CptS355 - Assignment 5
Static Scoping PostScript Interpreter (SSPS)
Spring 2020

Assigned: Monday, April 20, 2020

Due: Friday, May 1st, 2020

Weight: Assignment 5 will count for 4% of your course grade.

Your solutions to the assignment problems are to be your own work. Refer to the course academic integrity statement in the syllabus.

The Postscript is a dynamically scoped language. In this assignment, you will be modifying your SPS interpreter (Assignment 4 - Simple Post Script Interpreter) to handle a slightly different language which supports static scoping. We will call this language Scoped Simple PostScript - SSPS. **SSPS has no `dict`, `begin` or `end` operations. Instead, each time a postscript function is called a new dictionary is automatically pushed on the dictionary stack. And when the function execution is complete, this dictionary will be popped out of the stack.** The dictionary must be able to hold an arbitrary number of names.

Turning in your assignment

All code should be developed in the file called **HW5.py**.

At the top of the file in a comment, please include your name and the names of the students with whom you discussed any of the problems in this homework. This is an individual assignment and the final writing in the submitted file should be ***solely yours***. You may NOT copy another student's code or work together on writing code. You may not copy code from the web, or anything else that lets you avoid solving the problems for yourself.

You may turn in your assignment up to 3 times. Only the last one submitted will be graded.

Grading

The assignment will be marked for good programming style (appropriate algorithms, good indentation and appropriate comments -- refer to the Python style guide) -- as well as thoroughness of testing and clean and correct execution. You will lose points if you don't (1) provide test functions, (2) explain your code with appropriate comments, and (3) follow a good programming style. Make sure that debugging code is removed from the required functions themselves (i.e. no print statements other than those that print the final output).

Project Description

You will be modifying your SPS interpreter (Assignment 4 - Simple Post Script Interpreter) to handle a slightly different language which we will call Scoped Simple PostScript - SSPS. SSPS has no `dict`, `begin` or `end` operations. Instead, each time a postscript function is called a new dictionary is automatically pushed on the dictionary stack.

Compared to your SPS interpreter function, the SPSS interpreter will take an addition argument whose value will be either the string "static" or the string "dynamic", to indicate whether it should behave using static scope rules or dynamic scope rules. For example:

```
# This is the recursive function to interpret a given code array.
# code is a code array; scope is a string (either "static" or  "dynamic")
def interpretSPS(code,scope):
    pass

# add an argument to interpret to specify the scoping rule that will be
# applied
# s is a string; scope is a string (either "static" or "dynamic")
def interpreter(s,scope):
    interpretSPS(parse(tokenize(s)),scope)
```

Since `if`, `ifelse`, `repeat`, and `forall` operators also call `interpretSPS` (to execute their code arrays), you should also add the "scope" argument to these operator implementations.

To interpret code using static scoping rules call `interpreter(s,'static')` and to interpret code "s" using dynamic scoping rules call `interpreter(s,'dynamic')`.

- Each time a **postscript function is about to be called push a new dictionary** and when a postscript **function is about to return pop the dictionary**.
- To implement **static scope rules** you need a static chain which is the set of dictionaries visited by following the static links (also called the access links) we discussed in class. You already have a stack of dictionaries, you don't need to have explicit dynamic links. The dynamic chain is implicit in the order of the dictionaries in the list. To search for a declaration you search from the top of the stack.
- ***How can you implement the static chain?*** I suggest making the stack be a stack of tuples (instead of just a stack of dictionaries) where each tuple contains an integer index and a dictionary. The integer index represents the static link that tells you the position in the list of the (dictionary, static-link-index) tuple for the parent scope.
- ***Where do static-link values come from?*** As we saw in class, at the point when a function is called, the static link in the new stack entry needs to be set to point to the stack entry where the function's definition was found. (Note that with the stack being a list, this "pointer" is just an index in the list.) So when calling a postscript function you create a new dictionary automatically and what you push on the dictionary stack is a pair : (index-of-definition's stack entry, dictionary).
    - Hint: In an effective implementation this should all take only a handful of lines of new code but it is tricky to get all the pieces right. My advice is think more, write less for this part of the project.
    - As discussed in class, variable lookups using static scope rules proceed by looking in the current dictionary at the top of the dictionary stack and then following the static-link fields to other dictionaries (instead of just looking at the dictionaries in order).
        - Note: In Assignment 3, you already implemented the lookup function using static scoping rule, where you search the dictionaries following the index links in the tuples (i.e., following the static links).

- Using **dynamic scope rules**, the SPSS interpreter will behave very much like SPS except that it won't support `dict`, `begin` or `end` operations in programs (indeed, there will be no implementation of these operations in SSPS).
    - I suggest you to use the same dictstack for both static and dynamic scope implementations.
    - When the scoping rule is dynamic, the lookup should just look at the dictionaries on the `dictstack` starting from top (ignoring the static links).

Additional Notes:

You **should not have two separate interpret functions** for the static and dynamic scoping implementations. Please include the scoping rule as an argument as suggested above and customize the interpretation if static scoping is specified.
For each test input we will test your code for static and dynamic scoping as follows (assume "input1" is the SPS code that we will interpret):

```
interpreter(input1, "static")
interpreter(input1, "dynamic")
```

- As explained before, your interpreter should store 2-tuples in `dictstack` where first value in the tuple is the *dictionary* and second value is the *static link index*.
- A new tuple will be pushed onto the `dictstack` whenever a function call is made.
- *"Static link index"* is the index of the dictionary (in the `dictstack`) where the function is defined. I will discuss the algorithm for finding this in class.
- Change your `lookup` function for static scoping. If the scoping rule is dynamic, perform lookup by searching the activation record (tuples) top to down. If it is static, use static links for the search.
- When a function call returns, remember to pop the tuple for that function call from the `dictstack`.
- Change your `stack` operator implementation as explained below.

Output of the Interpreter

Whenever the stack operation is executed, the contents of the operand and dictionary stacks are printed. (Remember that stack only printed the contents of the operand stack in Assignment-4)

- Print a line containing "==========" to separate the stack from what has come before.
- Print the operand stack one value per line; print the top-of-stack element first.
- Print a line containing "==========" to separate the stack from the dictionary stack.
- Print the contents of the dictionary stack, beginning with the top-of-stack dictionary one name and value per line with a line containing `{---- m---- n ----}` before each dictionary. *m* is the index that will identify the dictionary printed (dictionary index) and *n* is the index that

represents the static link for the dictionary printed (in the case that static scoping is being used). Please see below for an example.

- Print a line containing "==========" to separate the dictionary stack from any subsequent output.

Remember please the difference between a dictionary and a dictionary entry.

What if my SPS interpreter didn't work correctly?

You will need to fix it so it works. You can visit with the TA or me for help.

How can I tell if my static scoping code is working correctly?

You will have to create some test cases for which you can predict the correct answers. Below are couple examples for initial tests. Please create additional tests (at least 3) to test your interpreter.
When we grade your assignment, we will compare the outputs of your interpreter prints.

```
/x 4 def
/g { x stack } def
/f { /x 7 def g } def
f
```

The above SPS code will leave **7 on the stack using dynamic scoping and 4 using static scoping**. The output from the **`stack`** operator in function **g** would look like this when using static scoping:

```
==========
4
==========
----2----0----
----1----0----
/x   7
----0----0----
/x   4
/g   ['x', 'stack']
/f   ['/x', 7, 'def', 'g']
==========
```

And using dynamic scoping, the output from the **`stack`** operator will look like the following:

```
==========
7
==========
----2----0----
----1----0----
/x   7
----0----0----
/x   4
/g   ['x', 'stack']
/f   ['/x', 7, 'def', 'g']
==========
```

For the values of *m* and *n* you may use anything you like as along as it is possible to tell where the static links point. For printing code array values, I suggest using [ ] around the values in the array.

Additional Test Cases:

2)

```
testinput2 = """
/x 4 def
[1 1 1] 1 [2 3] putinterval /arr exch def
/g { x stack } def
/f { 0 arr {7 mul add} forall /x exch def g } def
f
"""
```

Expected Output

Using static scoping

```
Static
==============
4
==============
----2----0----
----1----0----
/x   42
----0----0----
/x   4
/arr   [1, 2, 3]
/g   {'codearray': ['x', 'stack']}
/f   {'codearray': [0, 'arr', {'codearray': [7, 'mul', 'add']}, 'forall',
'/x', 'exch', 'def', 'g']}
==============
```

Using dynamic scoping

```
Dynamic
==============
42
==============
----2----0----
----1----0----
/x   42
----0----0----
/x   4
/arr   [1, 2, 3]
/g   {'codearray': ['x', 'stack']}
/f   {'codearray': [0, 'arr', {'codearray': [7, 'mul', 'add']}, 'forall',
'/x', 'exch', 'def', 'g']}
==============
```

3)

```
testinput3 = """
/m 50 def
/n 100 def
/egg1 {/m 25 def n} def
/chic
   { /n 1 def
   /egg2 { n stack} def
   m  n
   egg1
   egg2
 } def
n
chic
"""
```

Expected Output

Using static scoping

```
Static
==============
1
100
1
50
100
==============
----2----1----
----1----0----
/n   1
/egg2   {'codearray': ['n', 'stack']}
----0----0----
/m   50
/n   100
/egg1   {'codearray': ['/m', 25, 'def', 'n']}
/chic   {'codearray': ['/n', 1, 'def', '/egg2', {'codearray': ['n',
'stack']}, 'def', 'm', 'n', 'egg1', 'egg2']}
==============
```

Using dynamic scoping

```
Dynamic
==============
1
1
1
50
100
==============
----2----1----
----1----0----
/n   1
/egg2   {'codearray': ['n', 'stack']}
```

```
----0----0----
/m   50
/n   100
/egg1   {'codearray': ['/m', 25, 'def', 'n']}
/chic   {'codearray': ['/n', 1, 'def', '/egg2', {'codearray': ['n',
'stack']}, 'def', 'm', 'n', 'egg1', 'egg2']}
==============
```

4)

```
testinput4 = """
/x 10 def
/A { x } def
/C { /x 40 def A stack } def
/B { /x 30 def /A { x } def C } def
B
"""
```

Expected Output

Using static scoping

```
Static
==============
10
==============
----2----0----
/x   40
----1----0----
/x   30
/A   {'codearray': ['x']}
----0----0----
/x   10
/A   {'codearray': ['x']}
/C   {'codearray': ['/x', 40, 'def', 'A', 'stack']}
/B   {'codearray': ['/x', 30, 'def', '/A', {'codearray': ['x']}, 'def', 'C']}
==============
```

Using dynamic scoping

```
Dynamic
==============
40
==============
----2----0----
/x   40
----1----0----
/x   30
/A   {'codearray': ['x']}
----0----0----
/x   10
/A   {'codearray': ['x']}
/C   {'codearray': ['/x', 40, 'def', 'A', 'stack']}
/B   {'codearray': ['/x', 30, 'def', '/A', {'codearray': ['x']}, 'def', 'C']}
==============
```

5)

```
testinput5 = """
/x 10 def
/n 5  def
/A { 0  n {x add} repeat} def
/C { /n 3 def /x 40 def A stack } def
/B { /x 30 def /A { x } def C } def
B
"""
```

Expected Output

Using static scoping

```
Static
==============
50
==============
----2----0----
/n   3
/x   40
----1----0----
/x   30
/A   {'codearray': ['x']}
----0----0----
/x   10
/n   5
/A   {'codearray': [0, 'n', {'codearray': ['x', 'add']}, 'repeat']}
/C   {'codearray': ['/n', 3, 'def', '/x', 40, 'def', 'A', 'stack']}
/B   {'codearray': ['/x', 30, 'def', '/A', {'codearray': ['x']}, 'def', 'C']}
==============
```

Using dynamic scoping

```
Dynamic
==============
40
==============
----2----0----
/n   3
/x   40
----1----0----
/x   30
/A   {'codearray': ['x']}
----0----0----
/x   10
/n   5
/A   {'codearray': [0, 'n', {'codearray': ['x', 'add']}, 'repeat']}
/C   {'codearray': ['/n', 3, 'def', '/x', 40, 'def', 'A', 'stack']}
/B   {'codearray': ['/x', 30, 'def', '/A', {'codearray': ['x']}, 'def', 'C']}
==============
```

6)

```
testinput6 = """
/out true def
/xand { true eq {pop false} {true eq { false } { true } ifelse} ifelse dup /x
        exch def stack} def
/myput { out dup /x exch def xand } def
/f { /out false def myput } def
false f
"""
```

Expected Output

Using static scoping

```
Static
==============
False
==============
----3----0----
/x   False
----2----0----
/x   True
----1----0----
/out   False
----0----0----
/out   True
/xand   {'codearray': [True, 'eq', {'codearray': ['pop', False]},
{'codearray': [True, 'eq', {'codearray': [False]}, {'codearray': [True]},
'ifelse']}, 'ifelse', 'dup', '/x', 'exch', 'def', 'stack']}
/myput   {'codearray': ['out', 'dup', '/x', 'exch', 'def', 'xand']}
/f   {'codearray': ['/out', False, 'def', 'myput']}
==============
```

Using dynamic scoping

```
Dynamic
==============
True
==============
----3----0----
/x   True
----2----0----
/x   False
----1----0----
/out   False
----0----0----
/out   True
/xand   {'codearray': [True, 'eq', {'codearray': ['pop', False]},
{'codearray': [True, 'eq', {'codearray': [False]}, {'codearray': [True]},
'ifelse']}, 'ifelse', 'dup', '/x', 'exch', 'def', 'stack']}
/myput   {'codearray': ['out', 'dup', '/x', 'exch', 'def', 'xand']}
/f   {'codearray': ['/out', False, 'def', 'myput']}
==============
```

7)

```
testinput7 = """
/x [1 2 3 4] def
/A { x length } def
/C { /x [10 20 30 40 50 60] def A stack } def
/B { /x [6 7 8 9] def /A { x 0 get} def C } def
B
"""
```

Expected Output

Using static scoping

```
Static
==============
4
==============
----2----0----
/x   [10, 20, 30, 40, 50, 60]
----1----0----
/x   [6, 7, 8, 9]
/A   {'codearray': ['x', 0, 'get']}
----0----0----
/x   [1, 2, 3, 4]
/A   {'codearray': ['x', 'length']}
/C   {'codearray': ['/x', [10, 20, 30, 40, 50, 60], 'def', 'A', 'stack']}
/B   {'codearray': ['/x', [6, 7, 8, 9], 'def', '/A', {'codearray': ['x', 0,
'get']}, 'def', 'C']}
==============
```

Using dynamic scoping

```
Dynamic
==============
10
==============
----2----0----
/x   [10, 20, 30, 40, 50, 60]
----1----0----
/x   [6, 7, 8, 9]
/A   {'codearray': ['x', 0, 'get']}
----0----0----
/x   [1, 2, 3, 4]
/A   {'codearray': ['x', 'length']}
/C   {'codearray': ['/x', [10, 20, 30, 40, 50, 60], 'def', 'A', 'stack']}
/B   {'codearray': ['/x', [6, 7, 8, 9], 'def', '/A', {'codearray': ['x', 0,
'get']}, 'def', 'C']}
==============
```

8)

```
testinput8 = """
[0 1 2 3 4 5 6 7 8 9 10] 3 4 getinterval /x exch def
/a 10 def
/A { x length } def
/C { /x [a 2 mul a 3 mul dup a 4 mul] def A  a x stack } def
/B { /x [6 7 8 9] def /A { x 0 get} def /a 5 def C } def
B
"""
```

Expected Output

Using static scoping

```
Static
==============
[20, 30, 30, 40]
10
4
==============
----2----0----
/x   [20, 30, 30, 40]
----1----0----
/x   [6, 7, 8, 9]
/A   {'codearray': ['x', 0, 'get']}
/a   5
----0----0----
/x   [3, 4, 5, 6]
/a   10
/A   {'codearray': ['x', 'length']}
/C   {'codearray': ['/x', ['a', 2, 'mul', 'a', 3, 'mul', 'dup', 'a', 4,
'mul'], 'def', 'A', 'a', 'x', 'stack']}
/B   {'codearray': ['/x', [6, 7, 8, 9], 'def', '/A', {'codearray': ['x', 0,
'get']}, 'def', '/a', 5, 'def', 'C']}
==============
```

Using dynamic scoping

```
Dynamic
==============
[10, 15, 15, 20]
5
10
==============
----2----0----
/x   [10, 15, 15, 20]
----1----0----
/x   [6, 7, 8, 9]
/A   {'codearray': ['x', 0, 'get']}
/a   5
----0----0----
/x   [3, 4, 5, 6]
/a   10
/A   {'codearray': ['x', 'length']}
/C   {'codearray': ['/x', ['a', 2, 'mul', 'a', 3, 'mul', 'dup', 'a', 4,
'mul'], 'def', 'A', 'a', 'x', 'stack']}
```

```
/B   {'codearray': ['/x', [6, 7, 8, 9], 'def', '/A', {'codearray': ['x', 0,
'get']}, 'def', '/a', 5, 'def', 'C']}
==============
```