. It has concurrent, its dynamic, its always compiled, it has the power of Lisp, and most importantly its a functional programming language. With Nightcode, we were able to make a parser which showed us how Clojure worked.

3 CLOJURE BASICS

3.1 Clojure Syntax: Whats up with all those parentheses?

Looking at a basic program in Clojure you will notice there are alot of parentheses. In Clojure when doing any sort of declaration or evaluation it must be enclosed in parentheses. For any use of a function or operator in Clojure we must enclose it in parentheses, with the function name or operator followed by the a space and the arguments. Lets look at a simple example of variable declaration and an arithmetic operation on those variables. In this example we can see we use def followed by the variable name and the what it will be assigned to. For the arithmetic operation start with the arithmetic symbol that we want the variables to be worked on and then followed by a space and the 1st argument and once again another space followed by the 2nd argument.

```
(def x 4)
(def y 2)
(+ x y)
```

3.2 Declaring Functions in Clojure

All function declarations within Clojure start with defn followed by the functions name and then square brackets with the function parameters. Then the function body. Example of a simple function with in Clojure.

[1] Hopefully we can get through it, if so then we need to edit this footnote

```
(defn Example2 [x] (+ 3 x))
```

3.3 Core Concepts of PPL And Clojure

In Clojure we can see that there are many similarities to Scala. Clojure allows for Higher Order functions, anonymous functions, and abstract data types.
To Start lets look at how Clojure deals with functions. Much like Scala we can look at functions as values. Therefore we create Higher Order functions by passing a function as an argument into a function.

3.3.1 Anonymous Functions

When declaring a an anonymous function in Clojure we need to include an fn before declaring the function parameters and body.

```
  (defn Example []   ((fn [x] (* 2 x)) 2))
```

3.3.2 Higher-Order Functions

There many syntactical things about Clojure to point out in this code. First is the loop/recur combination. This is one of the ways we can produce recursive functions within Clojure. The loop is followed by brackets with parameters that will be looped upon in the loop statement or the recur statement. In addition when declaring a function as an argument we do not need to enter and parameters within or parentheses around it until it is need to be evaluated. With in this function we have three Clojure methods being used: "empty?" which returns true if a list is empty and false otherwise, "conj" which prepends a list, and "rest" which returns the all the elements in the list after the first element

```
 (defn Map[f x]
    (loop [xn x
           ff f
           acc `()]
      (if (empty? xn)
        acc
        (recur (rest xn) f (concat acc (list (ff (first xn))))))))
```

In addition we can create Abstract data type with in Clojure. This is due to Conjure being a marco system which allows for an extension of the complier. Through the use of our own defined marcos we can create ADTs.

4 CREATING THE FIRST CLOJURE PARSER

To start, we wanted to make a very simple operation. We came up with a addition rule. `S` is our starting rule. `AB` indicates that we use two separate types to make a binary expression. We use `A` and `B` to indicate the second and third rule respectively.

So we create a grammar rule that is free of ambiguity and we run into our first syntax problem. Because Clojure is very arithmatic intensive, it wont recognize the plus symbol to join an `A` and `B`. Therefore, we have to include it in the actual `A` and `B` expressions respectively. And the Grammar of understanding the grammar, having `a` and `b` in the single quotes denotes that is the character we will have. We also want to have support for an empty string.

$$\begin{aligned} \texttt{S} &::= \texttt{A*} \\ \texttt{A} &::= \texttt{'a'} \mid \texttt{B} \\ \texttt{B} &::= \texttt{'b'} \mid \epsilon \end{aligned}$$

Luckily for us, to create a parser in Clojure is easier than we have previously thought. In the words of a great TA: *"Just follow the grammar"*. So we did. The grammar rules to programming the actual parser are very similar to the actual syntax of the language.

```
(def parserab
  (insta/parser
    "S = A*
    A = 'a'| B
    B = 'b'| Epsilon"))

plmini.core=> (parserab "a")
[:S [:A "a"]]

plmini.core=> (parserab "b")
[:S [:A [:B "b"]]]

plmini.core=> (parserab "ab")
[:S [:A "a"] [:A [:B "b"]]]

plmini.core=> (parserab "abba")
[:S [:A "a"] [:A [:B "b"]] [:A [:B "b"]] [:A "a"]]

plmini.core=> (parserab "baba")
[:S [:A [:B "b"]] [:A "a"] [:A [:B "b"]] [:A "a"]]
```

Here it is creating the actual tree for us also. It is showing the steps from `S`, to `A`, and `B` calling the right grammar respectively. I tried a couple of letters to see if they would work or not to follow our rules. Sure enough, we can have completely unambiguous grammar using Clojure grammar rules that involve any character in any order of 'a' and 'b'.

5 GETTING MORE BANG FOR YOUR PARSER

With our Clojure parser, there is a lot more we can add to it such as grouping, concatenation, comments, compaction, recursion, and even letting us know whether or not our grammar is ambiguous.

5.1 Compaction

I included this bit to make our program seem easier than what is actually going on. We are just adding `a` and `b` in any order, but we can make this grammar a bit easier to understand and completely eliminate ambiguity.

```
(def parseabrcompact
  (insta/parser
      "S = ('a' | 'b')*"))

plmini.core=> (parserabcompact "abba")
[:S "a" "b" "b" "a"]
```

5.2 Right Recursive

Right recursion on the other hand, can save us space. So our characters are just going to be `a` to make it easier on us. With right recursion, the parser is typing in an `a`, going to the next character (which is an `a` and continuing on with the list until it reaches the final epsilon. By calling the recursion letter until we reach the end, we aren't building a stack [2] and we are saving space.

```
(def parserabright
  (insta/parser
      "S = 'a' S | Epsilon"))

plmini.core=> (parserabright "aaaa")
[:S "a"
  [:S "a"
    [:S "a"
      [:S "a"
        [:S]]]]]
```

5.3 Left Recursion

Left recursion on the other hand will continue until we hit epsilon, and then recursively work our way back. By doing our work this way, we aren't calling any instances or objects of characters until we are done. We are running through our functions until we finish. It does

[2]Hey brad can you check to make sure this is ok?— I think we are still building a stack with the recursive calls because it has to remember the context in which the recursion was called

make our program take up more stack and use more space, but it is easier to walk through (I find at least).

```
(def parserableft
  (insta/parser
      "S = S 'a' | Epsilon"))

plmini.core=> (parserableft "aaaa")
[:S
  [:S
    [:S
      [:S
        [:S]
      "a"]
    "a"]
  "a"]
"a"]
```

5.4 Ambiguity

Ambiguity is a huge mistake when creating BNF grammars. Fortunately for us, Clojure has the capability of telling us whenever we have an ambiguous grammar. Note, the following code is intended to have ambiguity for purposes of this subsection.

```
(def parserambiguous
  (insta/parser
    "S = A A
     A = 'a'*"))

plmini.core=> (->>(insta/parses parserambiguous "aaaa"))
([:S [:A "a"] [:A "a" "a" "a"]]
[:S [:A "a" "a" "a" "a"] [:A]]
[:S [:A "a" "a"] [:A "a" "a"]]
[:S [:A "a" "a" "a"] [:A "a"]]
[:S [:A] [:A "a" "a" "a" "a"]])
```

6 Arithmetic Parsing

Now that we have the basics of parsing we can create a simple parser for Clojure Arithmetic. With Clojure Arithmetic we needed to include non-terminals for the arithmetic symbols, space and numbers. We allowed for optionality of spaces between the parentheses because they are allowed but not necessary for writing the expression. With this we needed to include spaces for between the digits and the symbols because Clojure requires that. In addition we allowed for more than one number to follow the first number as in Clojure to perform arithmetic on multiple numbers at once.

```
(def qq
    (insta/parser
      "S = '(' SPACES* ARITH SPACES DIGIT (SPACES DIGIT)+ SPACES* ')';
      ARITH = '+' | '-' | '*' | '/';
      SPACES = ' '*;
      DIGIT = #'[0-9]+'"))

pplmin.core=> (qq "(/ 33 3)")
[:S "(" [:ARITH "/"] [:SPACES " "] [:DIGIT "33"] [:SPACES " "] [:DIGIT "3"] ")
    "]
```

7 FUNCTIONALITY

Clojure is a very powerful functional programming language. One of its many features is that its a functional programming language. We are able to have full control over our recursive functions.

7.1 Basic Addition Function

Lets suppose we want to make a basic function to add two numbers together. We know we need a parameter name `add`, and to take in two numbers, `n1` and `n2`. And we need to return the sum of the two values. Similar to our first lab, this example is very simple.

```
(defn add [n1 n2]
  (+ n1 n2))

plmini.core=> (add 2 3)
5
```

And there is a basic addition function. Lets grow on this idea and see how far we can actually go with our functional programming practices.

7.2 Counting by 5s... anonymously

Suppose we want to count by 5's now. Sure we can make a function like in the basic addition, but where is the fun in that? We want meat. We want to be able to call multiple functions, have variable declarations, and anonymous functions! Lets start off with two functions, `add5` and `adder`. To make it harder, `add5` takes in zero parameters and `adder` takes in one. Well how does a function add by 5 and only take in one number for two functions? We know that in functional programming our functions are values. We spent an entire semester going over it. Functions are values. It is some type `A` and it returns some type `B` such that `A=>B`.
The first function `add5` calls `adder` and 5. The 5 indicates that we are counting by 5. If we were to count by 3 we would change the 5 to 3. The second function `adder` is a bit more complex but it is where the magic happens. A hidden power of `adder` is that it has an

anonymous function in its body. With this known, the anonymous function is what takes in the value which we are adding 5 to.

```
(defn adder[e1]
  (let [y e1]
    (fn [z] (+ y z))))
(def add5
  (adder 5))

plmini.core=> (add5 15)
20
```

We combined multiple function calls, anonymous functions, and variable declarations all in about 6 lines. So what about data structures? What if we wanted to add5 over an entire list? We will get to those shortly. First, an intermission for data structures.

7.3 Data Structures, what you already know

Data structures are an important part of any programming language. For the sake of length, we have mapping, vectors, and list. To initialize them and print them out is easy.

```
(let [my-vector [1 2 3 4]
      my-map {:m1 "foo"}
      my-list (list 2 4 6 8)]
  (list
    (print (str "\n" my-map "\n" my-vector "\n" my-list "\n"))))

plmini.core=>
{:m1 "foo"}
[1 2 3 4]
(2 4 6 8)
```

With data structures, and functions in mind, the question begs, what if we were to run an arithmetic operation on one of them.

7.4 BRB Bad Recursion BRB

Finally, after going through a brief overview of Clojure's functionality in a couple paragraphs, we are now prepped to create a `foldleft` and `foldright` tail recursive function. For those who do not know what either of these are, `foldleft` takes an list and multiplies the head of the list, to its next element and so forth until the tail is reached. When the tail is reached, it returns a value of the total multiplied numbers. `foldright` on the other hand starts at the tail and works its way forward. In true fashion, let us create a definition `mul` which takes in the head element and the next element of the list, multiplies them, and then returns their product. Conceptually wise, it seems easy enough to do. Here is the code:

```
(defn mul [n1 n2]
  (* n1 n2))
```

```

(defn foldleft
  [ns]
  (if (empty? ns)
    1
    (mul (first ns)
       (foldleft (rest ns)))))

(defn foldright
  [ns]
  (if (empty? ns)
    1
    (mul (last ns)
       (foldright (butlast ns)))))


plmini.core=> (foldright [2 6 4 9 8])
3456
plmini.core=> (foldleft [2 6 4 9 8])
3456
```

Lets walk through the code. The first function `mul` is just like the basic addition. The next two functions `foldleft` and `foldright` are a bit more in depth. For `foldleft` we are taking in a list of `ints`. Due to clojure's automatic type checking, if we threw in another type we would get an error. First we are checking to see if the list is empty. If it is, then we return 1. If it isn't, then we want to call `mul` on our first value and the next item of the list. From there we are tail recursively calling the fold function again with all but the last one. Fold right works in a very similar way, but we are gobbling up the first `n` and going through until we hit the end.

8 GITHUB Link

https://github.com/jasonlubrano/CSCI3155-JasonLubrano/tree/master/JL-BS-PLMiniProj/plmini

9 CITATIONS

http://www.try-clojure.org/
https://github.com/Engelberg/instaparse
https://clojuredocs.org/
https://clojure.org/about/functional_programming
https://stackoverflow.com
https://stackexchange.com
https://github.com
https://www.tutorialspoint.com/clojure
https://clojure.org/api/cheatsheet