Project Assignment 5: Interpreter

Overview

This assignment furthers your understanding of how programming languages are interpreted. You will implement an interpreter for a simple language, “Its”, which is a subset of Python.

You will use a Python library, ast, to do the front end work of lexing and parsing the source code written in our language. You will write code that traverses the abstract syntax tree, produced by the ast library, and executes the code using an environmental data structure to keep track of the variable bindings and function definitions.

Aside from the course material, these links provide documentation on the ast library and especially the nodes in the abstract syntax tree you will work with.

https://docs.python.org/3/library/ast.html

https://greentreesnakes.readthedocs.io/en/latest/

Learning Goals

- To read and understand existing problem documentation.
- To write code that conforms to a specification.
- Gain experience developing code incrementally, testing the individual functions and then testing them in integration.
- Gain experience interfacing to a library of objects.
- Understand what the goal of this software is, and that testing it on data that it is ultimately designed to process is a required step in the development process.
- To understand the interplay between lexical specification, grammar, and execution.
- To further your understanding of how to work with an immutable data structure
- To understand how an environment data structure can implement frames of execution.

General Information

Read this entire document. If, after a careful reading, something seems ambiguous or unclear to you, then communicate to the course staff immediately. Start this assignment as soon as possible. Do not wait until the night the assignment is due to tell us you don’t understand something, as our ability to help you will be minimal. This sort of task always takes longer than you anticipate. After all, why do you think most software projects are late and over budget?

Reminder: Although sharing ideas with your peers is generally a good thing, copying partial or whole solutions, obtained from other students or elsewhere, is academic dishonesty. Do not share your code with your classmates, and do not use your classmates' code. If you are confused about what constitutes academic dishonesty you should re-read the course syllabus and policies. We assume you have read the course information in detail and by submitting this assignment you have provided your virtual signature in agreement with these policies.

Grading: You are responsible for submitting project assignments that run and are configured correctly. If your project submission does not follow these policies exactly you may receive a grade of zero for this assignment. Code that does not run will generally receive a score of zero points.

Policies:

- For all assignments, you must use the script provided to zip up your code. Your code file, class and method names must match exactly to the specification we provide. Refer to our coding style guide for naming and style conventions.
- The course staff are here to help you figure out errors (not solve them for you), BEFORE you submit your solution. When you submit your solution, be sure to remove all compilation errors from your project. Any compilation errors in your project will cause the auto-grader to fail your assignment, and you will receive a zero for your submission.

Overview of Programming Language Interpretation.

Review the slides from lectures about programming language structure and interpretation. To recap, a language design requires a lexical specification including a set of allowable symbols- the "alphabet" of the language. It also requires a grammar to define the syntax of the language. The grammar's productions are used to generate valid instances of the language (programs). Finally, an interpreter examines a parsed instance of the language and executes the statements. The source code is written in the designed language and the interpreter is written in the defining language.

The main elements of this process are: the front end, consisting of a lexer and a parser, and an interpreter, or evaluator. The focus of this project will be on how the evaluator works with an abstract syntax tree, the output of the parser, to execute the designed language.

The general outline of the process of executing source code is depicted by the following figure:

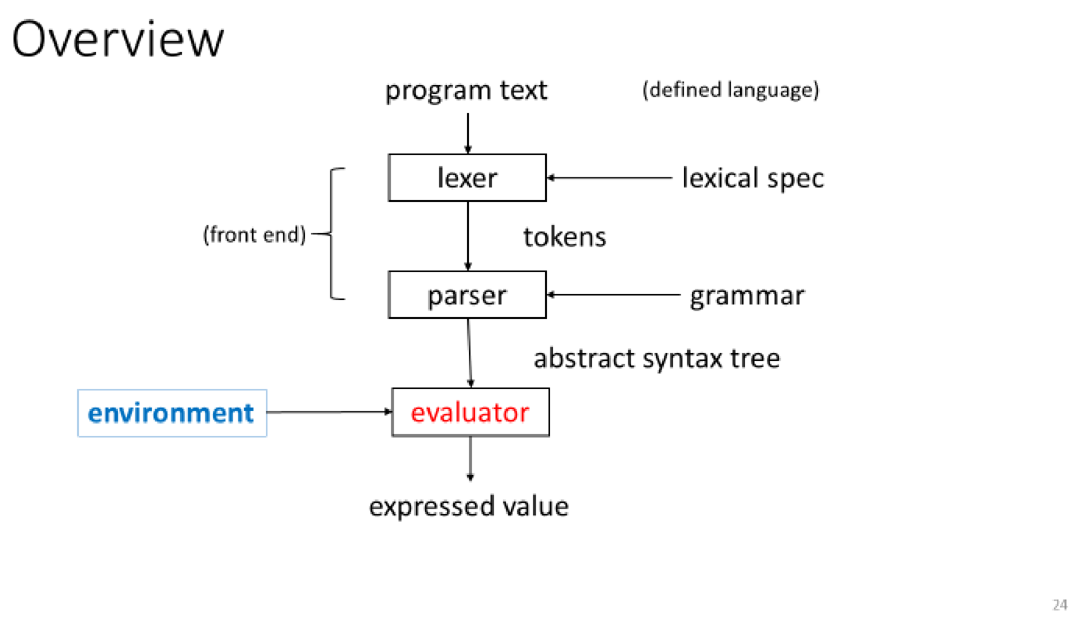


Figure 1: Overview of Execution Process

Evaluation and Grammar

You will use the Python ast library to do the work of the front end. The front-end work, especially the parser, is quite complex and we gain much by using this library. However, the ast parser uses Python grammar, so our defined language will have to be a subset of Python's grammar. The defining language will be Python.

The output of the ast parse function is an abstract syntax tree. Recall that each node in the tree represents a function linked to productions in the grammar. Your code will traverse this tree to execute the program that was parsed by the front end. It will use a data structure to store variable/value bindings that are established by certain statements in the source code, namely assignment and function definitions.

The "Its" language will support expressions, assignment, arithmetic operations (add, subtract, multiply, divide, mod), function definitions, and function calls. We will not bother with looping, although that would be easy enough to add later. Since we are adopting a subset of Python symbols and grammar, we'll use the nodes in the AST tree described below as a way of realizing the grammar of our language.

The AST Tree.

After obtaining the abstract parse tree, the interpreter evaluates the nodes of the tree. Consult the links provided for documentation on the types of nodes and their attributes. Your interpreter code will process each Node in the tree, extract the relevant data, and perform the relevant operation.

For example, the parse tree for this statement:

```
num1 = 5
```

Would look like this:

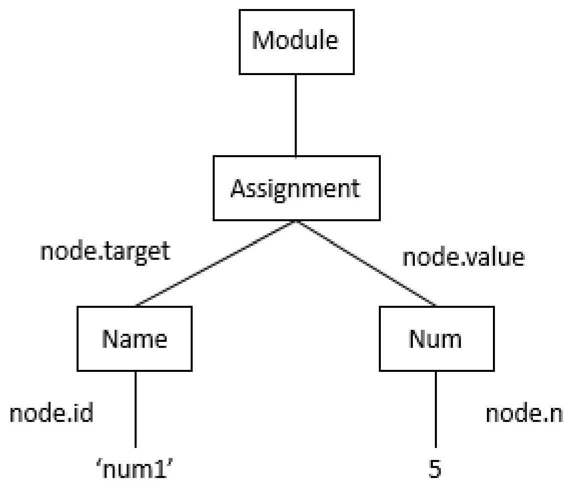


Figure 2: An AST tree

The tree is an ast.Module object. Its body attribute is a list of its children, which are ast.Node objects. The type of Node can be gotten by accessing the name attribute of its type:

```
type(node).__name__
```

These are the types of ast nodes you will need to evaluate. The relevant attributes are given in parentheses:

1. **Expr** (value) the "value" is an ast.Node object that represents the expression. An expression such as: 7 + 5 would be a BinOp object (see below).

2. **Assign** (targets, value) "targets" is a list. We will only use the first element, the variable that will be assigned to the "value". You will expand the environment to add this binding.

3. **BinOp** (left, op, right) "op" is the operator, "left" and "right" are the operands. We'll just do binary operators in our language: operand1 op operand2. The operations you must implement are: Add, Sub, Mult, Div, Mod.

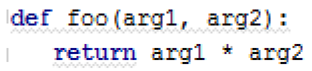
4. **FunctionDef** (name, args, body) "name"- the function name. "args"- an ast.argument object, "body"- a list of objects representing statements in the function body. The last statement is an ast.Return object. Note: out language will assume all functions return a value. We are not going to enforce this programmatically, but we will assume this.

The main "ingredients" you want to extract from this node are the function name, a list of the formal parameters and the body statements.

You will extend the environment with this pair: [name] [[args, body]]. You probably want the args and body in a list so it is one "value".

You may choose how you "package" the args and body. The information is encapsulated in ast objects. You can extract them when you retrieve this pair from the environment when the function is called. For example, the ast.argument object "args" attribute is a list of ast.arg object objects. You can extract the names there.

Example function definition:



```
def foo(arg1, arg2):
    return arg1 * arg2
```

The ast FunctionDef Node attributes for the code above are:

name- "foo"

args- an ast.argument object, where the names of the params will be an ast.Name nodes in the ast.arg objects.

body- [ast.Return] this would be the expression: arg1 * arg2

5. **Call** (func, args) "func"- the name of the function (an ast.Name node type), "args"- the values (or expressions) that will be bound to the formal params.

   You need to use the function name to lookup the formal parameters and the body that corresponds to this function in the environment.

   The parameters and values are "positional" so they correspond to the order they appear in their lists.

   To execute the function, create a local environment, add the bindings of params and arg values. Then use that environment to evaluate the body statements.

6. **Return** (value) the "value" is the expression to be evaluated and whose result is to be returned by the function.

7. **Name** (id) -- the "id" attribute is the variable's name. Lookup the variable in the environment and return the value, env.

8. **Num** (n) -- a number literal such as 9 This code is provided.

The Environment Data Structure

Another key part of this project is building the data structure for environments. You will implement an immutable structure that stores variable-value pairs in lists. Each addition to the structure results in a new pair of lists, with a reference to the previous structure.

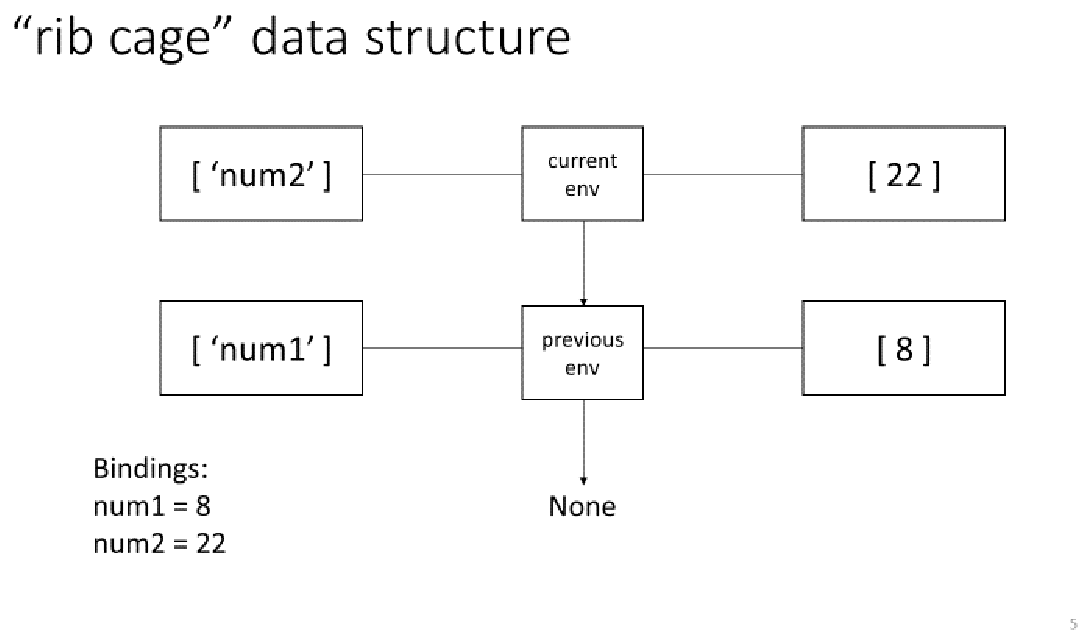


Figure 3: Rib Cage Implementation of an Environment

An environment is a reference to a linked list of var-value pairs. You will implement this as a GlobalEnv class in Python. Each instance of a GlobalEnv contains a bindings list, a values list, and a reference to a previous instance of a GlobalEnv object.

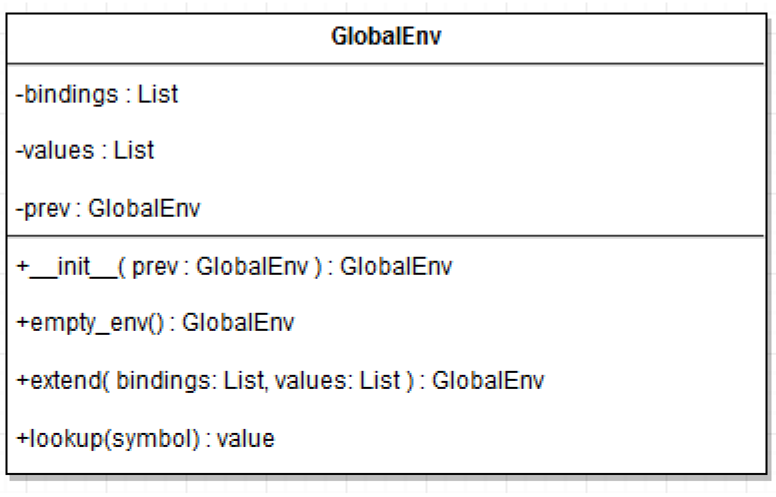


Figure 4: GlobalEnv Class Diagram

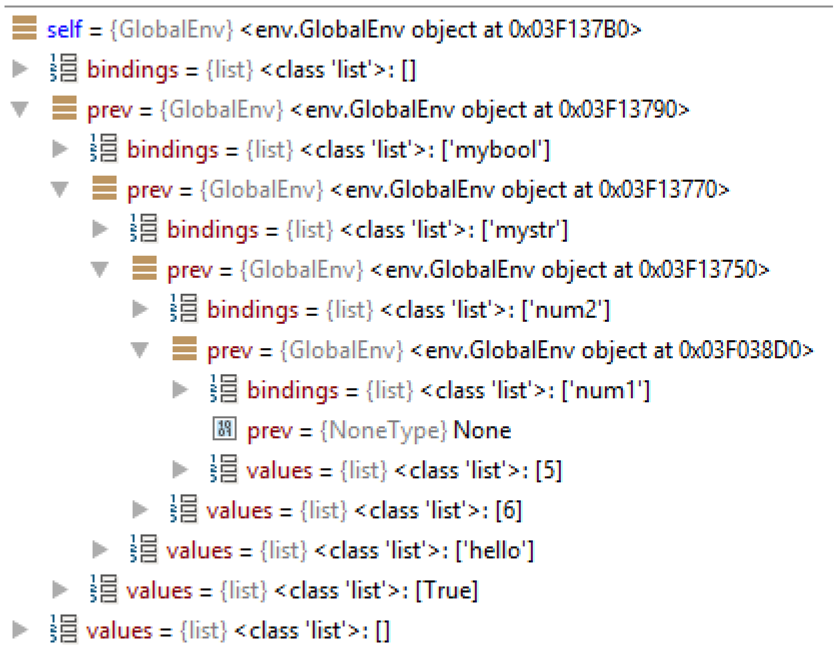

```
self = {GlobalEnv} <env.GlobalEnv object at 0x03F137B0>
▶ bindings = {list} <class 'list'>: []
▼ prev = {GlobalEnv} <env.GlobalEnv object at 0x03F13790>
    ▶ bindings = {list} <class 'list'>: ['mybool']
    ▼ prev = {GlobalEnv} <env.GlobalEnv object at 0x03F13770>
        ▶ bindings = {list} <class 'list'>: ['mystr']
        ▼ prev = {GlobalEnv} <env.GlobalEnv object at 0x03F13750>
            ▶ bindings = {list} <class 'list'>: ['num2']
            ▼ prev = {GlobalEnv} <env.GlobalEnv object at 0x03F038D0>
                ▶ bindings = {list} <class 'list'>: ['num1']
                  prev = {NoneType} None
                ▶ values = {list} <class 'list'>: [5]
            ▶ values = {list} <class 'list'>: [6]
        ▶ values = {list} <class 'list'>: ['hello']
    ▶ values = {list} <class 'list'>: [True]
▶ values = {list} <class 'list'>: []
```

Figure 5: Structure of Rib Cage

The structure in figure 4 was created by creating an empty GlobalEnv, then extending it for each of the following statements:

```
num1 = 5
num2 = 6
mystr='hello'
mybool=True
```

Now, suppose you encounter the following statement:

```
res = num1 + num2
```

You would lookup the values for the two variables in the environment, getting 5 and 6. You would then perform addition yielding 11. Next, you would extend the environment, creating a binding of ‘res’ and 11.

Note: make empty_env() a static method (use the @staticmethod decorator).

Local environments.

A function call creates the need for a local environment. The function’s parameters get bound to values passed in in a local environment. Then, the body of the function is executed in the local environment. Note that a local environment keeps a pointer to a global environment.

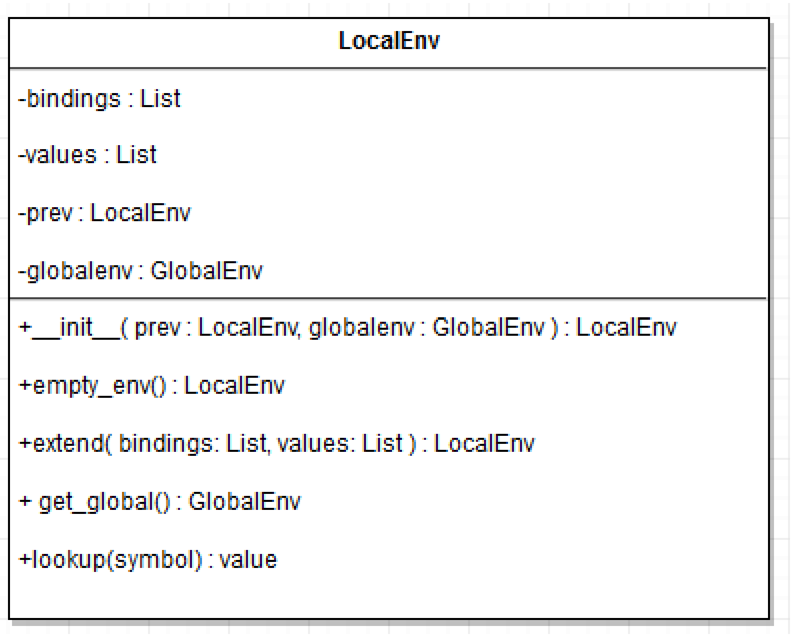

LocalEnv
-bindings : List
-values : List
-prev : LocalEnv
-globalenv : GlobalEnv
+__init__(prev : LocalEnv, globalenv : GlobalEnv) : LocalEnv
+empty_env() : LocalEnv
+extend(bindings: List, values: List) : LocalEnv
+ get_global() : GlobalEnv
+lookup(symbol) : value

Figure 6: LocalEnv Class Diagram

A local environment object is created upon processing a function class node. The function arguments are bound to the values passed in in this environment and the body of the function, a list of statements, is executed in this environment. The only difference between a LocalEnv and a GlobalEnv is that the localEnv has a pointer to the GlobalEnv.

Project Requirements and Structure

After you download and unzip the starter kit you should take some time reviewing the provided code and comments.

The files that comprise this project are:

- interpreter.py – Contains the interpreter code. Defines the eval_tree function.
- env.py – Contains the class definitions for GlobalEnv and LocalEnv.
- interpreter_main.py – (provided) Creates the ast tree and runs the interpreter.

Some testing code is provided in interpreter_main. The interpreter module must define the following two functions:

eval_tree (tree) Takes the AST tree object, evaluates and executes the nodes in the tree and returns a result, the expressed value of the program that was executed. The default return is 0.

eval_node (node, env) Takes an ast Node object and an environment (GlobalEnv or LocalEnv) object. It returns both a numeric value and an environment object (GlobalEnv or LocalEnv).

Notes: create an empty GlobalEnv object at the top of the module. There is one global environment in any program. There may be local environments created as needed, but you need a global environment. Actually, you will want to have two global variables: the global environment and the result that is returned at the end of the eval_tree function.

You may use loops in your code. Use a loop to visit the top level child nodes of the tree root (the ast.Module object). Call eval_node on each node, passing the environment along.

Your submission files and all functions must be named exactly as above to receive credit for this assignment. The following section provides a description of the modules and functions that you must define and implement in each file. Remember that you must define these signatures exactly to receive any credit for this assignment.

You may use any Python data structures and iterative constructs to implement the interpreter. You must use the environment data structures GlobalEnv and LocalEnv in your evaluation. You may only obtain data from the ast Node objects. You must write the code that executes or handles the data you obtain from the node.

Testing

Develop your code incrementally, testing as you make progress. Don't write a lot of code and then test it. Remember that we will not assess code that does not run.

Create the environment objects and test them in isolation from the rest of the code. When you are sure they work, integrate them with the interpreter code.

Start with the simplest of statements and walk through the parse tree. Build the simplest node evaluation code first and test it. Add another node and test that. Build up to the more complex nodes. Leave the function calls for later. Remember, you only need to extend the environment for assignment and function definitions. You extend a local environment for function calls. You use the environment to lookup variable and function names.

Write helper functions if you need to. Test them in isolation. No matter how simple, test it before you rely on it! Note that this takes time, but if you want full credit and a piece of working software, this is the best way to achieve that end.

When you are confident a function works, test it in integration with other functions it is called by or that it calls.

Some test source code strings are provided in the interpreter_main.py file.

Submission

Run the submit.py script to ‘zip’ up your files and submit them to the Moodle assignment before the due date/time.