CC2016
Project Second Phase

JCF implementations

For this assignment we used the following JCF:

1. HashMaps:

Here hash maps were used extensively, for example they were used in the Scanner, to construct the different finite state machines.

These maps followed the structure of:

HashMap<Integer, Transition> fsm

The idea is that the string from lisp is introduced to the code, and it labels each character with an integer, if the integer matches the character in the FSM then it proceeds to the next state. The process of the next state is used with Transition, which is a class structure on itself, which is pretty much an indicator function that either returns an acceptable symbol or returns nothing. If it returns an acceptable symbol it moves to the next state of the machine.

Most of these finite state machines had only one state structure, like arithmetic operators.

However, the finite state machines for Numbers had multiple states due to the situation that the number is not an integer, for such cases the finite state machine keeps running until it finds an empty space " ".

A similar idea is used if the machine finds a Identifier, here identifier is used to say words typed in the code. This is really important in defun, because defun needs an identifier so that the code knows which defun is it referring to.

We used HashMaps instead of other maps types, mainly due to performances reasons, because using HashMaps doesn't order the identifier, and because the logic on how it organizes the token is handle in the interpreter, which as we will discuss, we used an List<> to organize the tokens.

Other instances of HashMaps are found in:

InterpreterLisp with the method: getPlaceHolderCache, this variable takes the value the Map<String, Object>, this is purely to assign the parameters used in the defun. So when the user inserts the values for the defun, it automatically assigns the values to the defun to interpret it with java.

2. Another JCF used is List<>:

List<> was very important in the interpreter, because for simplicity reasons, we take the ArrayList from the tokenizer in the Scanner, then we just take it as a List, and we interpret each token one by one. Even So this is evidently a rather slow method, the reason is that a faster approach like a Syntax Tree is something we didn't have the time to implement, so instead we approach the problem with a very direct approach.

3. ENUM class

We used ENUM to categorize the tokens, so that each token in the prefix would be a finite state machine with a respective TOKEN type. This is purely to make the identification of the token much easier to implement in the interpreter.

4. Factory Design Method

Evenso is not a JCF in itself, it uses various JFC to work. The idea is that we did a polymorphism on each type of operation, whether they are arithmetic, conditional, or functional. It assigns a character value to the operation and a method that will run for that character. Then this factory is an instance in the InterpreterLisp, to compute the token with its corresponding java process.

5. Interface

We used an interface in the InterpreterLisp, although quite unnecessary, the idea is to define the methods that the interpreter would require to run. But given that the approach we used for this interpreter is not very sophisticated, it is rather a pointless interface because it uses some internal methods to run the entire class.

6. Logger Manager

Although not a JCF at all, at first we thought that it would be a good idea that all the classes would have love logger, but we decided that for better debugger, we would have a logger manager class that would log all the bugs into a predefined file. But due to time we didn't

implement the logger manager in every class, in the end it is mostly used in the Scanner and the Factory.

References:

GeeksforGeeks. (n.d.). *Collections in Java - GeeksforGeeks*. https://www.geeksforgeeks.org/collections-in-java-2/