Making a (small) Lisp Interpreter (in Python) : A Report

Katelynn Rainey
New Mexico Institute of Mining and Technology
Department of Computer Science
katelynn.rainey@student.nmt.edu

1. FINAL STATUS TABLE

Lisp Construct	Status (Done, partially done, not)
Variable Reference	Done
Constant Literal	Done
Quotation	Done
Conditional	Done
Variable Definition	Done
Function Calls	Done
Function Definitions	Done
The arithmetic operators, car and cdr, cons, sqrt exp	Partially done, cons has some spacing issues in output but works
><==<=>=!=not or and	Done

Extra credit was not attempted

2. The implementation/details/how it relates to class topics

First, in order for expressions to be evaluated, the input expression is first split into an individual character list and then fed into the build() function of the program to build a new list of sublists that acts as an abstract syntax tree, which mirrors the nested structure of the input expression. To do this, recursion was used to go through the list of characters and build new sublists, where once a new ")" is found, that sublist returns until a list of sublists has been formed. Thus, leading to something like this to be constructed:

['+', '3', '4'].

Then, the new "abstract syntax tree" is sent to evaluate() where the first argument, ("+" in this case) is evaluated as a function call and the subsequent arguments are evaluated within the context of the function call. Thus, again utilizing the concept of recursion, which makes use of the "stack" - where as we step through the abstract syntax tree

expression and evaluate() (i.e pushing evaluate() calls onto the stack) we eventually get to the point where evaluate() is no longer being called. Then from that step, each evaluation is then popped off the stack and the function operation is performed (as each call to evaluate() returns). This is also the function where the lisp constructs are handled.

VARIABLE DEFINITION

To handle variable definition, if "define" is recognized as the first argument in the abstract syntax tree representation (a list), the second argument is then treated as the variable name. Then, the variable name key is then added to the global environment (or global dictionary since an environment maps names to values) with the value of the evaluation of the third argument. So define r 10 will put r into the global environment as a key and 10 will be the value (10 will return as itself during evaluation). Then once the definition process is complete, the name of the variable is returned and printed in the terminal.

VARIABLE REFERENCE

If a variable has been defined, evaluate() will check if the argument 'buildparse' is a string rather than a list using isinstance(). Then if it is, the find method specified in the environment class will statically check to see if the variable has been defined (the key: value pair) in each environment instance one at a time. If it has been defined, the variable/key value will then be returned to the terminal.

To do this, the function the_env() had to be created where a class Env() was made featuring a constructor and the find() method, where if the variable is not in the current environment, it goes and checks the next nested environment. This concept looks like the contour diagrams shown in class. Then the global environment was set to genv = the_env() and subsequently set equal to operationsdict, where operationsdict is

where evaluate() looks for dictionary keys and adds them. This was also done with the help of following Peter Norvig's “(How to Write a (Lisp) Interpreter (in Python)) “ tutorial, where he goes over these concepts.

CONSTANT LITERALS

To return constant literals as themselves, evaluate() will check if the abstract syntax tree representation is a string, then if it’s either T or NIL and will return them . Then for numbers, they will be returned when the conditional statement elif not isinstance(buildparse, list): evaluates to true. Any other strings not in an expression and not in the global environment will cause an error, and the message “Syntax Error” will be shown.

QUOTATIONS

To handle quotations “ ‘ “, additional handling was added to build() and exptolist() for handling. So if ‘(the cat in the hat) is input it will return (the cat in the hat). Variable assignments can also be made to quotations, and when that happens the string after the quotation is added to the strchecklist, where evaluate will check in when it attempts to evaluate the string. Then the assignment will be made. The string (a representation of an atom) is also added to the strchecklist if cons, car, or cdr are called.

ASSIGNMENT

For assignment, if “set!” is recognized as the first argument in the abstract syntax tree representation, a lookup in the global dictionary will be performed to see if the second argument var has been defined. If not, an error message will show. If it is, the third list argument will be evaluated and the specified variable’s value will be changed in the global environment (dictionary).

FUNCTION CALLS

For function calls, if none of the other conditionals in evaluate() are satisfied, then the function call handling will trigger. There, it will evaluate the functional call specified by finding the key in the global environment

where the function is defined. Here, this is where the operator library was used to define the integer arithmetic operations (other than division) and the logical relational operations. Otherwise, lambda functions were used to make nameless functions that would execute when the key value was returned (i.e for car, cons, cdr). Next, the arguments associated with the function call are also evaluated, and then those arguments are used in the function that was looked up. Then the full operation will return.

Additional handling was also put in for finding user-defined function calls so calls execute when parentheses are around the parameters. So (ADD (4 5)) will execute.

FUNCTION DEFINITIONS

For function definitions, if "defun" is recognized as the first argument, the "defun" block will perform similarly to define and create a key for the function to set the value to the evaluated arguments. However, before that - a new list will be created with "lambda" as the first element and the lists associated with the parameters and body of the function. Then when evaluate() is called, the "lambda" block will execute and a new procedure will be returned. Here, the Procedure class was defined to have a constructor for the procedure and a __call__ method to allow that instance of Procedure to behave like a function (so it can be called). Finally, after that, the function name is returned and can be used for calls.

CONDITIONALS

For conditionals, if "if" is the first argument in the abstract syntax tree representation, then a list is created where it is set equal to "(ifcond if evaluate(cond, operationsdict) else elsecond)" and then the result of that is then evaluated to get the final result.

CONCLUDING THOUGHTS/ETC.

To conclude, I thought this project was interesting and allowed me to get a better

handling of both python and how an interpreter works. However, I think a few of the functions can be adjusted, specifically “cons” as for now it just mainly does string manipulation to get the desired result - however with a few spacing issues. Overall, though, this project has made me interested in looking more into interpreters going forward.

REFERENCES

[1] Norvig, P. *(How to Write a (Lisp) Interpreter (in Python))*, *norvig.com*. Available at: https://www.norvig.com/lispy.html.